Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Laboratorio de ICARO

DISEÑO, IMPLEMENTACIÓN Y ADECUACIÓN DE UNA HERRAMIENTA EDUCATIVA BASADA EN MINIX PARA CURSOS DE SISTEMAS OPERATIVOS

Presentado ante la ilustre
Universidad Central de Venezuela
Por los Bachilleres:

Dumar De León
C.I.:18.465.540
E-mail:dumardeleon@gmail.com

José España
C.I.:16.115.715
E-mail:espana.jose.manuel@gmail.com

Tutor: David Pérez
Caracas, Marzo de 2011

Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Laboratorio de ICARO

ACTA DEL VEREDICTO

Quienes suscriben, Miembros del Jurado designado por el Consejo de la Escuela de Computación para examinar el Trabajo Especial de Grado, presentado por los Bachilleres Dumar De León C.I.:18.465.540 y José España C.I.: 16.115.715, con el título **"DISEÑO, IMPLEMENTACIÓN Y ADECUACIÓN DE UNA HERRAMIENTA EDUCATIVA BASADA EN MINIX PARA CURSOS DE SISTEMAS OPERATIVOS"**, a los fines de cumplir con el requisito legal para optar al título de Licenciado en Computación, dejan constancia de lo siguiente:

Leído el trabajo por cada uno de los Miembros del Jurado, se fijó el día 8 de Abril de 2011, a las 3:00 PM, para que sus autores lo defendieran en forma pública, en Laboratorio de Internet II, lo cual estos realizaron mediante una exposición oral de su contenido, y luego respondieron satisfactoriamente a las preguntas que les fueron formuladas por el Jurado, todo ello conforme a lo dispuesto en la Ley de Universidades y demás normativas vigentes de la Universidad Central de Venezuela. Finalizada la defensa pública del Trabajo Especial de Grado, el jurado decidió aprobarlo.

En fe de lo cual se levanta la presente acta, en Caracas el 8 de Abril de 2011, dejándose también constancia de que actuó como Coordinador del Jurado el Profesor Tutor David Pérez

Prof. David Pérez
(Tutor)

Prof. Carlos Acosta

Prof. Jaime Parada

RESUMEN

Título:

DISEÑO, IMPLEMENTACIÓN Y ADECUACIÓN DE UNA HERRAMIENTA EDUCATIVA BASADA EN MINIX PARA CURSOS DE SISTEMAS OPERATIVOS

Autor(es):

Dumar De León y José España

Tutor:

Prof. David Pérez

En el siguiente Trabajo Especial de Grado se plantea la inserción del sistema operativo Minix 3 al curso ofertado por la Escuela de Computación de la UCV con el objetivo de solventar la poca sincronización entre la planificación teórica-práctica existente. Es decir, no existe ninguna metodología que permita reforzar ambas dinámicas durante la ejecución del mismo. La solución propuesta es adecuar el sistema operativo Minix 3 a partir de un conjunto de siete laboratorios que abarcan tópicos asociados a los temas impartidos por el curso de Sistemas Operativos actual. Los laboratorios tienen los siguientes títulos: Instalación de Minix y entorno de desarrollo, Introducción a Minix 3, Estudio del proceso de arranque, Implementación de un intérprete de comandos, Implementación de llamadas al sistema, Implementación de semáforos y Modificación del planificador de procesos. De cada laboratorio se desarrollo la estructura y solución, dejando una sólida documentación en distintos medios (documentos, implementación del código necesario y videos tutoriales). Por último, se adecuo una aplicación para el manejo de proyectos (wiki) la cual apoya al curso y permite integrar toda la información recopilada, tanto para el grupo docente como para los estudiantes. Al culminar este Trabajo Especial de Grado se logró completar los objetivos planteados desde el inicio a través de las soluciones anteriormente descritas, es por ello que se desea que el trabajo realizado sea tomado en consideración por la Escuela de Computación para impartir cursos futuros; dándole a mismo la didáctica de los cursos de las principales universidades a nivel mundial.

Palabras Claves: Sistema Operativo Instruccional, Minix 3, Laboratorio, implementación, documentación.

Tabla de contenido

Índice de Figuras

Índice de Tablas

1 Introducción

Las primeras computadoras que surgieron no poseían sistemas operativos, cada programa necesitaba la especificación completa del hardware y sus propios controladores de dispositivos periféricos para funcionar correctamente y desempeñar sus tareas; además, la creciente complejidad del hardware y los programas de usuarios crearon la necesidad de un software que se encargara de los inconvenientes antes mencionados, en consecuencia surgieron los sistemas operativos. Los cuales se encargan, a groso modo, de proporcionar a las aplicaciones una interfaz para manejar el hardware de un computador, convirtiéndose así en una herramienta por excelencia debido al uso masivo por parte de los usuarios de los computadores. Es por eso que el diseño, desarrollo e investigación de los sistemas operativos han jugado un rol principal en Ciencias de la Computación.

Hay que tomar en cuenta que, implementar un sistema operativo que permita manejar el hardware, administrar los recursos computacionales de forma eficiente, habilitar la ejecución de procesos o aplicaciones independientes a su desarrollo como los programas de usuario, no es una tarea trivial. La implementación de un sistema operativo es una tarea sumamente complicada que requiere de ciertas pericias, talentos y buenas prácticas en el área de la programación. Además, de una gran inversión en capital humano y sobre ellos un excepcional conocimiento en el área. Dada la complejidad de los sistemas operativos, estudiar el diseño e implementación de los sistemas operativos modernos es una tarea ardua y difícil, pero necesaria.

El conocimiento de los conceptos de sistemas operativos son considerados importantes en la mayoría de los pensum académicos referentes a cualquier carrera profesional de computación. En estos cursos se imparte los conceptos básicos de diseño, implementación y funcionalidad de los mismos. Existen dos enfoques al impartir dichos conocimientos, el enfoque teórico-abstracto y el práctico. El enfoque netamente práctico es considerado complejo debido a lo señalado con anterioridad. Por ende, es necesaria una herramienta que mitigue dicha complejidad. Es por esto que nacen lo sistemas operativos instruccionales.

Un sistema operativo instruccional es una herramienta educativa que permite el estudio y comprensión de forma sencilla el diseño, desarrollo e implementación de los principales conceptos y funcionalidades de un sistema operativo moderno. Este software es un sistema operativo sencillo y en la mayoría de las veces incompleto, careciendo de ciertas piezas las cuales los estudiantes se encargan de implementar.

Los sistemas operativos instruccionales se desarrollan bajo dos paradigmas, realismo y simulación. El primero (realismo) nos acerca a lo moderno, pero también a su complejidad. El segundo nos aleja de la realidad, pero nos añade la sencillez y la facilidad durante el desarrollo de proyectos.

Esta investigación pretende proponer un sistema operativo instruccional como apoyo a la enseñanza del curso de Sistemas Operativos de la Universidad Central de Venezuela de la Facultad de Ciencias de la Escuela de Computación.

1.1 *Planteamiento del problema*

Los Sistemas Operativos constituyen un tópico importante en la enseñanza en Ciencias de la Computación y acorde a la metodología pedagógica utilizada, impartir este conocimiento puede resultar una tarea compleja. Esta dificultad recae en la abstracción del contenido. Por ende, es necesaria una herramienta pedagógica que permita unificar los espacios teóricos-prácticos para mitigar la complejidad de instruir sobre este tópico. Dicha herramienta también debe brindar un entorno que permita al estudiante interactuar con los conceptos básicos de Sistemas Operativos. Como se dijo con anterioridad esta pieza de software educativa se conoce como Sistemas Operativos Instruccionales.

En la actualidad, los cursos de Sistemas Operativos de la Universidad Central de Venezuela no cuentan con una herramienta pedagógica que unifique y permita llevar una planificación consistente entre los tópicos dictados en las clases de teoría y los desarrollados en los laboratorios. En síntesis, el problema se refleja en la diacronía de la planificación entre la teoría y la práctica, es decir, no existe ninguna metodología que permita reforzar ambas dinámicas durante la ejecución del curso. Es importante destacar que durante los laboratorios y proyectos de la materia se utilizan distintas herramientas para generar conocimiento, donde los estudiantes deben aprender y manejar múltiples ambientes, plataformas, arquitecturas, etc., lo que trae como consecuencia que el curso no se centre netamente en explicar los tópicos de sistemas operativos, ya que debe invertir parte del tiempo en explicar el funcionamiento de cada ambiente. En conversaciones con el grupo docente se ha establecido la necesidad de utilizar herramientas con la finalidad de mejorar el desempeño estudiantil y la didáctica de los docentes. Con todos estos lineamientos discutidos surge la siguiente interrogante: **¿Es posible incorporar una herramienta educativa que permita ser adaptada al curriculum académico de los cursos de Sistemas Operativos para reforzar los tópicos dictados en las clases teóricas?**

1.2 *Objetivos*

A continuación se describen el objetivo general y los objetivos específicos planteados para este trabajo especial de grado.

1.2.1 Objetivo general

Adaptar al Sistema Operativo Instruccional Minix versión 3 en cuanto al contenido y objetivos de los cursos de Sistemas Operativos de la Escuela de Computación en la Universidad Central de Venezuela, con el fin de optimizar el proceso de aprendizaje de los estudiantes.

1.2.2 Objetivos específicos

- Definir la estructura de los laboratorios a impartir en el curso de SO.
- Adecuar el SOI Minix versión 3 a los laboratorios definidos con anterioridad.
- Determinar las actividades docentes a realizar en cada uno de los laboratorios soportados por el SOI Minix versión 3.
- Desarrollar las plantillas, solución y documentación de las actividades planteadas.
- Realizar pruebas de correctitud sobre el SOI Minix versión 3 y sus componentes.
- Documentar el proceso de adecuación de la herramienta.

1.3 *Justificación*

Los Sistemas Operativos Instruccionales son ampliamente usados en varias universidades, como Stanford, MIT, Berkeley, Harvard entre otras. Basados en este hecho surgen las preguntas: ¿Por qué?, esto se debe a que se ha comprobado la eficacia como herramienta pedagógica, ya que permite a los estudiantes implementar partes estratégicas de un sistema operativo, siendo una buena práctica de estudio; ¿Cómo? mejorando los aspectos didácticos relacionados a los cursos tradicionales de Sistemas Operativos, ofreciendo a docentes y estudiantes una herramienta capaz de estructurar los aspectos prácticos relacionados al curso. Estas herramientas instruccionales permiten enseñar los conceptos más importantes de los sistemas operativos mediante la modificación de un pequeño sistema operativo. Además, provee la unificación de la práctica en una sola herramienta. Esto se debe a que los sistemas operativos instruccionales proveen de una serie proyectos que pretenden enseñar cada tópico relevante del pensum de estudio del curso.

Específicamente, en la Universidad Central de Venezuela, en la materia Sistemas Operativos de la Escuela de Computación, no se cuenta con una herramienta educativa de este estilo. Asimismo, en sus espacios prácticos no existe una estructura lineal que permita mejorar la comprensión del alumnado sobre los aspectos básicos de sistemas operativos. Por todas las razones anteriormente expuestas, este Trabajo Especial de Grado busca suplir este déficit. Para lograrlo se plantea adaptar un sistema operativo instruccional que permita ser fuente generadora de conocimientos teóricos-prácticos.

1.4 *Distribución del documento*

El contenido de cada uno de los capítulos que integran este documento es el siguiente:

- ✓ **Capítulo 1: Introducción.** Se esboza el contexto de la investigación, permitiéndole al lector ubicarse rápidamente en los temas a tratar y la finalidad de la misma. Asimismo se expone la justificación, objetivos y planteamiento del problema de esta investigación.
- ✓ **Capítulo 2: Marco teórico.** Se introducen los principales conceptos asociados a sistemas operativos. Además, en este capítulo se abarcan las definiciones de un sistema operativo

instruccional, las herramientas de virtualización, simulación y emulación; los sistemas operativos instruccionales más usados (1), y una tabla comparativa de los mismos. Luego se explica los motivos por los cuales el sistema operativo Minix 3 fue elegido para ser adecuado al curso de Sistemas Operativos de la UCV.

- ✓ **Capítulo 3: Adecuación de Minix 3 a la UCV.** En esta sección serán descritos todos los laboratorios propuestos para la adecuación de Minix 3 al curso de Sistemas Operativos de la Escuela de Computación de la Universidad Central de Venezuela. Para cada laboratorio propuesto se desarrollan los siguientes puntos: motivación, objetivos, grupo docente, estudiantes, entregables, duración, terminando con la documentación y ayuda. Luego se expone una simple tabla con la planificación de los laboratorios por semana propuesta por esta investigación.
- ✓ **Capítulo 4: Herramientas de desarrollo.** Se describen los sistemas operativos y aplicaciones utilizadas para la adecuación del SOI Minix 3 al curso de Sistemas Operativos de pregrado de la Universidad Central de Venezuela. Para concluir este capítulo se muestra la metodología utilizada para el desarrollo de la herramienta educativa, especificando la misma por cada laboratorio propuesto.
- ✓ **Capítulo 5: Instalación de Minix y entorno de desarrollo.** Este laboratorio esta diseñado para sentar las bases de las herramientas necesarias para desarrollar a lo largo del curso todos los laboratorios propuestos por el grupo docente. Este es un punto clave ya que permite engranar todas las aplicaciones y el sistema operativo instruccional Minix 3 para facilitar en gran medida el desarrollo de la implementación de los laboratorios. Serán descritos los pasos que se dieron para la instalación de Minix versión 3.1.6, así como también para armar y configurar el entorno de desarrollo.
- ✓ **Capítulo 6: Introducción a Minix 3.** Se realiza una breve descripción del sistema operativo Minix, resaltando puntos como su historia, estructura, características, objetivos, ventajas, desventajas, entre otros.
- ✓ **Capítulo 7: Proceso de arranque.** En este capítulo se presenta los conceptos asociados al proceso de arranque del computador. Además, se estudia a fondo los pasos del proceso de arranque, desde la perspectiva del diseño en el sistema operativo minix versión 3.1.6, también se muestra una documentación a fondo de programas involucrados para dicho proceso.
- ✓ **Capítulo 8: Implementación de un intérprete de comandos simple.** Para comprender la importancia de estas llamadas se propone un laboratorio que implemente el uso de las mismas, el cual se propone realizar un intérprete de comandos simple.
- ✓ **Capítulo 9: Implementación de llamadas al sistema.** La idea principal es comprender el esquema de funcionamiento y pasos a seguir para implementar las posibles llamadas al sistema en el sistema operativo Minix versión 3.1.6.
- ✓ **Capítulo 10: Implementación de semáforos.** Se muestra los mecanismos de concurrencia y

sincronización usados por Minix versión 3.1.6. Como caso de estudio se implementa semáforos en Minix, explicando el diseño y pasos a seguir para obtener la solución.

- ✓ **Capítulo 11: Modificación del planificador de procesos.** Se enseña el diseño y funcionamiento del planificador de la CPU (Central Processing Unit) en el sistema operativo Minix versión 3.1.6. También, se explica cómo modificar los elementos de diseño del planificador, lo cual permite evaluar a través de ciertos parámetros los ambientes generados, diagnosticando el rendimiento de cada uno de estos ambientes.
- ✓ **Capítulo 12: Conclusión.** En este capítulo se presenta las conclusiones encontradas durante el desarrollo de este Trabajo Especial de Grado, indicando si se alcanzaron o no los objetivos propuestos anteriormente descritos. Además, se indican de una serie de recomendaciones para posibles trabajos futuros que se puedan iniciar a partir de esta investigación y sus limitantes.
- ✓ **Capítulo 13: Referencias.** Contiene el conjunto de fuentes utilizadas y/o consultadas para la realización de este documento.

2 Marco teórico

En este capítulo se pretende introducir los principales conceptos asociados a la investigación, los cuales permitirán al lector ubicarse en el contexto deseado para un mayor entendimiento de los conocimientos y de la importancia de este trabajo especial de grado. Básicamente se habla de los sistemas operativos y de los sistemas operativos instruccionales; haciendo especial énfasis en el sistema operativo instruccional Minix 3 debido a que es la base fundamental de la investigación, destacando su historia, objetivos, estructura, ventajas, desventajas, entre otros.

2.1 *Sistema Operativo*

Un Sistema Operativo (SO) es un programa que siempre está en ejecución, el cual administra el hardware de una computadora con el objetivo de ser versátil, de fácil uso, eficiente y tener la capacidad para evolucionar (2); las preguntas que surgen luego de esa premisa es para quién y cómo hace ésta administración. Básicamente la administración la hace para los usuarios finales del hardware (3). Específicamente la hace para las aplicaciones que usan los usuarios, por eso es que también se dice que un SO es una capa de abstracción entre el hardware y el software (4). El usuario de dichas aplicaciones, es decir, el usuario final, normalmente no se preocupa por los detalles del hardware del computador. Por tanto, el usuario final ve un sistema de computación en términos de un conjunto de aplicaciones (5). Para tener una idea ilustrada puede ver gráficamente en la Figura 2.1.

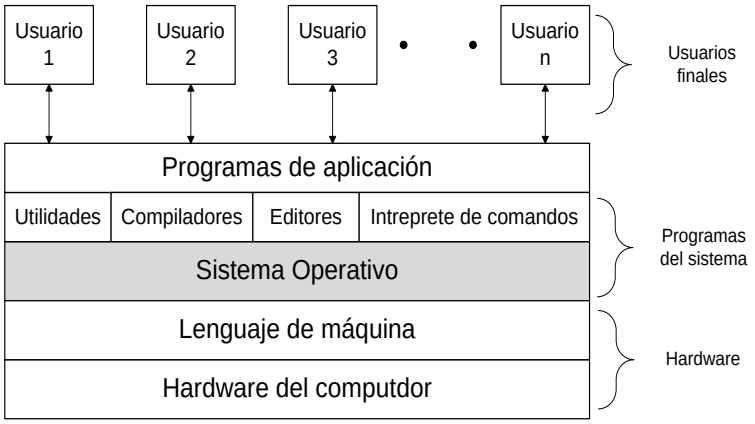


Figura 2.1 Capas de un sistema de computación

Una de las principales tareas de un SO es proporcionar un conjunto de primitivas para ser utilizadas por las aplicaciones. De forma resumida, estas primitivas proporcionan servicios en las siguientes áreas:

- Ejecución de programas: Se necesita realizar una serie de pasos para ejecutar un programa. Las instrucciones y los datos se deben cargar en memoria principal. Los dispositivos de E/S y los archivos se deben inicializar, y otros recursos deben prepararse. Los SO realizan estas labores de planificación para el usuario.

- Desarrollo de programas: proporciona una variedad de utilidades y servicios, tales como editores y depuradores, para ayudar al programador en el desarrollo de programas.
- Acceso a dispositivos de E/S: proporciona una interfaz uniforme que oculta los detalles de forma que los programadores y usuarios puedan acceder a dichos dispositivos utilizando lecturas y escrituras sencillas.
- Acceso a archivos: se debe tener una compresión detallada no sólo de la naturaleza del dispositivo de E/S, sino también de la estructura de los datos contenidos en los archivos del sistema de almacenamiento.
- Acceso al sistema: Acceso al sistema y recursos, brindando protección a los recursos y datos, evitando el uso no autorizado de los usuarios.
- Detección y respuesta a errores: Debe mantener un ambiente consistente al ocurrir cualquier error. Algunas de las acciones tomadas pueden oscilar entre finalizar el programa que causó el error hasta reintentar la operación, o simplemente informar del error.
- Contabilidad: Un buen sistema operativo recolecta estadísticas de uso de los diferentes recursos y monitorea los parámetros de rendimiento.

2.2 *Llamadas al sistema*

La interfaz entre el sistema operativo y los programas de usuario está definida por un conjunto de "operaciones extendidas" ofrecidas por el sistema operativo. Estas operaciones se definen como llamadas al sistema, en sí, son mecanismos por el cual un proceso solicita un servicio del núcleo (2) (3). Estas llamadas proveen funcionalidades adicionales a la aplicación, las cuales solo pueden ser ejecutadas en modo núcleo. Es decir, las funcionalidades de cierta manera permiten a las aplicaciones realizar un número mayor de operaciones, ya que éstas sólo se ejecutan en modo usuario.

Por encima del sistema operativo está el resto del software del sistema. Aquí se encuentra el intérprete de comandos (también conocido como Shell), los sistemas de ventanas, los compiladores, los editores y los demás programas independientes de la aplicación, como puede observarse en la Figura 2.1. Es importante darse cuenta de que ciertamente estos programas no son parte del sistema operativo. Éste es un punto crucial, pero sutil. El sistema operativo es (usualmente) la porción del software que se ejecuta en modo núcleo o modo supervisor, en el cual se pueden ejecutar instrucciones privilegiadas y se puede acceder a áreas de memoria protegida y a los dispositivos externos. Estas instrucciones conmutan la máquina del modo de usuario al modo núcleo y transfiere el control al sistema operativo. Es importante saber en su mayoría las CPUs tienen dos modos: modo núcleo para el SO, en el que permite todas las instrucciones y el modo usuario. El modo usuario es denominado así porque los programas de usuarios se ejecutan típicamente en este modo, es un modo que no tiene los privilegios del modo núcleo, con restricciones para acceder a ciertas áreas de memoria y ejecutar ciertas instrucciones.

2.3 *Procesos e Hilos*

Todas las computadoras modernas pueden realizar diferentes funcionalidades al mismo tiempo. Mientras ejecuta un programa de usuario, una computadora también puede estar leyendo de un disco y enviando texto a una pantalla o impresora. En un sistema de multiprogramación, la CPU también cambia de un programa a otro, ejecutando cada uno durante decenas de milisegundos. Si bien, estrictamente hablando, en un instante dado la CPU está ejecutando sólo un programa (suponiendo que solo tiene un procesador), en el curso de un segundo puede trabajar con varios programas, dando a los usuarios la ilusión de paralelismo. A veces se usa el término de seudoparalelismo para referirse a esta rápida conmutación de la CPU entre programas, para distinguirla del verdadero paralelismo de hardware de los sistemas multiprocesador (2) (3).

El concepto de proceso es fundamental en la estructura de los SO. Cada proceso tiene asociado un espacio de direcciones, una lista de posiciones de memoria desde algún mínimo hasta algún máximo, que el proceso puede leer y escribir. El espacio de direcciones contiene el programa ejecutable, los datos del programa, y su pila. A cada proceso también se asocia un conjunto de registros, que incluyen el contador del programa, el apuntador de la pila y otros registros de hardware. Así, como todo la demás información necesaria para ejecutar el programa. Este término de proceso tiene muchas definiciones en las cuales tenemos:

- Un programa en ejecución, que conceptualmente tiene su CPU virtual.
- Una instancia de un programa ejecutándose en un procesador.
- La entidad que se puede asignar o ejecutar en un procesador.
- Una unidad de actividad caracterizada por un solo hilo secuencial de ejecución, un estado actual, y un conjunto de recursos del sistema asociados.

Cada proceso tiene las siguientes dos características:

- Propiedad de recursos: Un proceso incluye un espacio de direcciones virtuales para el manejo de la imagen del proceso; la imagen de un proceso es la colección de programa, datos, pila y atributos definidos en el bloque de control del proceso. En ciertas ocasiones un proceso se le puede asignar control o propiedad de recursos tales como la memoria principal, dispositivos E/S y archivos. El sistema operativo realiza la función de protección para evitar interferencias no deseadas entre procesos en relación con los recursos.
- Planificación/ejecución: Un proceso tiene un estado de ejecución y una prioridad de activación, la cual es dependiente del algoritmo de planificación del sistema operativo.

2.4 *Exclusión mutua*

Hay que denotar que el sistema operativo busca coordinar las diversas actividades que la multiprogramación [1]provee, lo que resulta ser una tarea sumamente difícil. Por ende el sistema operativo debe lidiar y resolver las siguientes situaciones o problemas:

- Inapropiada sincronización: ocurre cuando existen fallas en el mecanismo de señalización, provocando que las señales se pierdan o se reciban duplicadas.
- Violación de la exclusión mutua: ocurren cuando programas intentan acceder simultáneamente a recursos compartidos y dichos accesos no son controlados.
- Interbloqueos: es posible que dos o más programas queden bloqueados esperándose entre sí.

La concurrencia es fundamental en todas estas áreas y en el diseño del sistema operativo. La concurrencia abarca varios aspectos, entre los cuales están la comunicación entre procesos, la compartición o competencia por recursos, y la sincronización de actividades de múltiples procesos. Hay que tomar en cuenta que todos estos asuntos no sólo suceden en el entorno del multiprocesamiento y el procesamiento distribuido, sino también en sistemas monoprocesador multiprogramados. Los principales conceptos de exclusión mutua son descritos en la Tabla 2.1 (2).

sección crítica	Sección de código dentro de un proceso que requiere acceso a recursos compartidos y que no puede ser ejecutada mientras otro proceso esté en dicha sección de código, es decir, sólo un proceso puede ejecutarla a la vez.
interbloqueo	Situación en la cual dos o más procesos son incapaces de actuar porque cada uno está esperando que alguno de los otros haga algo.
círculo vicioso	Situación en la cual dos o más procesos cambian continuamente su estado en respuesta a cambios en los otros procesos, sin realizar ningún trabajo útil.
exclusión mutua	Hace referencia a la condición que define el uso simultáneo de recursos comunes, como variables globales, por fragmentos de código conocidos como secciones críticas.
condición de carrera	Situación en la cual múltiples hilos o procesos leen y escriben un dato compartido y el resultado final depende de la coordinación relativa de sus ejecuciones.

[1] **Multiprogramación:** gestión de múltiples procesos dentro de un sistema monoprocesador.

inanición	Situación en la cual un proceso preparado para avanzar es ignorado indefinidamente por el planificador; aunque es capaz de avanzar, nunca se le escoge.

Tabla 2.1 Conceptos claves de concurrencia

2.4.1 Semáforos

En esta sección se describen los mecanismos usados por el sistema operativo y lenguajes de programación para proporcionar concurrencia. El primer avance fundamental en el tratamiento de los problemas de programación concurrente fue realizado por Dijkstra. Él estaba involucrado en el diseño de un sistema operativo representado como una colección de procesos secuenciales cooperantes, además, con el desarrollo de mecanismos eficientes y fiables para dar soporte a la cooperación.

Dijkstra plantea en su avance que dos o más procesos pueden cooperar por medio de simples señales, tales que un proceso pueda ser obligado a parar en un lugar específico hasta que haya recibido una señal específica. Cualquier requisito complejo de coordinación puede ser satisfecho con la estructura de señales apropiada. Para la señalización, se utilizan unas variables especiales llamadas semáforos. Para transmitir una señal vía el semáforo *S*, el proceso ejecutará la primitiva *semSignal*(*S*). Para recibir una señal vía el semáforo s, el proceso ejecutará la primitiva *semWait*(*S*); si la correspondiente señal no se ha transmitido todavía, el proceso se suspenderá hasta que la transmisión tenga lugar. Para conseguir el efecto deseado, el semáforo puede ser visto como una variable que contiene un valor entero sobre el cual sólo están definidas tres operaciones:

- Un semáforo puede ser inicializado a un valor no negativo.
- La operación *semWait*(*S*) decrementa el valor del semáforo. Si el valor pasa a ser negativo, entonces el proceso que está ejecutando *semWait*(*S*) se bloquea. En otro caso, el proceso continúa su ejecución.
- La operación *semSignal*(*S*) incrementa el valor del semáforo. Si el valor es menor o igual que cero, entonces se desbloquea uno de los procesos bloqueados en la operación *semWait*(*S*).

2.5 *Gestión de memoria*

Mientras se analizan varios mecanismos y políticas asociados con la gestión de la memoria, es útil mantener en mente los requisitos que la gestión de la memoria debe satisfacer. Se sugieren cinco requisitos:

- Reubicación.
- Protección.
- Compartición.

- Organización lógica.
- Organización física.

2.5.1 Reubicación

En un sistema multiprogramado, la memoria principal disponible se comparte generalmente entre varios procesos. Es una buena práctica poder intercambiar procesos en la memoria principal para maximizar la utilización del procesador, proporcionando un gran número de procesos para la ejecución. Una vez que un programa se ha llevado al disco, sería bastante limitante tener que colocarlo en la misma región de memoria principal donde se hallaba anteriormente, cuando éste se trae de nuevo a la memoria. Es por esto que nace la reubicación, que hace referencia al hecho de poder localizar a los programas para su ejecución en diferentes zonas de memoria.

2.5.2 Protección

Cada proceso debe protegerse contra interferencias no deseadas por parte de otros procesos, sean accidentales o intencionadas. Por tanto, los programas de otros procesos no deben ser capaces de referenciar sin permiso posiciones de memoria de un proceso, tanto en modo lectura como escritura. Por un lado, lograr los requisitos de la reubicación incrementa la dificultad de satisfacer los requisitos de protección. Por tanto, todas las referencias de memoria generadas por un proceso deben comprobarse en tiempo de ejecución para poder asegurar que se refieren sólo al espacio de memoria asignado a dicho proceso. Afortunadamente, los mecanismos que dan soporte a la reasignación también dan soporte al requisito de protección.

2.5.3 Compartición

Cualquier mecanismo de protección debe tener la flexibilidad de permitir a varios procesos acceder a la misma porción de memoria principal. Por ejemplo, si varios procesos están ejecutando el mismo programa, es ventajoso permitir que cada proceso pueda acceder a la misma copia del programa en lugar de tener su propia copia separada. Así como también, los procesos que estén cooperando en la misma tarea podrían necesitar compartir el acceso a la misma estructura de datos. Por tanto, el sistema de gestión de la memoria debe permitir el acceso controlado a áreas de memoria compartidas sin comprometer la protección esencial.

2.5.4 Organización lógica

Casi invariablemente, la memoria principal de un computador se organiza como un espacio de almacenamiento lineal o unidimensional, compuesto por una secuencia de bytes o palabras. A nivel físico, la memoria secundaria está organizada de forma similar. Mientras que esta organización es similar al hardware real de la máquina, no se corresponde a la forma en la cual los programas se construyen normalmente.

2.5.5 Organización física

La memoria del computador se organiza en al menos dos niveles, conocidos como memoria principal y memoria secundaria. La memoria principal proporciona acceso rápido a un coste relativamente alto. Adicionalmente, la memoria principal es volátil; es decir, no proporciona almacenamiento permanente. La memoria secundaria es más lenta, más barata que la memoria principal y normalmente no es volátil.

2.5.6 Particionamiento de la memoria

La tarea de mover la información entre los dos niveles de la memoria debería ser una responsabilidad del sistema. Esta tarea es la esencia de la gestión de la memoria. Las principales técnicas de gestión de memoria están descritas en la Tabla 2.2 (2).

Técnica	Descripción	Ventajas	Desventajas
Particionamiento fijo	La memoria principal se divide en particiones estáticas. Un proceso se puede cargar en una partición con igual o superior tamaño.	Sencilla de implementar, poca sobrecarga para el sistema operativo.	Uso ineficiente de la memoria, debido a la fragmentación interna.
Particionamiento dinámico	Las particiones se crean de forma dinámica, de tal forma que cada proceso se carga en una partición del mismo tamaño que el proceso.	No existe fragmentación interna, uso más eficiente de memoria principal.	Uso ineficiente del procesador, debido a la necesidad de compactación para evitar la fragmentación externa.
Paginación sencilla	La memoria principal se divide en marcos del mismo tamaño. Cada proceso se divide en páginas del mismo tamaño que los marcos. Un proceso se carga a través de la carga de todas sus páginas en marcos disponibles, no necesariamente contiguos.	No existe fragmentación externa.	Una pequeña cantidad de fragmentación interna.
Segmentación sencilla	Cada proceso se divide en segmentos. Un proceso se carga cargando todos sus segmentos en particiones dinámicas, no necesariamente contiguas.	No existe fragmentación interna; mejora la utilización de la memoria y reduce la sobrecargada respecto al particionamiento dinámico.	Fragmentación externa.

Paginación con memoria virtual	Exactamente igual que la paginación sencilla, excepto que no es necesario cargar todas las páginas de un proceso. Las páginas no residentes se traen bajo demanda de forma automática.	No existe fragmentación externa; mayor grado de multiprogramación; gran espacio de direcciones virtuales.	Sobrecarga por la gestión compleja de la memoria.
Segmentación con memoria virtual	Exactamente igual que la segmentación, excepto que no es necesario cargar todos los segmentos de un proceso. Los segmentos no residentes se traen bajo demanda de forma automática.	No existe fragmentación interna; mayor grado de multiprogramación; gran espacio de direcciones virtuales; soporte a protección y compartición.	Sobrecarga por la gestión compleja de la memoria.

Tabla 2.2 Técnicas de gestión de memoria

2.6 *Sistemas de archivos*

Desde el punto de vista del usuario, una de las partes más importantes de un sistema operativo es el sistema de archivos. El sistema de archivos proporciona las abstracciones de recursos típicamente asociadas con el almacenamiento secundario. El sistema de archivos permite a los usuarios administrar los archivos, los cuales tienen las siguientes propiedades:

- Existencia a largo plazo: Los archivos se almacenan en disco u otro almacenamiento secundario y no desaparece cuando un usuario se desconecta.
- Compartible entre procesos: Los archivos tienen nombres y pueden tener permisos de acceso asociados que permitan controlar la compartición.
- Estructura: Dependiendo del sistema de archivos, un archivo puede tener una estructura interna que es conveniente para aplicaciones particulares. Adicionalmente, los archivos se pueden organizar en estructuras jerárquicas o más complejas para reflejar las relaciones entre los mismos.

El sistema de archivos además de almacenar los datos organizados como archivos, también provee una colección de funciones que se pueden llevar a cabo sobre los archivos. Algunas operaciones son las siguientes:

- Crear: Se define un nuevo archivo y se posiciona dentro de la estructura de archivos.
- Borrar: Se elimina un archivo de la estructura de archivos y se destruye.
- Abrir: Un archivo existente se declara «abierto» por un proceso, permitiendo al proceso realizar funciones sobre dicho archivo.

- Cerrar: Un determinado proceso cierra un archivo, de forma que no puede volver a realizar determinadas funciones sobre el mismo, a no ser que vuelva a abrirlo.
- Leer: Un proceso lee de un archivo todos los datos o una porción de ellos.
- Escribir: Un proceso actualiza un archivo, bien añadiendo nuevos datos que expanden el tamaño del archivo, bien cambiando los valores de elementos de datos existentes en el archivo.

Típicamente, un sistema de archivos mantiene un conjunto de atributos asociados al archivo. Estos incluyen el propietario, tiempo de creación, tiempo de última modificación, privilegios de acceso, etc.

2.7 *Herramientas de enseñanza en Sistemas Operativos.*

En algunas casas de estudios las clases de pregrado de sistemas operativos se han impartido tradicionalmente usando sistemas operativos instruccionales. Estos sistemas operativos están destinados a ser simples y de fácil entendimiento, a su vez ellos carecen intencionalmente de ciertas piezas que los estudiantes deben implementar y les sirven como ejercicios (6).

Los sistemas operativos constituyen uno de los tópicos más importantes y complejos de enseñar en cualquier pensum de estudio en las Ciencias de la Computación. Los conceptos y definiciones asociados a su contenido representan un reto para la metodología educativa a utilizar, cuyo objetivo primordial es ofrecer al estudiante los conocimientos para comprender y manejar los principios básicos en los que se fundamenta el diseño e implementación de los sistemas operativos modernos (7).

Con la finalidad de mejorar el desempeño estudiantil y la pedagogía de un curso de pregrado de sistemas operativos surge la necesidad de utilizar una herramienta educativa. Ésta herramienta debe permitir a cada uno de los distintos actores participar en un experiencia completa en el diseño, desarrollo y evaluación referente a los tópicos primordiales dictados en los espacios teóricos-prácticos del curso.

A la hora de enseñar al estudiante los tópicos de los sistemas operativos, el instructor debe decidir qué tipo de tareas realizarán los estudiantes. Estas asignaciones pueden ser puramente teóricas como preguntas y respuestas sobre los temas. Alternativamente, un instructor puede optar por proveer tareas prácticas. Hay una serie de posibilidades para las asignaciones de programación. Una de ellas involucra los conceptos relacionados con los sistemas operativos sin tener que incluir la programación de un sistema operativo. Por ejemplo, los estudiantes podrían plantear una solución al problema de la cena de los filósofos. Sin embargo, lo ideal sería que los estudiantes puedan tener asignaciones para modificar o desarrollar un sistema operativo; la pregunta que surge es ¿Sobre cuál sistema operativo deberían trabajar los estudiantes?

La solución obvia es crear un sistema operativo “pequeño” que posea las características básicas asociadas a las estructuras computacionales modernas, conceptos, diseños, además que sirva como

plataforma educativa, más que como un fin totalmente funcional. A estos los llamaremos sistemas operativos instruccionales (1).

Durante las últimas dos décadas el uso de este material didáctico se ha convertido en el medio de concepción, generación y aceptación de conocimiento sobre los diferentes paradigmas que sustentan a los sistemas operativos modernos. Lo cual presenta una problemática mayor, como manejar el crecimiento exponencial de estas piezas de software y como mantener las herramientas educativas actualizadas para que puedan explicar los nuevos enfoques científicos y tecnológicos de la computación, por supuesto sin alterar ni romper los límites académicos del curso[2].

2.8 *¿Qué es un Sistema Operativo Instruccional (SOI)?*

Los cursos de pregrado de sistemas operativos generalmente se enseñan utilizando uno de dos enfoques: abstracto o concreto. En el enfoque abstracto, los estudiantes aprenden los conceptos subyacentes a la teoría de los sistemas operativos, y los aplican utilizando hilos a nivel de usuario en un sistema operativo anfitrión. En el enfoque concreto, los estudiantes aplican los conceptos para trabajar sobre un verdadero núcleo de sistema operativo. En la más pura manifestación del enfoque concreto, los estudiantes ponen en práctica los proyectos de sistemas operativos que se ejecutan en hardware real (8).

Un sistema operativo instruccional es un software que tiene como objetivo enseñar los conceptos más importantes de sistemas operativos mediante el diseño e implementación de las funciones que permiten desempeñar el trabajo del mismo a través de un enfoque concreto (7) (6). Esto se logra mediante un conjunto de asignaciones las cuales los estudiantes deben implementar para desarrollar o mejorar el sistema operativo instruccional (1). Finalmente, se desea que estos sistemas operativos sean lo suficientemente realistas como para mostrar cómo funcionan los sistemas operativos reales e igualmente sean bastante simples para que los estudiantes puedan comprenderlo y modificarlo de manera significativa sin mayores dificultades[1].

Estos sistemas operativos instruccionales están destinados a ser utilizado en cursos de enseñanza de sistemas operativos. Los objetivos y mecanismos planteados por estos sistemas operativos se basan en un conjunto de asignaciones que permiten desarrollar o mejorar el sistema operativo instruccional. Permitiendo obtener la comprensión de la carga cognitiva de los aspectos teóricos en un ambiente o entorno de programación sencillo y amigable que garantice el desarrollo sustentable de conocimiento. Por ejemplo, la asignación de técnicas de programación concurrente permitiría afianzar los puntos primordiales asociados a la concurrencia de procesos, o en otros casos con un enfoque más real

[2] "The NachOS Instructional Operating System" - http://techreports.lib.berkeley.edu/

podríamos modificar algún código asociado al manejo de procesos en el núcleo de nuestro sistema operativo para verificar y comprender su comportamiento.

Durante las últimas dos décadas el desarrollo de los sistemas operativos ha aumentado generando un extenso número de variantes, ya sean tanto propietarias como de código abierto. Lo que permite escoger al más adecuado dentro de un catálogo de posibilidades con la finalidad de estudiarlo, analizarlo y modificarlo. Los proyectos reales de los sistemas operativos modernos, su complejidad y desarrollo son tan avanzados que pueden suprimir el objeto mismo de estudio y convertir la experiencia de aprendizaje en una pesadilla engorrosa de miles de líneas de código fuente.

Antes de seleccionar un sistema operativo instruccional para el estudio o incluso para la creación de uno se está en la obligación de revisar un poco la historia y los proyectos correspondientes a este tipo de software educativo. También se debe considerar ciertos aspectos que permitirán tener una visión clara y objetiva. Una de las problemáticas en torno a estas herramientas parten sobre las plataformas soportadas y a su vez sobre como lo hacen. Desconocer la interacción existente entre el sistema operativo y el hardware con el que se comunica puede originar un fuerte impacto sobre los estudiantes durante la fase de obtención y generación de conocimiento; esto debido a que gran parte de la comunicación hacia los dispositivos se lleva a cabo a través de lenguaje ensamblador. Por ser un entorno de programación muy distinto a los usualmente utilizados puede causar en el estudiante desmotivación al momento de generar un proyecto de sistema operativo.

Entre los componentes de hardware el CPU se le otorga mayor importancia por ser el principal dispositivo en el computador moderno, dentro de la alta gama de procesadores en el mercado el más común entre ellos es la familia de procesadores Intel x86 o x86-64, esta realidad hace necesario que los estudiantes conozcan y dominen este tipo de arquitectura, pero presenta una desventaja debido a que dicho procesador posee un amplio conjunto de instrucciones que le permiten mayor robustez, pero inevitablemente mayor complejidad en su uso. Este hecho hace pensar en arquitecturas con menor complejidad, como la ofrecida por los procesadores MIPS (Microprocessor without Interlocked Pipeline Stages), cuya principal adversidad se presenta en la escasez de recursos y de herramientas suficientes para su gestión (1).

Dependiendo del planteamiento seleccionado también se debe tomar en cuenta que tipo de interacción se le ofrecerá al estudiante con respecto al hardware. Trabajar directamente con el hardware puede acarrear ciertas desventajas tanto en el ámbito educativo como en la parte asociada a la programación, además, estos conceptos se encuentran fuera de los tópicos básicos de sistemas operativos. Una solución que permite evitar esta problemática es manejar emuladores, los cuales permiten establecer una nueva capa de interacción entre el sistema operativo y el hardware o arquitectura del computador.

2.9 *Herramientas de virtualización, simulación y emulación*

El hardware x86 actual está diseñado originalmente para ejecutar un único sistema operativo y al menos una aplicación, pero la virtualización ha acabado con estas limitaciones haciendo posible la ejecución concurrente de varios sistemas operativos, y varias aplicaciones en el mismo computador, aumentando con ello la utilización y la flexibilidad del hardware.

Básicamente, la virtualización permite transformar hardware en software. Utilizar software para transformar o virtualizar los recursos de hardware de un computador x86, incluidos CPU, RAM, disco duro, y controlador de red; para crear una máquina virtual completamente funcional que puede ejecutar su propio sistema operativo y aplicaciones de la misma forma que lo hace un computador "real".

Un emulador es en sí un programa que crea una capa extra entre una plataforma existente (plataforma anfitrión) y la plataforma a ser reproducida (plataforma de objetivo)[3]. A menudo es confuso distinguir entre un emulador y un simulador. Normalmente, un emulador se ejecuta en el hardware (aunque también se puede ejecutar sobre software), mientras que un simulador se implementa en el software. Por ejemplo, un emulador de router se utiliza para probar el rendimiento o errores en el hardware y software del router. Los errores pueden incluir los tiempos de reloj, los problemas en la secuenciación de instrucciones, y la prueba de velocidad. Un simulador se implementa solamente en el software. Como resultado, no tendrá la capacidad de emular el entorno de hardware, tales como los tiempos de reloj, simuladores de las pruebas de velocidad, etc. son relativamente lentos, ya que se ejecutan en el software[4].

	Ejemplos
virtualizador	OpenVZ, VMWare, VirtualBox, Virtuozzo, etc.
simulador	Simics, SPIM, etc.
emulador	Bochs, DOSBox, E/OS (Emulator Operating System), Qemu, etc.

Tabla 2.3 Ejemplos de virtualizadores, simuladores y emuladores

2.10 *Sistemas Operativos Instruccionales*

Varias universidades han desarrollado sus propias herramientas educativas, conocidas como sistemas operativos instruccionales. Cada una de estas se han adecuado al enfoque u objetivos planteados por las mismas. Es por ello que a lo largo del tiempo han ido creando, evolucionando o mejorando más herramientas de este estilo. Existen distintos proyectos los más comunes se describen a continuación:

---

[3] "What is emulation?" - http://www.kb.nl/

[4] "Router Simulator Vs. Emulator" - http://routersimulator.certexams.com/

2.10.1 OS/161

Fue desarrollado en la Universidad de Harvard por David Holland para ser utilizado como herramienta educativa en los cursos de sistemas operativos de esta casa de estudios superiores. Los objetivos en su desarrollo fueron proporcionar un entorno de ejecución realista; facilitar la depuración y mantener la simplicidad. OS/161 hace el intento de simular un sistema operativo real, y al mismo tiempo ser lo suficientemente simple para ser manejado por estudiantes de pregrado. Es intencionadamente similar a BSD Unix en la organización y estructura. Viene con una docena o más de los comandos básicos de Unix y permite utilizar una interfaz para las llamadas al sistema parecida a Unix (9).

2.10.2 NachOS

Es un programa instruccional desarrollado por Christopher, Procter y Anderson en la Universidad de California. NachOS es usado por Berkeley y numeras universidades (10). Su objetivo es proporcionar un entorno para que los estudiantes de pregrado desarrollen un sistema operativo. Se provee a los estudiantes un diseño básico, en este caso, de las piezas de trabajo suficientes para cargar y ejecutar un simple programa de usuario (dicho programa es NachOS). A través de una serie de tareas, el estudiante implementa la funcionalidad de multiprogramación, memoria virtual, y un sistema de archivos. Aunque se trabajan sobre una máquina simulada, la máquina se basa en un procesador real, así que las cuestiones que el estudiante debe resolver son realistas y representativas del desarrollo sistema operativo real (11).

NachOS simula un procesador real MIPS R2/3000. La primera versión de NachOS se completó en enero de 1992 y se utilizó como un proyecto de pregrado sistemas operativos en Berkeley. La versión 3.4 se implementó sólo en C++. La versión 4.0 introduce instrucciones en C y fue finalizada en el año 1996. Posteriormente se implementó la versión 5.0j en Java, desarrollada en Berkeley por Hettena Dan y Rick Cox. NachOS 5.0j es una reescritura casi total, con una estructura similar a la 4.0. Dicha versión 5.0j fue desarrollado en 2001[5].

2.10.3 Minix

Fue desarrollado por Andrew Tanenbaum, es un sistema operativo instruccional famoso y conocido debido a que fue objeto de inspiración de Linus Torvalds para iniciar el sistema operativo Linux. Es un clon del sistema operativo Unix. Es distribuido junto con su código fuente y desarrollado por el profesor Andrew S. Tanenbaum en 1987. Gracias a su reducido tamaño, diseño basado en el paradigma del micronúcleo, y su amplia documentación, resulta bastante apropiado para personas que desean instalar un sistema operativo compatible con Unix en su máquina personal así como aprender sobre su funcionamiento interno.

---

[5] "A Guide to NachOS 5.0j" - http://www-inst.eecs.berkeley.edu/

2.10.4 GeekOS

Desarrollado en la Universidad de Maryland. El objetivo de GeekOS es ser una herramienta para aprender acerca del funcionamiento del núcleo. Desde la versión 0.2.0, viene con un conjunto de proyectos adecuados para su uso en un curso de pregrado sistemas operativos, o para el aprendizaje autodirigido. GeekOS se ha utilizado en los cursos de varias universidades.

Sus objetivos principales son realismo, simplicidad y fácil entendimiento. Posee las siguientes características técnicas, manejo de interrupciones, manejador de memoria, manejo de hilos de núcleo por slots de tiempo predefinidos con un esquema de planificación estático de prioridades, manejo de variables de condición para garantizar el procedimiento de sincronización de hilos y soporte a dispositivos entrada/salida. Esta desarrollado bajo el lenguaje de programación C (8).

2.10.5 JOS

Es un esqueleto de un sistema operativo el cual tiene funciones al estilo Unix (ejemplo: fork, exec), con la diferencia de que está diseñando e implementado como exonúcleo (es decir, las funciones de Unix están implementadas en su mayoría como bibliotecas a nivel de usurario en lugar estar integradas en el núcleo). Es usado en MIT como código fuente para que los estudiantes desarrollen a partir del mismo su sistema operativo[6] (12).

2.11 *Comparación entre los Sistemas Operativos Instruccionales*

Una vez presentado los conceptos, la finalidad, filosofía de los SOI y los distintos software de virtualización, simulación y emulación; se tienen argumentos suficientes para establecer criterios comparativos de estos sistemas. A partir de estos criterios se presentará un cuadro comparativo donde se muestra los puntos claves de estos SOI de una manera concisa y precisa, permitiendo calificarlos de manera cuantitativa, vea la Tabla 2.4. Para la Tabla 2.4 los atributos tomados a consideración son los siguientes: SOI (nombre del SOI), desarrollador, característica principal, limitaciones, licencia y modo de desarrollo (MDD), última versión, programado en (lenguaje(s) utilizado(s) para su implementación), plataforma requerida (ambiente o arquitectura requerida para poder ejecutar el SOI) y plataforma destino (arquitectura para la cual el SOI está diseñado, es la arquitectura y conjunto de instrucciones que conoce y utiliza el SOI).

[6] "Operating System Engineering" - http://pdos.csail.mit.edu/

SOI	Desarrollador	Característica Principal	Limitaciones	Última versión	Lenguaje	Plataforma Requerida	Plataforma Destino
OS/161	Universidad de Harvard. (Cambridge - Massachusetts)	BSD-like, OS/161 intenta dar un sentido realista como sistema operativo, al mismo tiempo ser lo suficientemente simple para repartir a los estudiantes	No se puede ejecutar directamente sobre el hardware.	1.14. septiembre de 2005	Todo escrito en C	Un simulador de arquitectura MIPS, puede ser cualquiera, pero sus desarrolladores crearon System/161 para este fin, el cual necesita un SO Unix-like	MIPS R2/2000
NachOS	Universidad de Berkeley (Berkeley - California).	Núcleo monolítico. En las asignaciones se estudian e implementan todas las áreas de los sistemas operativos modernos	Se ejecuta como un proceso de usuario en el sistema operativo. Lo que le quita realismo.	4.0 1996	Principalmente implementado en C++	SunOS, Solaris, Linux, NetBSD y FreeBSD	MIPS R2/3000, Sun SPARC, DEC Alpha, RS/6000
NachOS (versión en java)	Universidad de Berkeley (Berkeley - California).	Núcleo monolítico. En las asignaciones se estudian e implementan todas las áreas de los sistemas operativos modernos. Java es más simple que C++. Java es relativamente portable	Se ejecuta como un proceso de usuario en el sistema operativo. Lo que le quita realismo.	5.0j 2001	Implementado en Java	Cualquier plataforma que soporte la máquina virtual de Java. Se puede utilizar las aplicaciones Eclipse o Netbeans para desarrollar los proyectos.	MIPS R2/3000
Minix	Andrew Tanenbaum (Amsterdam - Netherlands)	Unix-like, Está basado en una estructura de micronúcleo. Es muy realista y completo. Es extremadamente pequeño, flexible, seguro y estable. También cuenta con debugger. Tiene dos enfoques tanto de enseñanza como comercial	Debido a su pequeño tamaño el código es denso, sin embargo, su documentación es extensa	3.1.6 febrero de 2010	Principalmente implementado en C	Directamente sobre el hardware o usando un emulador o máquina virtual como virtual VMWare en Windows, VirtualBox o Qemu	Intel 386 o superior, pero no soporta x86 64-bit.

GeekOS	Universidad de Maryland (Maryland)	Sus características principales son realismo, simplicidad y fácil entendimiento. Su objetivo principal es servir como un ejemplo sencillo, pero realista, de un núcleo de sistema operativo.	No soporta paginación con memoria virtual.	0.3.0 Abril de 2005	C	Directamente sobre el hardware o puede ejecutarse sobre un simulador, recomiendan Bochs. Probado sobre las plataformas Linux/i386, FreeBSD, Windows con cygwin y Unix.	IA-32
JOS	Massachusetts Institute of Technology (Cambridge - Massachusetts)	Unix-like, exonúcleo, es un esqueleto de un sistema operativo provisto por el MIT para los estudiantes de sus cursos para que los mismos desarrollen uno a partir de JOS. Muy realista y provee mecanismos para su depuración.	Poca información acerca de la documentación del SO	Última publicación en diciembre de 2007	C	Directamente sobre el hardware o puede ejecutarse sobre Bochs.	IA-32

Tabla 2.4 Tabla comparativa de los Sistemas Operativos Instruccionales

A la hora de evaluar un sistema operativo instruccional se tomaron en cuenta varios aspectos, como por ejemplo realismo, simplicidad, documentación, ambiente de depuración, vigencia, en cuantas universidades se usa, entre otros. Fueron en gran parte estos criterios los tabuladores a la hora de evaluar una gama de sistemas operativos instruccionales, este estudio se realizó en una investigación previa (13), en la cual el sistema operativo instruccional Minix fue el que mejor se ajustaba al curso de Sistemas Operativos de la Universidad Central de Venezuela.

¿Por qué utilizar Minix en la UCV? Para responder esta pregunta se tomaran en cuenta las características de este sistema operativo instruccional, las cuales serán presentadas a continuación:

- Minix 3 es una herramienta educativa que cuenta con distintos niveles de documentación, entre los cuales resaltan:
    - Posee una lista de correo activa donde se le permite a los usuarios emitir preguntas y soluciones que se presentan sobre este sistema operativo, la cual es soportada por la comunidad de desarrollo.
    - Posee un sitio Web donde es alojado el proyecto, y además se encuentran disponibles todas las versiones de Minix desde su lanzamiento.
    - Posee un portal dedicado a manuales de referencia para usuarios y para desarrolladores, el primero incluye la información necesaria para la instalación, gestión e interacción con el sistema operativo. El segundo incluye el API (Application Programming Interface) e información para desarrollar aplicaciones en dicho sistema operativo, así como también, información sobre su código fuente.
    - Posee una publicación bibliográfica titulada “Operating System, Design and Implementation” donde los principales autores de este SO, Andrew Tanenbaum y Albert Woodhull, explican los principios básicos de los sistemas operativos modernos. Asimismo, se tiene como caso de estudio a Minix 3, en donde se muestra y explica de forma detallada su código fuente, estructuras de datos y funcionamiento del sistema operativo.
- Debido a su grupo de desarrolladores activos es una herramienta que posee actualizaciones continuas, un ejemplo de esto es que su última actualización fue publicada para 4 febrero de 2010. Lo cual refleja la continuidad del proyecto.
- Al ser un sistema operativo que tiene como plataforma destino IA-32 es lo suficientemente realista como para ser instalado sobre el hardware al desnudo. Teniendo en cuenta este escenario se pretende proveer al estudiantado una experiencia básica pero real, relacionada a los principios concernientes a los sistemas operativos, a través de una herramienta que se puede instalar sobre la una arquitectura popular (IA-32), sin necesidad de realizar programación de controladores.

- Al ser un sistema operativo totalmente funcional, provee la ventaja de entender y modificar de forma más sencilla su código fuente a diferencia de tener que desarrollar todo desde cero.
- Permite trabajar sobre un entorno básico de desarrollo Unix-like, el cual puede instalarse sobre herramientas de virtualización como VMware, Virtual Box, Qemu, etc.

3 Adecuación de Minix 3 a la UCV

En este capítulo serán descritos todos los laboratorios propuestos para la adecuación de Minix 3 al curso de Sistemas Operativos de la Escuela de Computación de la Universidad Central de Venezuela. Esta sección presenta el enunciado de los laboratorios propuestos. La implementación de la solución y la documentación de la misma serán explicadas con destalle en los capítulos posteriores.

3.1 *Laboratorio 0 – Instalación de Minix y entorno de desarrollo*

El primer laboratorio está diseñado para sentar las bases de las herramientas necesarias para desarrollar a lo largo del curso todos los laboratorios propuestos por el grupo docente. Este es un punto clave ya que permite engranar todas las aplicaciones y el SOI Minix 3 para facilitar en gran medida el desarrollo de la implementación de los laboratorios.

3.1.1 Motivación

Este laboratorio pretende introducir el sistema operativo instruccional Minix versión 3.1.6, el cual será la herramienta educativa a utilizar a lo largo de los laboratorios docentes de la materia Sistemas Operativos. Igualmente se desea familiarizar a los estudiantes con el entorno de desarrollo de Minix versión 3.1.6. Es importante señalar que Minix es ser un sistema operativo Unix-like.

3.1.2 Objetivos

Los objetivos a alcanzar es dar a conocer las herramientas a utilizar en el manejo de los laboratorios, para esto el estudiante debe aprender a instalarlas, configurarlas y desenvolverse en las mismas.

3.1.3 Grupo docente

Para cumplir los objetivos planteados en este laboratorio el grupo docente debe encargarse de las siguientes asignaciones:

- Proveer a los estudiantes los archivos ejecutables necesarios para la instalación del entorno de desarrollo propuesto. Esto se logra por medio de una aplicación para control de proyectos (wiki) desde la cual los estudiantes podrán descargarse los archivos mencionados y la imagen del sistema operativo Minix versión 3.1.6.
- Explicar la instalación de cada una de las herramientas. Para esto se debe proveer a los estudiantes de los parámetros de configuración y un conjunto de pasos que permitan conseguir el entorno de programación deseado. Esto se lograra mediante el apoyo de una serie de video tutoriales realizados por esta investigación.
- Enseñar un conjunto de órdenes básicas para el manejo del intérprete de comandos del SOI Minix versión 3.1.6.
- El grupo docente debe explicar los requerimientos planteados para este laboratorio.

3.1.4 Estudiantes

El estudiante deberá asistir a su clase de laboratorio correspondiente y realizar las actividades de instalación siguiendo las instrucciones del grupo docente y realizar las siguientes actividades

- Instalar la herramienta de virtualización VMWare Workstation versión 6.5.1.
- Instalar el Sistema Operativo Instruccional Minix versión 3.1.6.
- Instalar el IDE Eclipse (classic) versión 3.5.0 para el desarrollo del SOI Minix versión 3.1.6.
- Integrar el IDE Eclipse y el SOI Minix versión 3.1.6.

3.1.5 Entregables

Para evaluar este primer laboratorio se requiere a los estudiantes que entreguen su primera máquina virtual la cual contiene el sistema operativo Minix versión 3.1.6 con un programa simple que imprima por salida estándar “Hola Mundo”.

3.1.6 Duración

Este laboratorio se pretende impartir en 2 horas de clase equivalentes a una clase de laboratorios.

3.1.7 Documentación y ayuda

Como se menciono con anterioridad tanto los estudiantes como el grupo docente podrán apoyarse con una aplicación para control de proyecto y varios videos tutoriales.

3.2 *Laboratorio 1 – Introducción a Minix 3*

En este laboratorio se da a conocer el SOI Minix 3, así como también, una introducción de su historia, de su estructura, de las principales características, etc.

3.2.1 Motivación

La motivación recae en explicar de manera teórica los tópicos referentes a Minix, para que el estudiante conozca el sistema operativo y pueda desenvolverse con fluidez en el desarrollo de los laboratorios.

3.2.2 Objetivos

El objetivo a alcanzar en este laboratorio es mostrar a los estudiantes los conceptos asociados al SOI Minix 3, para esto se hizo especial énfasis en su historia, su estructura, principales características, etc.

3.2.3 Grupo docente

El grupo docente debe:

- Introducir los principales conceptos teóricos asociados al SOI, su objetivo, las ventajas, desventajas, etc.

- Explicar el SOI Minix describiendo su estructura, de las principales características y en donde puede obtener información del mismo.
- El grupo docente debe explicar los requerimientos planteados por este laboratorio.

3.2.4 Estudiantes

El estudiante debe asistir a la clase de laboratorio y prestar la atención necesaria para poder entregar los requerimientos solicitados por el grupo docente.

3.2.5 Entregables

El estudiante debe entregar un informe con lo siguiente:

- Descripción del Sistema Operativo Minix 3.
- La Historia de Minix.
- Estructura de Minix 3.
- Ventajas de la arquitectura.
- Desventajas de la arquitectura.
- Descripción del proceso de instalación de Minix 3.
- Requerimientos necesarios para la instalación de Minix 3.
- Describa los comandos utilizados en clase, su sintaxis y funcionamiento.

3.2.6 Duración

Este laboratorio se pretende impartir en 2 horas de clase equivalentes a una clase de laboratorios.

3.2.7 Documentación y ayuda

El grupo docente y los estudiantes podrán apoyarse con una aplicación para control de proyecto la cual contendrá parte de esta investigación y el seminario referente a la misma.

3.3 *Laboratorio 2 – Estudio del proceso de arranque*

El proceso de arranque es el inicio de la ejecución de cualquier sistema operativo, siendo este un tema básico para iniciar el entendimiento del funcionamiento del mismo; es por esto que entre los primero conocimientos que un estudiante de un curso de sistemas operativos debe adquirir es este proceso; ya que garantiza un aprendizaje secuencial.

3.3.1 Motivación

Como se menciono el proceso de arranque es uno del los tópicos principales para la compresión del funcionamiento de un sistema operativo moderno, es por ello que se decidió realizar un laboratorio dedicado a este tema.

3.3.2 Objetivos

La tarea de este laboratorio es conocer a fondo cómo funciona el proceso de arranque de un SO real. Además, al realizar este laboratorio el estudiante aprenderá los pasos necesarios para la ejecución de un SO, en este caso Minix 3, tanto la parte de hardware como de software.

3.3.3 Grupo docente

En este laboratorio se debe explicar específicamente cuáles son los pasos que sigue Minix 3 para poder arrancar el SO. Explicando los pasos de hardware, luego los pasos de software. Se debe hacer especial énfasis en las particiones, como también en los archivos *masterboot.s* y *bootblock.s.*

3.3.4 Estudiantes

El estudiante debe de asistir a clases y realizar las actividades propuestas por el grupo docente de manera de alcanzar los objetivos planteados en este laboratorio. El grupo docente debe explicar los requerimientos planteados por este laboratorio.

3.3.5 Entregables

El estudiante debe entregar un informe con las respuestas de las preguntas señaladas. Para esto se le solicita al estudiante la descripción de:

- ¿Cuál es el conjunto de paso que realiza un computador para poder ejecutar el SO instalado? Describa cada paso con detalle.
- ¿Qué papel desempeña la BIOS (Basic Input Output System) en este proceso?
- ¿Cuáles son los principales dispositivos de almacenamiento que están involucrados en este proceso y cuál es su estructura?
- ¿Cuáles son los modos de acceso a los discos e indique la razón de cada uno?
- Identifique cuales son los programas involucrados para llegar a la ejecución de Minix 3. Posteriormente, describa a groso modo el funcionamiento de cada uno y para finalizar describa cada instrucción que ejecutan dichos programas.

3.3.6 Duración

Este laboratorio se pretende impartir en 2 horas de clase equivalentes a una clase de laboratorios. Y se plantea como tarea entregar el informe asociado al mismo dando un lapso de una semana.

3.3.7 Documentación y ayuda

El grupo docente y los estudiantes podrán apoyarse con una aplicación para control de proyecto la cual contendrá parte de esta investigación y el seminario referente a la misma.

3.4 *Laboratorio 3 – Implementación de un intérprete de comandos*

Un intérprete de comandos es una pieza de software que provee una interfaz para los usuarios de un sistema operativo que proporciona acceso a los servicios del núcleo. Se plantea en este laboratorio una implementación sencilla de un intérprete de comandos.

3.4.1 Motivación

Las llamadas al sistema juegan un rol importante en los programas de usuario, estas llamadas proveen funcionalidades adicionales a las aplicaciones, las cuales solo pueden ser ejecutadas en modo núcleo. Para comprender la importancia de estas llamadas se propone un laboratorio que implemente el uso de las mismas, el cual propone realizar un intérprete de comandos simple.

3.4.2 Objetivos

El objetivo principal es proveer la oportunidad de aprender cómo utilizar las llamadas al sistema. Para hacer esto, se debe implementar un intérprete de comandos de Unix. Un intérprete de comandos es simplemente un programa que permite ejecutar otros programas. El programa resultante se parecerá a los intérpretes de comando de Unix/Linux.

3.4.3 Grupo docente

El grupo docente debe explicar cada una de las llamadas al sistema a utilizar para la implementación de un intérprete de comandos. Además, para ayudar al desarrollo del intérprete de comandos se provee un código base a los estudiantes, esto ayuda a la implementación de la solución, ya que provee ciertas funcionalidades básicas que el intérprete de comandos debe cumplir. En si el grupo docente debe realizar las siguientes actividades:

- El grupo docente debe proporcionar el enunciado de los requerimientos de este laboratorio. Además, debe aclarar cualquier duda que del mismo pueda surgir.
- Para la inicialización de este laboratorio también debe proveer a los estudiantes de la plantilla asociada, explicando cómo manipularla y dando una breve descripción de cada uno de los archivos que la componen.
- Como ayuda se propone explicar las principales llamadas al sistema que deben utilizarse para la implementación del intérprete de comandos, para esto se explica la sintaxis y semántica de las mismas, culminando con un código de ejemplo.
- El grupo docente debe explicar los requerimientos planteados por este laboratorio.

3.4.4 Estudiantes

El estudiante debe de asistir a clases y realizar las actividades propuestas por el grupo docente de manera de alcanzar los objetivos planteados en este laboratorio.

3.4.5 Entregables

El estudiante debe implementar su propio intérprete de comandos el cual debe soportar las siguientes instrucciones:

- Para empezar el intérprete de comandos a implementar debe soportar el comando interno "*exit*", el cual terminará la ejecución del intérprete de comandos.
    - Conocimientos: funcionamiento de los comandos del intérprete.
    - Llamadas al sistema: *exit()*
    - Funciones de la biblioteca estándar: *strcmp()*
- Un comando sin argumentos. Ejemplo: ls. El intérprete de comandos debe bloquearse hasta que el comando complete su ejecución, si el código de retorno es anormal, debe imprimir un mensaje indicando el error. Los comandos se almacenan en */bin* y */usr/bin* (aunque si se utiliza *execvp*() en lugar de execv(), la ubicación de los comandos no importa).
    - Conocimientos: bifurcación, espera hasta que finalice el proceso hijo, la ejecución sincrónica.
    - Llamadas al sistema: *fork(), execvp(), exit(), wait()*
- Un comando con argumentos. Ejemplo: *ls -l*
    - Conocimientos: parámetros de línea de comandos. El argumento cero es el nombre del comando, los siguientes argumentos siguen la secuencia.
- Un comando, con o sin argumentos, cuya salida se redirige a un archivo. Ejemplo: *ls -l > archivo.txt*
    - Conocimientos: Las operaciones de archivo, la redirección de salida. Este toma la salida del comando y lo pone en el archivo indicado.
    - Llamadas al sistema: *close(), dup(), open().*
- Un comando, cuya entrada es redirigida a un archivo. Ejemplo: *sort < archivo.txt*
    - Conocimientos: redirección de entrada, las operaciones con archivos. Se utiliza el archivo denominado como entrada al comando.
    - Llamadas al sistema: *close(), dup(), open()*
- Un comando, con o sin argumentos, cuya resultado se redirecciona a la entrada de otro comando. Ejemplo: *ls -l | sort*
    - Conceptos: Tuberías. La salida del primer comando es la entrada al segundo.
    - Llamadas al sistema: *pipe(), close(), dup()*

El intérprete de comandos debe chequear y manejar correctamente todos los valores de retorno. Esto significa que necesita leer las páginas del manual, debe averiguar los posibles valores devueltos, qué errores se indican, y lo que debe hacer cuando llegue ese error.

Su intérprete de comandos debe soportar cualquier combinación de estos caracteres en una sola línea, siempre y cuando tenga sentido. Por ejemplo, "*ls -l* | *sort* > *resultado.txt*", en este caso debe ejecutar la salida del primer comando en la entrada del segundo y redirigir la salida del segundo comando en *resultado.txt.*

Una vez descrito el enunciado el estudiante debe entregar lo siguiente:

- El código fuente totalmente funcional del intérprete de comandos con los requerimientos antes mencionados.
- Un informe de la solución explicando el uso de las principales llamadas al sistema utilizadas.

3.4.6 Duración

Este laboratorio se pretende impartir en 4 horas de clase equivalentes a dos clases de laboratorio. La primera semana se explica el funcionamiento de las principales llamadas al sistema utilizadas para desarrollar un intérprete de comandos. También debe explicarse cómo utilizar las plantillas provistas, para finalizar se debe explicar el enunciado de los requerimientos, en este caso la implementación del intérprete de comandos. La segunda clase se debe aclarar ciertas dudas y apoyar a los estudiantes en los inconvenientes que puedan surgir a raíz de la implementación del intérprete de comandos. En la siguiente semana se debe recibir el código funcional del intérprete de comandos solicitado.

3.4.7 Documentación y ayuda

El grupo docente y los estudiantes podrán apoyarse con una aplicación para control de proyecto la cual contendrá parte de esta investigación, que provee la solución de todos los laboratorios, y el seminario referente a la misma. Además, existe un video desarrollado por esta investigación que apoya de manera audiovisual la implementación del intérprete de comandos. A continuación se describen los programas que componen la plantilla. La plantilla o código base creará una carpeta llamada intérprete de comandos, que proporciona los siguientes archivos *Makefile, shell.l* y *myshell.c.* Las funciones de los mismos serán explicadas a continuación:

- Shell.l: ofrece un programa de captura por entrada estándar (la función *getline*()), que se puede utilizar para controlar el flujo de entrada del usuario. No es necesario que modificar este archivo. El getline() devuelve un arreglo de apuntadores a cadenas de caracteres (*char* **). Cada cadena es una palabra que contiene letras, números, punto (.), barra (/), o una cadena de caracteres que contiene uno de los especiales caracteres: '<', '>' y '|'.
- myshell.c: contiene un código esqueleto de un intérprete de comandos simple. En este momento el intérprete de comandos lo único que puede realizar es leer una línea a la vez por entrada estándar. La implementación de la solución del intérprete será desarrollada en este archivo.

- Makefile: contiene todo lo necesario para compilar Shell.l y myshell.c. Con el fin de compilar y ejecutar el intérprete de comandos.

3.5 *Laboratorio 4 – Implementación de llamadas al sistema*

Debido a que Minix 3 está implementado con una arquitectura micronúcleo dividido en cuatro capas las existen dos llamadas al sistema posibles a implementar en el SO, la primera la cual es atendida por el proceso servidor indicado, dicha llamada es ejecutada en modo usuario debido a que los procesos servidores se ejecutan en este modo. La segunda que es atendida igualmente por el proceso servidor indicando, con la diferencia que dicho proceso redirecciona la llamada al núcleo del SO, para que finalmente se ejecute en modo núcleo. El grupo docente debe explicar los requerimientos planteados por este laboratorio.

3.5.1 Motivación

Este es el primer laboratorio donde los estudiantes modificaran por primera vez parte del sistema operativo, es un paso importante en la generación de conocimientos, ya que pocos estudiantes de un curso de sistemas operativos tienen la oportunidad de modificar un sistema operativo real.

3.5.2 Objetivos

El objetivo principal de este laboratorio es explicar cómo implementar ambas llamadas al sistema mencionadas con anterioridad a los estudiantes, para que los estudiantes aprendan como es el funcionamiento interno de una llamada al sistema en el SOI Minix 3 versión 3.1.6. En sí, este laboratorio tiene la finalidad de:

- Conocer las funcionalidades relacionadas a cada capa del SOI Minix versión 3.1.6.
- Conocer el funcionamiento de las llamadas al sistema del SOI Minix versión 3.1.6.
- Implementar llamadas al sistema para la capa 3 de los procesos servidores.
- Implementar llamadas al sistema para la capa 1 de los procesos del núcleo.

3.5.3 Grupo docente

Debe explicar toda la teoría asociada a las capas y llamadas al sistema en Minix 3. Así como también, debe mostrar como implementar las llamadas al sistema mencionadas con anterioridad, siendo detallista con el proceso y programas involucrados. El grupo docente debe explicar los requerimientos planteados por este laboratorio.

3.5.4 Estudiantes

El estudiante debe de asistir a clases y realizar las actividades propuestas por el grupo docente de manera de alcanzar los objetivos planteados en este laboratorio.

3.5.5 Entregables

El estudiante debe entregar una maquina virtual con las llamadas al sistema implementadas y un informe con la solución de las mismas.

3.5.6 Duración

Este laboratorio se pretende impartir en 4 horas de clase equivalentes a dos clases de laboratorio. La primera semana se explica cómo implementar las llamadas al sistema. La segunda clase se debe aclarar ciertas dudas y apoyar a los estudiantes en los inconvenientes que puedan surgir a raíz de la implementación de dichas llamadas. En la siguiente semana se debe recibir los entregables planteados.

3.5.7 Documentación y ayuda

El grupo docente y los estudiantes podrán apoyarse con una aplicación para control de proyecto la cual contendrá parte de esta investigación y el seminario referente a la misma. Además, existe un video desarrollado por esta investigación que apoya de manera audiovisual la implementación de ambas llamadas al sistema.

3.6 *Laboratorio 5 – Implementación de semáforos*

Un tópico interesante y común en sistemas operativos es la sincronización entre procesos, es por ello que se decidió dedicar todo un laboratorio para adquirir y practicar los conocimientos adquiridos en esta área. Es importante señalar que el método para la sincronización en Minix 3 es el pase de mensaje. Es por esto que se propone como laboratorio la implementación de la estructura de dato semáforo.

3.6.1 Motivación

Un tópico interesante y común en sistemas operativos es la sincronización entre procesos, es por ello que se decidió dedicar todo un laboratorio para adquirir y practicar los conocimientos adquiridos en esta área.

3.6.2 Objetivos

Los objetivos planteados para este laboratorio son los siguientes:

- Conocer los mecanismos de concurrencia y sincronización utilizados por el sistema operativo instruccional Minix versión 3.1.6.
- Entender la solución propuesta por el grupo de desarrollo de Minix versión 3 para la implementación de semáforos.
- Implementar semáforos para el sistema operativo instruccional Minix versión 3.1.6.

3.6.3 Grupo docente

Explicar cómo los mecanismos de concurrencia y sincronización son implementados por Minix 3. Además debe mostrar el funcionamiento de un proceso servidor, haciendo énfasis en el pase de mensajes. También debe esbozar cómo implementar semáforos en Minix 3, mostrando los pasos a seguir para dicha implementación; para esto debe proveer a los estudiantes la solución de semáforos en código pseudoformal, tal y como se muestra en el capítulo 7. El grupo docente debe explicar los requerimientos planteados por este laboratorio.

3.6.4 Estudiantes

El estudiante debe de asistir a clases y realizar las actividades propuestas por el grupo docente de manera de alcanzar los objetivos planteados en este laboratorio.

3.6.5 Entregables

El estudiante debe entregar una maquina virtual con la estructura de datos semáforo implementada, implementar un problema clásico de sincronización y un informe con la solución de lo anterior mencionado. La estructura de datos semáforo debe soportar las siguientes operaciones:

- *Inicialización*
- *Wait*
- *Signal*

3.6.6 Duración

Este laboratorio se pretende impartir en 6 horas de clase equivalentes a tres clases de laboratorio. La primera semana se explica los mecanismos de sincronización usados por Minix 3, así como también, el funcionamiento de un proceso servidor. Además, se explicara la solución planteada por esta investigación en código pseudoformal. La segunda y tercera clase son utilizadas para aclarar ciertas dudas y apoyar a los estudiantes en los inconvenientes que puedan surgir a raíz de la implementación. En la última semana se debe recibir los entregables planteados.

3.6.7 Documentación y ayuda

El grupo docente y los estudiantes podrán apoyarse con una aplicación para control de proyecto la cual contendrá parte de esta investigación y el seminario referente a la misma. Además, existe un video desarrollado por esta investigación que apoya de manera audiovisual la implementación de todo el proceso de implementación de los semáforos.

3.7 *Laboratorio 6 – Modificación del planificador de procesos*

Este laboratorio pretende dar a conocer el algoritmo de planificación de corto plazo utilizado por el sistema operativo instruccional Minix versión 3.1.6. Además de apoyar los conocimientos adquiridos en teoría, este laboratorio permite constatar que es importante diseñar un buen algoritmo de planificación ya que del mismo va a depender el uso óptimo del procesador.

3.7.1 Motivación

Una de las principales funcionalidades de un SO es gestionar la ejecución de procesos, es por esto que se desea que los estudiantes conozcan la importancia de esta función.

3.7.2 Objetivos

Los objetivos planteados para este laboratorio son los siguientes:

- Conocer los conceptos relacionados a la planificación de procesos en los Sistemas Operativos.
- Conocer los algoritmos de planificación de corto plazo en los Sistemas Operativos.
- Conocer el algoritmo de planificación de corto plazo utilizado por el Sistema Operativo Instruccional Minix versión 3.1.6.
- Modificar el algoritmo de planificación de corto plazo utilizado por el Sistema Operativo Instruccional Minix versión 3.1.6.

3.7.3 Grupo docente

El grupo docente debe explicar el diseño del algoritmo de planificación de Minix 3, para luego hacer referencia a las porciones de código donde esta implementado dicho algoritmo. El grupo docente debe explicar los requerimientos planteados por este laboratorio.

3.7.4 Estudiantes

El estudiante debe de asistir a clases y realizar las actividades propuestas por el grupo docente de manera de alcanzar los objetivos planteados en este laboratorio.

3.7.5 Entregables

El estudiante debe entregar lo siguiente:

- Una maquina virtual donde se encuentre las diferentes imágenes del núcleo de Minix versión 3.1.6 donde se genero el ambiente de pruebas para analizar el rendimiento del planificador de corto plazo del sistema operativo instruccional Minix versión 3.1.6.
- Un informe describiendo de forma detallada la implementación del ambiente de pruebas sobre el planificador de corto plazo del SOI Minix 3 y su respectivo análisis sobre los resultados obtenidos.

3.7.6 Duración

Este laboratorio se pretende impartir en 4 horas de clase equivalentes a dos clases de laboratorio. La primera semana se explica el algoritmo de planificación de Minix 3 y los requerimientos funcionales del actual laboratorio. La segunda clase es utilizada para aclarar ciertas dudas y apoyar a los estudiantes en los inconvenientes que puedan surgir a raíz de la modificación. En la semana siguiente se debe recibir los entregables planteados.

3.7.7 Documentación y ayuda

El grupo docente y los estudiantes podrán apoyarse con una aplicación para control de proyecto la cual contendrá parte de esta investigación y el seminario referente a la misma. Además, existe un video desarrollado por esta investigación que apoya de manera audiovisual la modificación del algoritmo de planificación.

Para poder determinar el funcionamiento del planificador de corto plazo del Sistema Operativo Instruccional Minix versión 3.1.6 debe recordar que debido a la estructura de diseño de Minix los procesos de capas inferiores poseen mayor prioridad que aquellos en capas superiores, por lo tanto, se debe asegurar la ejecuciones de programas en la capa de usuarios, en la capa de servidores y en la capa del núcleo. El ambiente de pruebas a realizar es el siguiente:

- Implementar en el espacio de usuario dos programas que generen alta carga de CPU y alta carga de peticiones de entrada/salida.
- Implementar en el espacio de servidores dos programas que generen alta carga de CPU y alta carga de peticiones de entrada/salida.
- Implementar en el espacio de núcleo dos programas que generen alta carga de CPU y alta carga de peticiones de entrada/salida.

Nota: Debe apoyarse en el Laboratorio 4 para generar las llamadas al sistema que permitan ejecutar programas en la capa 3 y capa 1 respectivamente.

Además, el ambiente de pruebas a realizar debe calcularse los tiempos de ejecución de cada uno de los programas anteriormente propuestos utilizando el planificador de corto plazo de Minix definido con 16 colas y luego este debe modificarse con 8 colas y ejecutar nuevamente las pruebas.

3.8 *La planificación de los laboratorios*

Para resumir la planificación planteada por esta investigación, la cual está diseñada para un semestre de 16 semanas vea la siguiente:

semana 1	Laboratorio 0
semana 2	Laboratorio 1
semana 3	Laboratorio 2
semana 4	Laboratorio 3
semana 5	
semana 6	Laboratorio 4
semana 7	
semana 8	Laboratorio 5
semana 9	
semana 10	
semana 11	Laboratorio 6
semana 12	

Tabla 3.1 Planificación de los laboratorios por semanas

4 Herramientas de desarrollo

El presente capítulo realiza una descripción los sistemas operativos y aplicaciones utilizadas para la adecuación del SOI Minix 3 al curso de Sistemas Operativos de pregrado de la Universidad Central de Venezuela. Los principales sistemas operativos y aplicaciones utilizadas son el SOI Minix 3, las aplicaciones usadas fueron VMware Workstation, el IDE (entornos de desarrollo integrados) eclipse y el software de captura de vídeo de pantalla Camtasia Studio. Para concluir este capítulo se muestra la metodología utilizada para el desarrollo de la herramienta educativa, especificando la misma por cada laboratorio propuesto.

4.1 *Lenguaje de programación C*

El leguaje de programación C es un lenguaje de programación desarrollado entre 1969 y 1973 por Dennis Ritchie en los Laboratorios Telefónicos Bell para su uso con el Unix sistema operativo. El origen de C está estrechamente ligado al desarrollo del sistema operativo Unix, ya que originalmente estaba implementado en lenguaje ensamblador para luego ser reescrito casi todo en C. La mayoría de los sistemas operativos están escritos bajo este lenguaje ya que permite un control a muy bajo nivel, los compiladores suelen ofrecer extensiones al lenguaje que posibilitan mezclar código en ensamblador con código C[7].

4.2 *VMware Workstation*

VMware es un sistema de virtualización por software. Un sistema virtual por software es un programa que simula un sistema físico (un computador, un hardware) con unas características de hardware determinadas. Cuando se ejecuta el programa (simulador), proporciona un ambiente de ejecución similar a todos los efectos a un computador físico (excepto en el puro acceso físico al hardware simulado), puede simular la CPU (una o más), BIOS, tarjeta gráfica, memoria RAM, tarjeta de red, sistema de sonido, conexión USB, disco duro (puede ser más de uno también), etc. VMware Workstation se ejecuta en Microsoft Windows, Linux y Mac OS X.

VMware Workstation permite a los usuarios ejecutar varias instancias de x86 o x86-64. VMware inserta directamente una capa de software en el hardware del computador o en el sistema operativo host. Esta capa de software crea máquinas virtuales y contiene un monitor de máquina virtual que asigna recursos de hardware de forma dinámica, para poder ejecutar varios sistemas operativos de forma “simultánea” en un único computador físico de manera transparente.

---

[7] “Programming in C” - http://www.cs.cf.ac.uk

Además, VMware ofrece una sólida plataforma de virtualización que puede ampliarse por cientos de dispositivos de almacenamiento y computadores físicos interconectados para formar una infraestructura virtual completa. Es software propietario pero existen versiones gratuitas como VMware Player. VMware Workstation 7.0.1 fue publicado el 29 de enero de 2010 y VMware Player 3.0.0 fue publicado el 4 de diciembre de 2009[8].

En nuestra investigación se utilizó VMWare debido a que:

- Es una herramienta ampliamente utilizada en los laboratorios docentes de la Escuela de Computación.
- Su instalación, configuración y manejo de máquinas virtuales es muy sencilla e intuitiva para los usuarios.
- Se puede obtener fácilmente a través del Centro de Computación.
- Es una aplicación multiplataforma.

4.3 *IDE eclipse*

Eclipse es un entorno de desarrollo integrado de código abierto multiplataforma. Está escrito en su mayoría en Java y se puede utilizar para desarrollar aplicaciones en Java. Además, por medio de diversos plugins también se puede desarrollar en otros lenguajes de programación como Ada, C, C++, COBOL, Perl, PHP, Python, Ruby (incluyendo Ruby on Rails framework), Scala y Scheme.

La base de código inicial se originó a partir de VisualAge. En su forma predeterminada, es para los desarrolladores de Java, que consiste en las herramientas de desarrollo de Java (JDT). Los usuarios pueden ampliar su capacidad mediante la instalación de plugins escritos para la plataforma de software Eclipse, tales como kits de herramientas de desarrollo para otros lenguajes de programación, y puede escribir y contribuir con sus propios plugin. Distribuido bajo los términos de la Licencia Pública de Eclipse, Eclipse es un software libre y de código abierto. La última versión liberada es la 3.6.1 Helios, el 24 de septiembre de 2010, la cual puede descargarse desde su página web principal[9].

En nuestra investigación se utilizó el IDE eclipse debido a que:

- Es una aplicación multiplataforma.
- Al igual que VMware los estudiantes de la Escuela de Computación están familiarizados con el IDE debido a que en varias materias del pensum es utilizado como herramienta de desarrollo.

---

[8] "VMware" - http://www.vmware.com/

[9] "eclipse" - http://www.eclipse.org/

- Su instalación, configuración e instalación de plugins es sumamente sencillo en gran parte a su menú de instalación.
- Se pueda descargar gratuitamente desde su página web principal.

4.4 *Camtasia Studio*

Camtasia Studio es un software de captura de vídeo de pantalla, publicado por TechSmith. Puede ejecutarse sobre los sistemas operativos Windows y Mac OS X. La licencia es propietaria. Cantacsia permite crear screencasts, los screencasts ayudan a demostrar y enseñar el manejo de un software. La creación de un screencast ayuda a los desarrolladores de software a mostrar su trabajo, es una manera muy fácil de enseñar los conocimientos a través de un video. Es una herramienta útil tanto para los usuarios comunes de software, así como también, para ayudar a informar de errores o para mostrar a otros cómo se realiza una determinada tarea en un entorno de software específico. Los screencasts son herramientas excelentes para aprender a usar las computadoras y/o aplicaciones, y muchos tutoriales hoy en día se encuentran con esta tecnología, que permite enseñar a los usuarios desde la comodidad de su hogar, estudio o trabajo.

Teniendo en cuenta el alto costo de los instructores/profesores y la básica instrucción que se proporciona en computación, probablemente los screencasting se conviertan en una técnica muy popular para impartir conocimientos de alta calidad a un bajo costo.

Una desventaja es que la mayoría de los programas screencasting comerciales están realizados para Microsoft Windows, lo cual es una limitante para realizar videos de aplicaciones OpenGL, aunque Demo Builder, Fraps, y Guncam Growler puede hacer frente a esto.

Una de las ventajas de Camtasia es que el tiene una interfaz muy intuitiva. También permite editar los videos grabados minimizando el tiempo invertido para la grabación y proporcionando una alta tolerancia a errores por parte del usuario. Provee la facilidad de insertar audio aun cuando el video ya ha sido grabado desligando lo visual de lo auditivo. Para el usuario también está disponible una gama de herramientas de presentación y edición de videos. Para utilizar Camtasia el usuario define el área de la pantalla o la ventana que se va a capturar toda la pantalla o se puede grabar en su lugar. Esto se establece antes de empezar la grabación. Es posible grabar audio desde un micrófono o los altavoces y Camtasia Studio le permite colocar imágenes de la webcam en la pantalla.

4.5 *Metodología*

Será descrita la metodología utilizada para el desarrollo de los proyectos propuestos anteriormente. Se utilizara la siguiente metodología de adecuación de software (29).

4.5.1 Evaluación y Elección

En este paso inicial, se parte de la existencia de múltiples alternativas de software (NachOS, PintOS, GeekOS, Minix, xv6 y OS/161) para resolver el problema planteado. Lo cual hace referencia a la existen distintas alternativas que encajan con la solución que se desea tener. Considerando que estas piezas de software pueden cumplir o no todos los requisitos funcionales, es de vital importancia evaluar cual se adapta mejor al curriculum académico de los cursos de Sistemas Operativos de la Escuela de Computación, ya que habrá menos esfuerzo y tiempo de adecuación. Esta parte del proceso ya fue desarrollada durante la elaboración de este seminario y el SOI seleccionado fue Minix 3, como se mencionó y justificó con anterioridad.

4.5.2 Proceso de desarrollo

El objetivo de esta fase en sí constituye en obtener una versión del producto, que sea estable y funcional, a pesar de que sea una versión incompleta. En este caso, se tomó los requerimientos necesarios para instalar Minix 3; se documento este proceso indicando los procedimientos necesarios para instalación, configuración y manejo de la herramienta educativa. Los cuales pueden observarse en el marco teórico de este documento. Otro objetivo de esta fase es evaluar los posibles laboratorios que se pueden desarrollar a partir de este SOI, los cuales ya fueron descritos anteriormente. Estos laboratorios serán llamados lista de requisitos. En cada iteración de esta metodología se generó una versión instalable, hasta que los objetivos fueron cubiertos. Esta fase está compuesta por las siguientes etapas: Este esquema de planificación se puede apreciar de mejor manera en la Figura 4.1

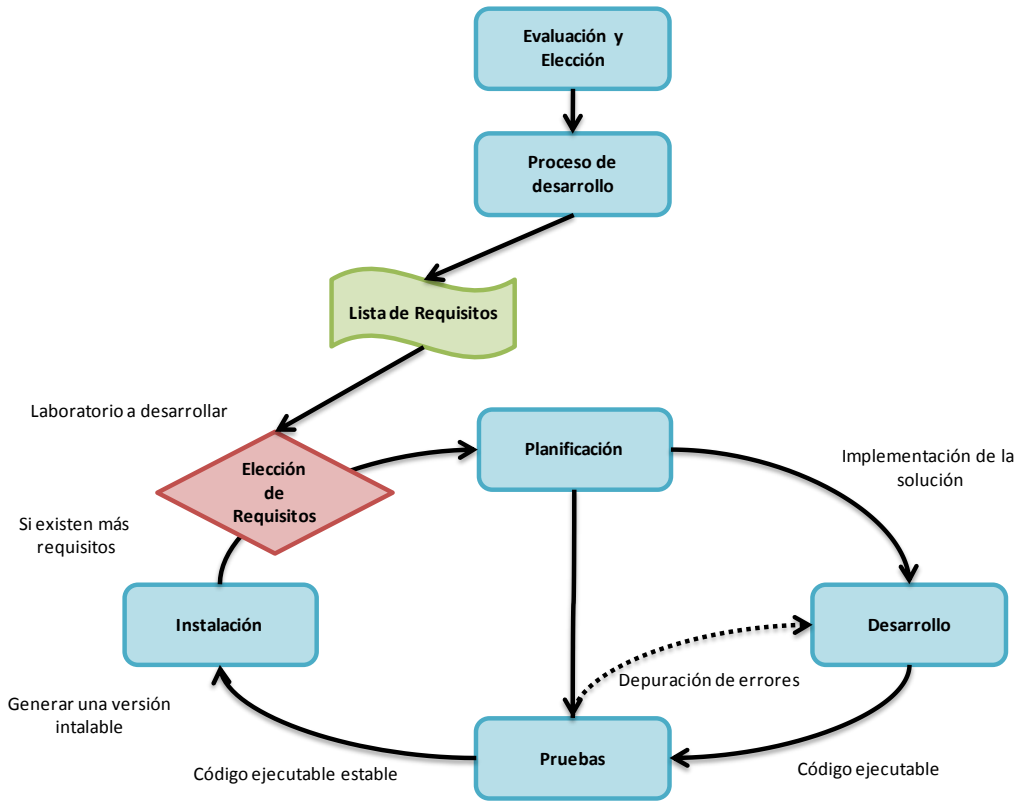


Figura 4.1 Metodología de desarrollo de software

4.5.3 Iteraciones

Esta fase comienza con la elección de uno o más objetivos, y estos deben dirigirse hacia un mismo requerimiento funcional. Las iteraciones se proponen en periodos de 7 a 12 días. Las primeras iteraciones se dedican principalmente en el conocimiento de parte de los desarrolladores sobre la herramienta a utilizar como diseño, arquitectura, código fuente y estructuras de datos. Al culminar una iteración, no siempre se genera una nueva versión. Una vez dominado la parte teórica de Minix 3, en las primeras iteraciones. Se procede a la escogencia de un proyecto de laboratorio, para dar solución y documentar el procedimiento. Durante una iteración se realizan las siguientes actividades:

- Planificación: se analizan los requerimientos escogidos de cada proyecto de laboratorio, en base a la complejidad de las tareas planteadas.
- Desarrollo: esta actividad consiste en la modificación y/o desarrollo del código fuente. Se analizan los requerimientos, se identifican los componentes que están involucrados en la obtención de la solución. Posteriormente se implementa la solución planteada. Por último, se realizan pruebas concernientes sobre la correctitud de los cambios aplicados.
- Pruebas: estas se pueden realizar en un primer nivel sobre los requerimientos involucrados en la iteración y un segundo nivel sobre el funcionamiento del laboratorio.

- Instalación: durante esta actividad se establecen un conjunto de tareas que permitan a nivel de usuario interactuar con la aplicación. Esta versión incluye al sistema y sus nuevos componentes.

Ahora se procederá a describir las iteraciones realizadas para alcanzar los objetivos específicos propuestos:

4.5.4 Iteración 1

En esta iteración se empleo para estudiar y entender toda la documentación referente a Minix 3, la cual duró 10 días.

- Planificación: se decidió utilizar las fuentes más confiables para la investigación las cuales fueron la Página web de Minix (14) y el Libro oficial (15).
- Desarrollo: en la página web se pudo obtener la imagen de Minix versión 3.1.6, además de los requerimientos necesarios para instalar Minix. También, se encontró información de cómo instalar Minix y el entorno de desarrollo para la implementación de los laboratorios. En el libro se encontró toda la información de la estructura y funcionamiento de Minix 3, así como también, gran parte de la documentación del código fuente.
- Pruebas: para esta iteración no aplican las pruebas.
- Instalación: con la información obtenida se generó la primera versión instalable de Minix 3.

4.5.5 Iteración 2 y Iteración 3

En esta iteración se procedió a la realización del laboratorio del gestor de arranque de Minix 3, cada iteración duró 12 días.

- Planificación: los requerimientos para esta iteración son:
    - Documentar todo el proceso de arranque de un sistema operativo.
    - Identificar según lo anterior las piezas claves en Minix que desarrollan esa tarea.
    - Formular el laboratorio y realizar la solución del mismo.
- Desarrollo: para el desarrollo de este laboratorio se hizo una documentación minuciosa del proceso que realiza Minix para poder ejecutar el núcleo. Además, se describió a fondo cómo funcionan los archivos *masterboot.s* y *bootblock.s*. Para culminar se planteó como laboratorio una serie de preguntas que los estudiantes debe responder para entregar un informe asociado.
- Pruebas: para esta iteración no aplican las pruebas.
- Instalación: no aplica instalación para esta iteración.

4.5.6 Iteración 4

Se planteó la realización de un laboratorio que deba implementar un intérprete de comandos simple, la cual duró 10 días.

- Planificación: los requerimientos para esta iteración son:
    - Documentar las principales llamadas al sistema que ofrece Minix 3 que dan soporte a la implementación de un intérprete de comandos simple.
    - Realizar el enunciado del laboratorio.
- Desarrollo: Se implemento un intérprete de comandos simple, y se documento las principales llamadas al sistema que dan soporte a la implementación de uno, con eso se hace referencia a la sintaxis y semántica de las mismas. Se redacto el enunciado asociado al laboratorio. Para culminar se documentó todo el proceso de implementación de la solución.
- Pruebas: Se compiló el programa myshell.c para probar la implementación del intérprete de comandos y se realizaron las pruebas necesarias para verificar su correcto funcionamiento.
- Instalación: para esta iteración no aplica instalación.

4.5.7 Iteración 5

Se sugirió la realización del laboratorio de llamadas al sistema Minix 3, la cual duró 12 días.

- Planificación: los requerimientos para esta iteración son:
    - Documentar los tipos de llamadas al sistema en Minix 3, cómo es su funcionamiento y cuáles son las características de las mismas. Además, documentar cómo se implementas dichas llamadas.
    - Identificar cuáles son los programas involucrados en la implementación de llamadas al sistema.
    - Implementar las llamadas al sistema en Minix, y realizar el enunciado del laboratorio.
- Desarrollo: Se implemento los dos tipos posibles de llamadas al sistema en Minix 3 con ayuda del libro oficial. Además, se realizó el enunciado del laboratorio correspondiente. Concluyendo con toda la documentación, paso por paso, de la solución.
- Pruebas: Se recompiló el núcleo y se probo el correcto funcionamiento de las llamadas.
- Instalación: como se mencionó se genero un nuevo release de Minix 3 con las llamadas al sistema implementadas.

4.5.8 Iteración 6 y Iteración 7

Para este laboratorio se quiso modificar el núcleo para que diera soporte a semáforos, cada iteración duró 12 días.

- Planificación: los requerimientos para esta iteración son:
    - Identificar una manera de implementar semáforos en Minix 3 y realizarla.
    - Realizar el enunciado del laboratorio.
    - Documentar todo el proceso de implementación de la solución.
- Desarrollo: Se modificó el núcleo de Minix y un proceso servidor para que pudiera soportar semáforos, en especial implementar las principales primitivas que permiten manipularlos. Se redacto el enunciado del laboratorio y todo el proceso de implementación.
- Pruebas: Se compiló el núcleo para probar la implementación realizada y se realizaron las pruebas necesarias para verificar su correcto funcionamiento. Se hicieron dos archivos de prueba para verificar la implementación.
- Instalación: se recompiló el núcleo y se verifico que arrojara una nueva versión.

4.5.9 Iteración 8

Para este laboratorio se planteó modificar el algoritmo de planificación de Minix 3, la cual duró 12 días.

- Planificación: los requerimientos para esta iteración son:
    - Identificar los archivos involucrados en la implementación de dicho algoritmo.
    - Realizar el enunciado del laboratorio.
    - Documentar todo el proceso de implementación de la solución.
- Desarrollo: Se identifico y modifico los archivos que implementan el algoritmo de planificación de Minix 3. Y realizo el enunciado del laboratorio.
- Pruebas: Se compiló el núcleo para probar los cambios realizados y se realizaron las pruebas necesarias, para verificar las modificaciones realizadas.
- Instalación: se recompiló el núcleo y se verifico que arrojara una nueva versión.

5 Instalación de Minix y entorno de desarrollo

Este laboratorio esta esbozado para sentar las bases de las herramientas necesarias para desarrollar a lo largo del curso todos los laboratorios propuestos por el grupo docente. Este es un punto clave ya que permite engranar todas las aplicaciones y el SOI Minix 3 para un facilitar en gran medida el desarrollo de la implementación de los laboratorios. Serán descritos los pasos que se dieron para la instalación de Minix versión 3.1.6, así como también para armar y configurar el entorno de desarrollo.

5.1 *Instalación de Minix*

Para la instalación de Minix versión 3.1.6 se utilizaron los siguientes programas:

- La imagen de Minix versión 3.1.6
- El sistema de virtualización VMware Workstation versión 6.5.1

5.1.1 Configuración de la máquina virtual

Se debe crear una máquina virtual en VMware siguiendo la configuración típica configurando las siguientes características:

- Indicarle a la máquina que debe iniciar desde la imagen .iso de Minix versión 3.1.6
- La memoria RAM (Random Access Memory)debe ser de al menos 16 MB, 600 MB como mínimo de disco duro y un procesador Pentium o compatible.

5.1.2 Instalación de Minix versión 3.1.6

A continuación, se exponen los pasos a seguir para la instalación de Minix 3.1.6, es importante destacar que este procedimiento esta mejor explicado en los videos tutoriales realizados por esta investigación. Asumiendo que el proceso de configuración de su máquina virtual se ha llevado a cabo de manera satisfactoria. Con esto se supone que usted ha creado al menos una partición en su disco duro donde residirá la imagen del SO. Así que los siguientes pasos describirán la instalación estándar de Minix 3:

- Se debe configurar la BIOS (Basic Input Ouput System) para que la unidad de CD-ROM sea el primer dispositivo arrancable del sistema.
- Introduzca en la unidad de CD-ROM el Live CD de Minix o en su defecto utilizar la imagen iso descargada desde la página Web. Todo eso con la finalidad de que al momento de arranque la BIOS pueda cargar el SO desde esta unidad.
- Si la ejecución del disco de instalación hasta ahora ha sido efectiva entonces se presentara la consola de comandos esperando que introduzcamos un nombre de usuario, y colocamos al usuario “*root*”. Debe recordar que en los SO derivados de Unix este es el “superusuario”.
- Al ingresar al sistema se introduce el comando “*setup*”, para iniciar la instalación de Minix 3.

- Seguidamente al ejecutar el comando "*setup*" el sistema preguntará sobre cuál es la configuración del teclado, colóquela según sea la misma.
- El sistema preguntará sobre el modo de instalación que se desea llevar a cabo, es decir, una instalación estándar o avanzada, se seleccionará la primera con presionar la tecla de "*enter*".
- El sistema hará un chequeo sobre los dispositivos de almacenamiento que el sistema ha detectado y pedirá que seleccione que dispositivo se utilizará.
- Se debe escoger en cual región del disco se alojará a Minix 3, inmediatamente dará un mensaje informativo, el cual indica que a partir de este paso el proceso de instalación se llevara a cabo y no hay oportunidad de redimirse; Por supuesto se elige aceptar.
- Luego, se preguntara la manera en la que se va a instalar, seleccionará la opción completa o full del sistema.
- En los siguientes pasos se preguntará el tamaño del directorio */home* que es aquella partición donde se encuentran los archivos de los usuarios del sistema.
- Ahora se requerirá el tamaño de bloque que desea manejar y muestra que la opción que por defecto es de 4 KB (Kilobyte) y la cual es la que se usará.
- Ahora el sistema empezara a copiar los resultados de la instalación al disco.
- Luego solicitará la información sobre tarjeta de red que posee nuestro equipo. En caso de poseer unan tarjeta de red sea soportada por Minix, el programa de instalación preguntará qué tipo de configuración se va a manejar sea modo estático o a través de un servidor DHCP (Dynamic Host Configuration Protocol).
- Al culminar estos pasos se nos llevara nuevamente al intérprete de comandos y debe ejecutar el comando "*reboot*", "*shutdown*" o "*halt*" para salir del programa de instalación.

5.2 *Instalación del entorno de desarrollo*

Para este paso es indispensable obtener el ejecutable de eclipse classic 3.5.0, posteriormente debe instalarlo. Luego instalar los paquetes necesarios para desarrollar en el lenguaje C y para poder realizar la conexión SSH (Secure Shell) con Minix. Por último, debe instalarle un paquete a Minix que de soporte a la conexión SSH. Todos estos pasos son descritos en el video tutorial de instalación de Minix desarrollado por esta investigación.

6 Introducción a Minix 3

En este laboratorio se da a conocer el SOI Minix 3, así como también, una introducción de su historia, de su estructura, de las principales características, entre otros.

6.1 *Sistema Operativo Minix*

Minix es un sistema operativo, distribuido conjuntamente con su código fuente y desarrollado por Andrew Tanenbaum. La última versión oficial de Minix es la 3.1.8 que se puede obtener directamente desde su página Web oficial[10]

En esta sección y las posteriores se pretende dar un esbozo acerca del funcionamiento interno de Minix 3. Se explicará el diseño y la arquitectura de Minix 3, dicha información será útil para la comprensión de los capítulos posteriores.

Esta última versión de Minix puede ser definida, como un nuevo SO de código abierto, cuya finalidad principal es ofrecer alta confiabilidad, flexibilidad y seguridad. Está basada en las versiones anteriores de Minix, sin embargo, posee diferencias significativas. Minix 1 y Minix 2 fueron concebidas como herramientas de enseñanza. Minix 3 añade nuevos objetivos con la finalidad de ser utilizado como un SO en computadoras con recursos limitados, embebidos y para aplicaciones que requieren alta confiabilidad en sus ambientes de ejecución. Actualmente Minix 3 se distribuye con una licencia similar a BSD, lo que permite su estudio y modificación.

6.1.1 La Historia de Minix

En los inicios del SO Unix específicamente en su versión 6, las universidades adoptaron esta pieza de software como herramienta educativa en sus cursos de SO debido a que podía obtener su código fuente. Debido al auge que empezó a tener Unix, éste se convirtió en un software comercial que podría generar ganancias a sus desarrolladores, en su caso a AT&T. Por lo que en su siguiente versión su licencia fue modificada, limitando el acceso a su código fuente. Esta medida desfavoreció a las universidades debido a que no podían utilizar la última versión como herramienta de estudio en los cursos de Sistemas Operativos, ocasionando que las dinámicas pedagógicas de los mismos fuesen orientadas más a los espacios teóricos-abstractos de la materia (15).

En búsqueda de una solución a la situación anterior, surge un proyecto que plantea la necesidad en desarrollar una herramienta educativa capaz de generar conocimientos de diseño e implementación en SO, la cual se conoce como SOI (Sistemas Operativos Instruccionales). Este proyecto bandera se

[10] "Minix 3" - **¡Error! Referencia de hipervínculo no válida.**

conoció como Minix, desarrollado por Andrew Tanenbaum, quien escribió un SO completo desde cero, que posee una interfaz de usuario similar a Unix, pero con una estructura diferente para evitar problemas relacionados a las licencias de software.

Debido al enfoque de diseño de Minix, es decir, de carácter pedagógico, su autor y creador no permitía que las modificaciones al mismo fuesen drásticas, motivado a que podría acarrear complicaciones en el sistema. Además, podría impedir el estudio y comprensión durante un semestre.

Minix 3 es conocido en los ambientes académicos computacionales por ser desarrollado por uno de los principales investigadores de los SO. A su vez, es famoso debido a que del mismo surge uno de los proyectos de SO más populares de las últimas dos décadas conocido como Linux, que fue desarrollado por Linus Torvalds.

El nombre de Minix viene de un juego de palabras que su autor llama como mini-Unix, es decir, es un SO lo suficientemente pequeño que puede ser estudiado y comprendido por cualquiera que se lo proponga sin necesidad de poseer un amplio conocimientos en computación.

6.1.2 Versiones de Minix

El SO Minix 1 en su inicio desarrollado por Andrew Tanenbaum, fue implementado para ejemplificar los principios explicados en su libro “Sistemas Operativos: Diseño e Implementación” publicado en el año 1987. En este se puede conseguir una parte del código fuente del núcleo, el controlador de memoria y el sistema de archivos (15).

Esta versión de Minix 1 fue desarrollada para trabajar sobre las arquitecturas IBM PC e IBM PC/AT que eran las plataformas más comunes para la época. Minix fue actualizado para ofrecer soporte al MicroChannel IBM/PS2 y también a las arquitecturas Motorola 68000 y SPARC. Debido a la popularidad del proyecto surgió una modificación no oficial de Minix que fue adaptado para las arquitecturas compatibles con Intel 80386, National Semiconductor NS32532, ARM y procesadores INMOS Transputer.

Minix 2 fue lanzado oficialmente en el año 1997, ofreciendo compatibilidad con las arquitecturas x86 y SPARC. Al igual que su versión anterior este lanzamiento posee una publicación bibliográfica escrita por Andrew Tanenbaum y Albert Woodhull. Minix 2 añadió compatibilidad con POSIX, soporte para arquitecturas Intel 80386. También aparecieron modificaciones no oficiales de Minix 2 dándole soporte a arquitecturas basadas en el 68020 ISICAD Prisma 7000 y las basadas en Hitachi SH3.

Minix-vmd es una variante de Minix 2 para procesadores compatibles con la arquitectura Intel IA-32 que fue creado por dos investigadores de la Universidad Vrije de Amsterdam, que añadió módulos de memoria virtual y a su vez soporte para el sistema grafico X Window.

Minix 3 fue anunciado públicamente en el año 2005 por Andrew Tanenbaum y al igual que sus antecesores posee una publicación bibliográfica “Operating Systems: Design and Implementation” en su tercera edición. Esta nueva versión fue completamente rediseñada para ser utilizada como un SO para computadoras con recursos de hardware limitados y para aplicaciones que requieren de un ambiente de alta confiabilidad (15).

Actualmente Minix 3 soporta solo arquitecturas derivadas a la Intel IA-32 y se encuentra disponible en Live CD que es una características bastante común hoy día que permite utilizar el SO sin necesidad de ser instalado en la maquina real.

6.1.3 Acerca de Minix 3

Minix 3 es un SO de código abierto cuya principal característica de diseño es la de ser un sistema altamente confiable, flexible y seguro, como se mencionó con anterioridad. Esta versión de Minix se puede considerar pequeña, debido a que la porción de código que se ejecuta en modo núcleo posee un promedio de 6000 líneas de código fuente, y aquellas piezas que se ejecutan en el modo usuario se dividen en pequeños módulos, aislados unos de otros, es decir, su ejecución es mutuamente excluyente.

Para entender el concepto anterior observe la siguiente situación, si cada controlador de un dispositivo se ejecuta como proceso independiente entonces al ocurrir un fallo en su ambiente de ejecución este proceso tendrá un fallo y tendrá que levantarse nuevamente. Pero sin comprometer la integridad del sistema. Esto ocurre debido a que el código fuente del controlador no se encuentra incluido dentro del núcleo debido al esquema micronúcleo que utiliza Minix 3. Normalmente el código fuente de los controladores son desarrollados por terceros que desconocen en su totalidad el diseño, implementación y desarrollo del SO en cuestión, por ejemplo OS X, Linux o Microsoft Windows (15).

De hecho, la mayoría de las veces cuando un controlador falla se sustituye automáticamente sin requerir la intervención del usuario, sin necesidad de reiniciar el sistema, y sin afectar los programas en ejecución. Debido a esta característica (pequeña cantidad de código del núcleo), se mejoran en gran medida la fiabilidad del sistema.

Uno de los principales objetivos de Minix 3 es la fiabilidad. A continuación se discuten algunos de los principios más importantes que mejoran la fiabilidad de Minix 3. Al ser mejorada la fiabilidad intrínsecamente se mejora la seguridad, ya que la mayoría de las fallas de seguridad se deben a que los atacantes explotan los errores en el código.

6.1.4 ¿Es Minix 3 un SO confiable?

Uno de los principales objetivos de MINIX 3 es la confiablidad. A continuación se discuten algunos de los principios más importantes que mejoran la confiablidad de MINIX 3. Estos principios también mejoran la

seguridad, ya que la mayoría de las fallas de seguridad se deben a los atacantes que explotan los errores en el código, al ser mejorada la confiablidad intrínsecamente se mejora la seguridad. Algunas de las medidas para garantizar la confiablidad son las siguientes (15):

- Reducir el tamaño del núcleo: algunos SO que poseen núcleos monolíticos por ejemplo, BSD, GNU/Linux y Microsoft Windows. Estos poseen núcleos que están escritos por millones de líneas de código fuente, para verificar la correctitud del mismo el procedimiento sería engorroso. Para solucionar esto Minix 3 tiene aproximadamente 6000 líneas de código fuente del núcleo que son ejecutables.
- Enjaular los posibles errores: en los SO comerciales que poseen núcleos monolíticos, los controladores de los dispositivos del computador residen en el núcleo del sistema. Lo anterior conlleva a que en la base del sistema en su punto más crudo de ejecución es instalado código fuente que son desarrollados por los fabricantes de los dispositivos. De los cuales no se puede medir su calidad ya que estos son ajenos al desarrollo del SO. Lo anterior es solventado en Minix 3 porque cada controlador de dispositivo es ejecutado en el SO como un proceso independiente de modo usuario, es decir, se garantiza de que en caso de que el código pueda contener un error este no podrá afectar el sistema.
- Limitar el acceso a memoria por medio de los controladores: En los SO monolíticos, un controlador puede escribir cualquier palabra en memoria, alguna de estas palabras pueden ser erróneas. En Minix 3 el sistema de archivos o el controlador le pide al núcleo escribir a través del descriptor, lo que hace imposible que escriban a las direcciones fuera del búfer.
- Tolerancia a bucles infinitos: si un proceso durante su ejecución entra en un bucle infinito, el planificador gradualmente irá reduciendo su prioridad hasta que se convierta en un proceso inactivo o IDLE (proceso con menor prioridad). Eventualmente el servidor reencarnación se percatara que el proceso no responde a los mensajes y podrá ser reiniciado.
- Restringir el acceso a las funciones del núcleo: Los controladores de dispositivos obtienen los servicios del núcleo a través de la realización de llamadas del núcleo. El núcleo de Minix 3 tiene un mapa de bits para cada controlador que especifica la llamada a realizar y verifica si ésta autorizada.
- Servidor reencarnación: un proceso especial, denominado servidor reencarnación, de forma periódica verifica cada controlador. Si el controlador muere o no responde correctamente a las peticiones, el servidor reencarnación automáticamente lo sustituye por una copia nueva. La detección y el reemplazo de los controladores que no funcionan son automáticos, sin intervención del usuario.

6.1.5 Mejoras sobre Minix 3

Minix 3 posee varias mejoras con respecto a su versión anterior Minix 2 y entre ellas podemos nombrar las siguientes (15):

- Instalación del SO a través de un Live CD.
- Soporte al sistema X Window.
- Soporte de 4 GB de memoria principal.
- Inclusión de un servidor de información para procedimientos de depuración.
- El servidor reencarnación.
- El núcleo ha sido reescrito, optimizado y depurado a 6000 líneas de código fuente.
- Cada controlador de dispositivo es ejecutado como un proceso de usuario exceptuando el manejador del reloj del procesador.
- Mecanismo de comunicación no bloqueantes, es decir, asincrónicos.
- El planificador del procesador ha sido modificado.

6.1.6 Objetivos de Minix 3

En secciones anteriores se comentó que las versiones anteriores a Minix 3 fueron desarrolladas con un enfoque puramente educativo siendo conocido mas como un SOI que como uno comercial. Sin embargo, Minix 3 tiene ambos enfoques, el educativo y el comercial; esto ocasiono grandes expectativas debido a que este SO fue lanzado para satisfacer las expectativas en los siguientes mercados (14):

- Ofrecer un ambiente de ejecución para aquellas aplicaciones que requieren alta confiabilidad.
- Ofrecer un SO para los proyectos de OLPC (One Laptop per Children) y también para su homologo el proyecto Magallanes.
- Ser utilizado como SO para sistemas embebidos.
- Como herramienta educativa, a pesar de que está apuntándose a convertirse en un SO comercial el tamaño de núcleo es lo suficientemente pequeño para ser utilizado como herramienta educativa en las instituciones académicas.

6.1.7 Estructura de Minix 3

El SO Minix 3 fue implementado con una arquitectura micronúcleo dividido en cuatro capas. Existen al menos cinco maneras en las que su núcleo puede ser estructurado, tales como: monolítico, en capas, máquina virtual, exonúcleos y arquitectura cliente/servidor (15). Minix 3 combina la estructura basada en capas en conjunto con la arquitectura cliente/servidor. La arquitectura basada en capas divide el sistema en una serie de niveles que implementan funciones específicas. Por lo tanto, es habitual que las capas más altas dependan de los servicios ofrecidos por otras capas de nivel inferior. Minix 3 tiene cuatro capas, cada una con una función específica y bien definida (15).

Acorde a las estructuras micronúcleo gran parte de las funcionalidades importantes del SO son implementados como servidores que son ejecutados por separado. Entre los servicios fundamentales que provee la estructura micronúcleo son la administración del espacio de direcciones, la administración de hilos, comunicación entre procesos y la administración de los temporizadores del sistema.

El SO Minix implementa una arquitectura micronúcleo separada en capas como se muestra en la Figura 6.1. Además, como se ha comentado con anterioridad, Minix 3 está estructurado en cuatro capas, para visualizar cada una vea la Figura 6.1 donde se detallan. Como puede observarse la única capa que se ejecuta en modo núcleo es la capa 1.

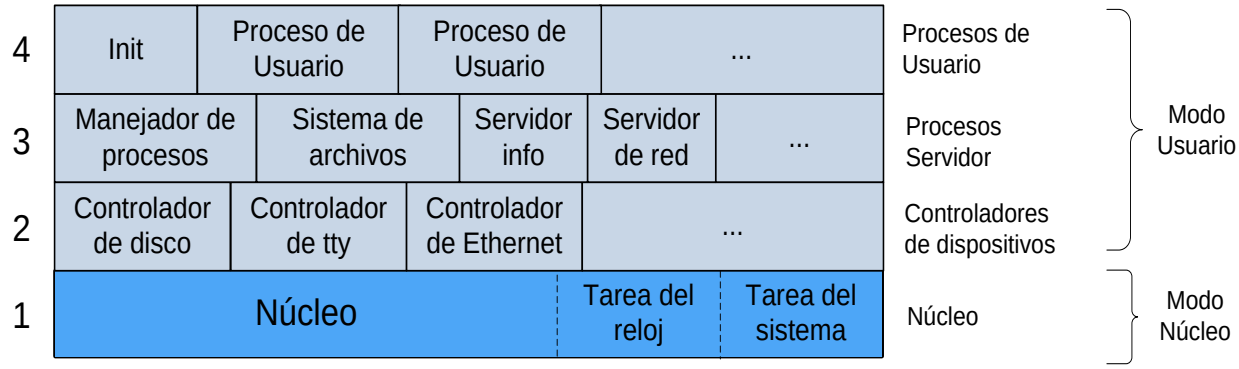


Figura 6.1 Estructura de Minix 3

Capa 1 (El núcleo): Esta capa provee los servicios de más bajo nivel que son necesarios para la ejecución del sistema. Entre ellos se incluyen la gestión de interrupciones, planificación y comunicación. La parte que ofrece servicios de más bajo nivel de esta capa, que trata con interrupciones y otros aspectos muy dependientes del hardware, está escrita en lenguaje ensamblador, mientras que el resto de funcionalidades están escritas en C. Esta capa se encarga de lo siguiente:

- Gestionar las interrupciones.
- Salvar y restaurar registros.
- Planificar procesos.
- Ofrecer servicios a la capa superior.
- Funciones de comunicación y mensajes.

Capa 2 (Controladores de dispositivos): En esta capa se encuentra el código que se encarga de las tareas de entrada/salida y da soporte a ciertas tareas que no pueden realizarse a nivel de usuario. Como por ejemplo el controlador de disco, controlador de tty y controlador de Ethernet, etc. Además, en esta capa se encuentran los controladores de dispositivos, para dar soporte a periféricos como discos duros, teclados, impresoras, lectores de CD-ROM.

Capa 3 (Servidores): Esta capa ofrece servicios que son utilizados por los programas que se ejecutan en la capa superior. Los procesos en esta capa pueden acceder a los servicios de la capa dos (controladores de dispositivos) pero los programas de la capa cuatro no tienen acceso directo a los procesos de la capa dos. Ejemplos de algunos de estos servicios incluyen: manejador de procesos, sistema de archivos, servidor info, servidor de red, etc.

Capa 4 (Procesos de usuario): Esta capa comprende la sección de usuario de Minix 3 en la que son ejecutados los programas de usuario. Estos programas utilizan los servicios que ofrecen las capas de nivel inferior. Los programas que se encuentran habitualmente en esta capa incluyen demonios de varios tipos, terminales, intérprete de comandos y cualquier otro programa que el usuario quiera ejecutar. Los proceso de esta capa tienen el nivel más bajo de privilegios para acceder a los recursos y normalmente acceden a ellos a través de los servicios que ofrecen las capas inferiores. Por ejemplo, un usuario podría ejecutar la orden *traceroute*, que necesita usar el controlador de red. La orden *traceroute* no invoca directamente al controlador de red. En su lugar, pasa a través del servidor de archivos; debido a que el intérprete de comandos está en la capa 4 y no puede comunicarse directamente con la capa 2, en su lugar solicita el servicio a través de los servidores (en este caso el servidor de archivos).

6.1.8 Ventajas de la arquitectura

Algunos de los beneficios más importantes de esta arquitectura de capas se describen en detalle a continuación (15):

- **Modularidad**: el sistema está bien estructurado y la relación entre los diferentes componentes está bien definida.
- **Seguridad**: la combinación de la estructura en capas y micronúcleo facilita la incorporación de mecanismos de seguridad. Las capas dos y tres se ejecutan en espacio de usuario, mientras que tan solo la capa uno se ejecuta en modo núcleo, que posee todos los privilegios necesarios para acceder a cualquier parte del sistema.
- **Extensible**: para poder tener un sistema funcional, es necesaria la configuración del núcleo, así como la de los servicios clave que son necesarios para comenzar. Todas las demás funciones pueden ser añadidas cuando sean necesarias. Esto hace más sencillo ampliar o especializar la función del sistema.
- **Rendimiento y estabilidad**: muchos problemas que provocan inestabilidad en un computador son resultado de controladores y programas mal diseñados. La arquitectura micronúcleo permite a estos programas ser ejecutados e implementados independientemente de los componentes principales del SO, lo que significa que un fallo en cualquiera de los controladores de dispositivo no es catastrófico para el sistema; puede mantenerse en ejecución, pese a los errores.

6.1.9 Desventajas de la arquitectura

Existen varios puntos que son considerados desfavorables para la estructura de la arquitectura, los cuales son descritos a continuación:

- **Complejidad**: la arquitectura de Minix tiene una estructura complicada, lo que dificulta, en primer lugar, su diseño y además, su evolución. A pesar de las posibles ventajas de un diseño modular a la hora de adaptar un software a un nuevo entorno, en Minix 3 la modularidad no es total, existiendo muchas dependencias entre sus distintas partes. La industria de la informática está entre las más cambiantes dentro de la economía mundial y por lo tanto sufre una necesidad real de adaptarse a los nuevos desarrollos tanto hardware como software, aspecto en el que Minix no se ha mostrado demasiado apropiado.
- **Comunicaciones y envío de mensajes**: este tipo de estructura necesita una arquitectura rápida y eficiente de comunicaciones para asegurar la máxima velocidad en la comunicación entre los distintos procesos que se ejecutan en su espacio individual de direcciones, así como con variados niveles de seguridad. Una mala implementación de las comunicaciones tendrá un gran impacto en el rendimiento del sistema (15).

6.1.10 ¿Dónde se puede obtener Minix 3?

El SO Minix 3 es distribuido en su página Web oficial[11] donde se publican todas las versiones de Minix incluyendo la que actualmente se encuentra en la fase de desarrollo. Para la fecha, la última versión estable es Minix 3.1.8 y la que se encuentra actualmente en la fase de desarrollo seria la Minix 3.1.9 (14).

Además, este SO como fue comentado en las secciones anteriores posee una publicación bibliográfica escrita por el creador de Minix, en este caso Andrew Tanenbaum con el nombre de “Operating Systems: Implementation and Design, The Minix 3rd Edition” y con la compra del mismo se entrega un CD-ROM con la versión Minix 3.1.0.

6.1.11 Requerimientos necesarios para la instalación de Minix 3.

Al igual que sus competidores, Minix 3 posee soporte para un tipo de hardware y entre ellos listamos los siguientes:

- **CPU**: Esta desarrollado para trabajar sobre la familia de procesadores x86 de 32 bits.
- **Memoria**: La instalación estándar o por defecto de Minix 3 requiere al menos 28 Mb (Megabyte) de RAM (Memoria de Acceso Aleatorio) y en instalaciones más avanzadas podría tan solo usar 8

[11] “Minix 3” - http://www.minix3.org/

Mb de RAM aunque para llevar a cabo el procedimiento de compilación del núcleo lo más recomendable es tener 64 MB.

- **Disco duro**: Actualmente Minix 3 da soporte a dispositivos IDE y serial-ATA. El SO requiere de una partición primaria y al menos 260 Mb libres en el disco duro, en caso de que se desee obtener el código fuente completo de Minix 3 se debe manejar un espacio de al menos 1 GB.
- **Tarjetas de red**: Minix 3 posee un escaso soporte con respecto a estos dispositivos.

7 Estudio del proceso de arranque

En este capítulo se muestra detalladamente el proceso relacionado al arranque del computador, incluyendo las partes relacionadas, dispositivos y sus especificaciones. Luego, se muestra la secuencia de programas involucrados para el proceso de arranque del SO Minix 3, entre ellos se explica el comportamiento del código *masterboot.s* y *bootblock.s*. También se expone los tipos de direccionamientos de discos duros y la estructura lógica de los mismos para que el SO puede arrancar.

Se mostrará paso por paso el proceso de arranque de un SO. Uno de los principales actores en éste proceso es la BIOS (Basic Input Output System) siendo un dispositivo indispensable, es por ello que se estudia las principales interrupciones utilizadas y las estructuras de datos que esta emplea.

7.1 *BIOS (Basic Input Output System)*

La BIOS es una colección de rutinas y datos que el fabricante del computador proporciona para manejar los dispositivos que componen al equipo, este código es almacenado en una memoria ROM (Read Only Memory). Anteriormente la BIOS era almacenada en dispositivos de memoria con tecnología ROM o EPROM (Erasable Programmable Read-Only Memory) pero actualmente se implementan sobre memorias de clase EEPROM (Electrically-Erasable Programmable Read-Only Memory) mejor conocida como memoria flash.

Normalmente una vez que la máquina es encendida las rutinas de la BIOS ocupan generalmente un espacio de 256 bytes en memoria RAM, donde contiene detalles sobre el estado de Bloq Num, el búfer de teclado, etc. La BIOS podemos dividirla en tres partes (16):

- Setup: es una utilidad del BIOS que puede utilizarse para modificar datos de configuración del sistema, tal como la cadena de discos de arranque.
- Rutinas de servicios: son un conjunto de llamadas que le permiten a las aplicaciones o al programador interactuar con los dispositivos del computador.
- Secuencia de arranque: es una secuencia donde se comprueban los componentes del sistema, inicializa las estructuras de datos para poder cargar un SO en el computador.

7.2 *Dispositivos de Almacenamiento*

Las unidades de almacenamiento de datos son dispositivos capaces de leer o escribir datos en medios o soportes de almacenamiento, son normalmente conocidos como la memoria secundaria de los computadores.

En su mayoría las unidades de almacenamiento más comunes son los discos magnéticos todos ellos son formateados o estructurados de forma similar y están divididos en áreas denominadas como (17) (18):

- Sectores: Las pistas se subdivide en varias secciones. Cada sección se llama sector. Los sectores son las unidades más pequeñas de almacenamiento en un disco duro. Cada sector contiene el mismo número de bits de datos (típicamente 512 bytes) codificados en el material magnético.
- Pistas: Cada superficie se compone de una colección de anillos concéntricos llamados pistas, que son delgadas tiras circulares de cinta magnética en la superficie del plato; las cuales contienen realmente los datos.
- Cilindros: es el conjunto de pistas de todas las superficies que son equidistantes del centro del eje.
- Cabecera: Los datos se escriben y se leen desde la superficie de un plato con un dispositivo llamado cabecera. Naturalmente, un disco tiene dos caras y por lo tanto dos superficies en las que los datos podrían ser manipulados, por lo general hay 2 cabezas por plato.

7.2.1 Unidad de disquete (Floppy).

Es un tipo de disco de almacenamiento magnético pequeño, flexible y barato (17). Existen disquetes de varias capacidades, el más común es el de 3,5 pulgadas, el cual permite almacenar hasta 1,44 MB. Es una unidad obsoleta, reemplazada por las unidades flash USB, discos duros externos, discos ópticos, tarjetas de memoria y redes informáticas. A pesar de ser una unidad obsoleta, abarca los conceptos relacionados a las unidades de almacenamiento más complejas, ayudando a entender los conceptos de forma sencilla. Es importante destacar que para acceder a esta unidad en código ensamblador el registro *dl* debe poseer el valor 0x00 o 0x01, el cual hace referencia a la primera o segunda unidad de disquete posible en una máquina. Para entender mejor la geometría de un disquete vea la Figura 7.1.

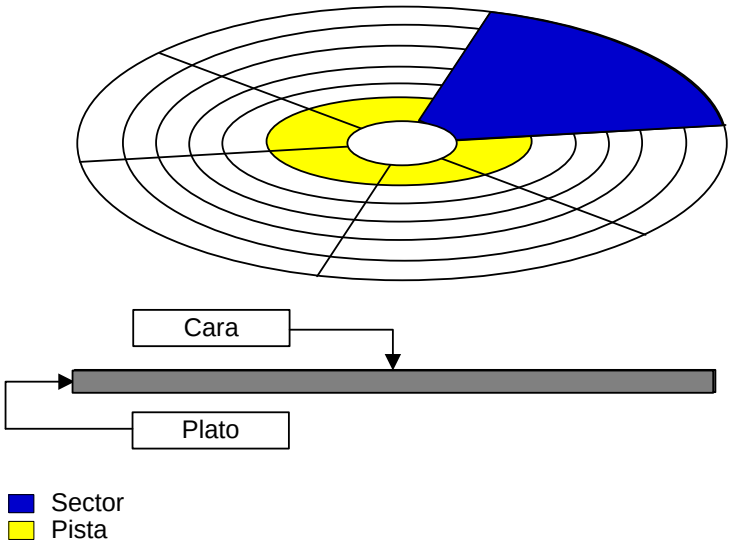


Figura 7.1 Geometría de un disquete

7.3 *Unidad de disco duro.*

Son caracterizados por poseer uno o más platos rígidos girando sobre un eje (18). Los discos duros tienen una gran capacidad de almacenamiento de información. El disco duro almacena casi toda la información que manejamos al trabajar con una computadora. En él se aloja, por ejemplo, el SO que permite arrancar la máquina, los programas, archivos de texto, imágenes, vídeos, etc. Dicha unidad puede ser interna (fija) o externa (portátil). Un disco duro está formado por varios discos apilados sobre los que se mueve una pequeña cabeza magnética que graba y lee la información. Es importante destacar que para acceder a esta unidad en código ensamblador el registro *dl* debe poseer el valor 0x80, 0x81, 0x82 o 0x83, el cual hace referencia las posibles unidades de disco duro en una máquina. Para entender mejor la geometría de un disco duro vea la Figura 7.2.

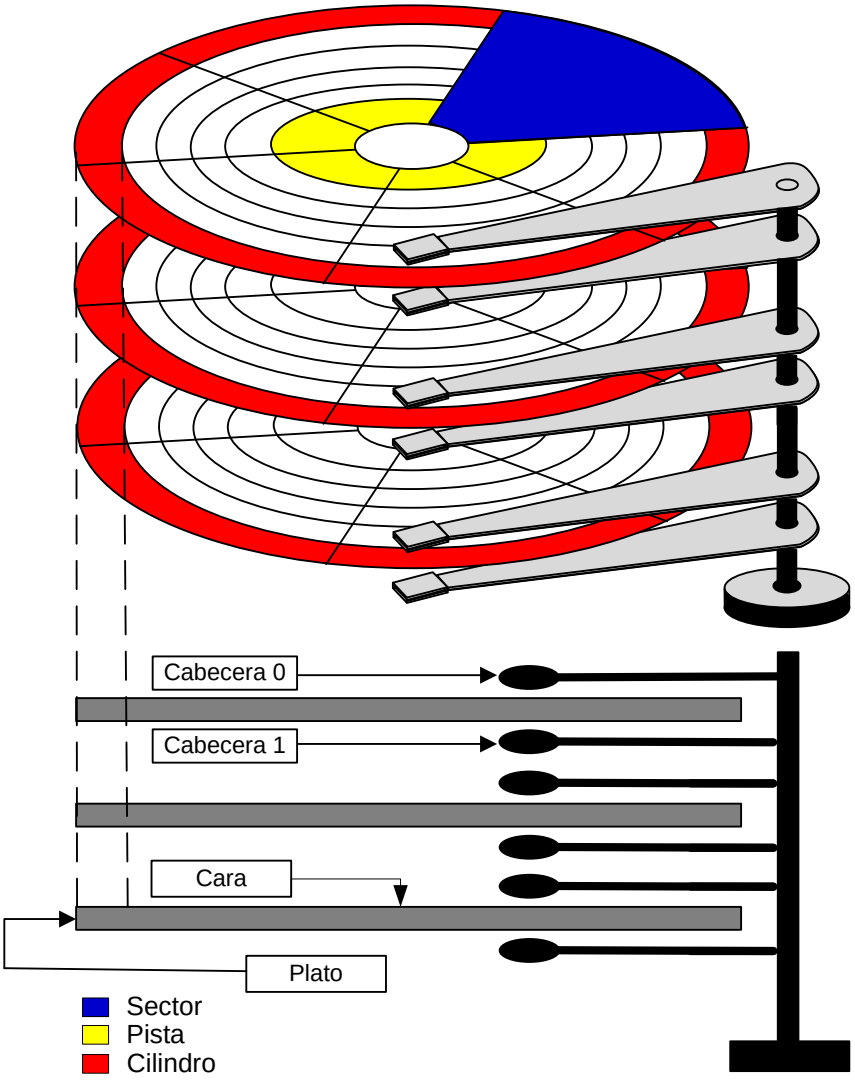


Figura 7.2 Geometría de un disco duro

7.4 *Modos de direccionamiento de sectores*

Existen dos métodos principales que permiten direccionar y acceder a los bloques físicos de un disco duro, ellos son CHS y LBA, los cuales serán explicados a continuación.

7.4.1 CHS (Cylinder Head Sector)

El modo CHS es el modo tradicional de acceso a los discos. Este método permite acceder a los bloques a través de una tripla que se define por el cilindro, cabeza y sector. Existen dos tipos de direccionamiento CHS físico y el direccionamiento CHS lógico. El **direccionamiento CHS físico** solo puede direccionar 504MB (16). Para entender esta premisa vea la Tabla 7.1.

	Bits	Valor máximo teórico = 2^n	Rango permitido	**Total utilizable**
Cilindro	10	1024	0-1023	**1.024**
Cabeza	4	16	0- 15	**16**
Sector	6	64	1-63	**63**

Tabla 7.1 Direccionamiento CHS físico

Estos números nos conducen a un total máximo de 1024 ***** 16 * 63 = 1.032.192 sectores, como en todos los discos duros cada sector es de 512 bytes, el resultado final es de 528.482.304bytes (528 MB). Este sería el máximo espacio de disco direccionable mediante los servicios de la interrupción 0x13 estándar BIOS, también conocida como int 13 (posteriormente se explicará dicha interrupción).

Este modo se amplió posteriormente para dar soporte hasta 8.064 MB exactamente con lo que comúnmente se conoce como el **direccionamiento CHS lógico.** Debido a que dichos valores son lógicos, los verdaderos valores correspondientes a la geometría real, son asunto exclusivamente del controlador de la unidad (16).La nomenclatura usada por este método se puede visualizar en Tabla 7.2.

	Bits	Valor máximo teórico = 2^n	Rango permitido	**Total utilizable**
Cilindro	10	1024	0-1023	**1.024**
Cabeza	8	256	0-255	**256**
Sector	6	64	1-63	**63**

Tabla 7.2 Direccionamiento CHS lógico

Los valores anteriores arrojan un total de 1024 * 256 * 63 = 16.515.072 sectores a direccionar. Los servicios de la BIOS podían direccionar un máximo de 1024 * 256 * 63 * 512 = 8.455.716.864Bytes, 8.455GB. Este es el límite teórico del direccionamiento CHS directo o de la interrupción 0x13 de la BIOS estándar.

Debido a esta limitante se buscó una solución que permitiera utilizar disco de mayor capacidad a la soportada de 8.45 GB que teóricamente podía proporcionar la BIOS estándar. Una de las soluciones provisionales fue la ECHS extended cylinder head sector, la ATA Specification (hasta 137 GB)[12], etc. Sin embargo, los alcances de las mismas no daban abasto al creciente aumento en la capacidad de los discos duros.

7.4.2 LBA (Logical Block Addressing)

Debido a que la capacidad de los discos fue creciendo con el tiempo, como se mencionó anteriormente, se hizo necesario sobrepasar también el límite de los 137 GB de la interrupción 0x13 de la BIOS que permite el método CHS. Para esto se ideó un sistema denominado LBA que diseña un sistema distinto para direccionar los sectores.

LBA, hoy en día es el esquema común utilizado para especificar la ubicación de los bloques de datos almacenados en la memoria secundaria, tales como los sistemas de discos duros. LBA fue desarrollada por primera vez para las unidades de disco duro SCSI.LBA es un esquema simple de direccionamiento lineal; los bloques son ubicados mediante un índice, el primer bloque es LBA=0, el segundo LBA=1, y así sucesivamente. El esquema de LBA sustituye a los regímenes anteriores, que expone los detalles físicos del dispositivo de almacenamiento para el software del SO[13].

El direccionamiento LBA en las unidades ATA puede ser de 28 bits o de 48 bits, lo que resulta en límites de 128 GB (2^28 sectores * 512 bytes por sector) y 128 PB (Petabyte) (2^48 * 512 bytes por sector). Desde luego, las BIOS que detectan el modo LBA también disponen de la traducción adecuada para solventar las limitaciones de la combinación BIOS/ATA (saltar la limitación de 8.455 GB). Debido a que la interrupción 0x13 no sabe nada sobre direccionamientos LBA, es la traducción la que resuelve dicho inconveniente.

Por supuesto todas las nuevas unidades de disco soportan LBA, y cuando esta circunstancia se presenta la BIOS la auto-detectada, estableciendo automáticamente el modo de direccionamiento y habilita la traducción correspondiente. Las direcciones de CHS se pueden convertir en direcciones LBA utilizando la siguiente fórmula[14]:

---

[12] "BIOS Disk Access" - http://oss.sgi.com/

[13] "48-bit LBA Technology" - http://www.48bitlba.com/

[14] "LBA and CHS format, LBA mapping" - http://www.*boot*-us.com

fórmula	LBA = ((C * Num_Head) + H) *Num_Sec)+ S - 1
leyenda	C = número de cilindros
	H = número de cabecera
	S = número de sector
	LBA = es la dirección lógica del bloque
	Num_Head = es el número de cabecera por cilindro
	Num_Sec = es el número de sectores por pista

Tabla 7.3 Fórmula de conversión de CHS a LBA

7.5 *Interrupción 0x13 de la BIOS*

La interrupción int 0x13 permite acceder directamente al disco duro utilizando cualquier lenguaje de programación de bajo nivel[15]. INT es una instrucción x86 que provoca una interrupción de software. La BIOS generalmente establece un manejador de interrupciones de modo real, este vector es el que proporciona la lectura y escritura de los sectores de disco duro o disquete utilizando la nomenclatura CHS. Para tener una visión clara, se explicará en qué consiste el modo real. El modo real de direcciones (a menudo llamado simplemente "modo real") es el modo que adopta el procesador inmediatamente después de la iniciación. En modo real la memoria es limitada a 1 MB, además, no ofrece soporte para la protección de memoria, multitarea, o los niveles de privilegio de código. A continuación será explicada la interrupción 0x13 con los parámetros más comunes utilizados para el proceso de arranque del SO.

7.5.1 INT 0x13, AH = 0x00

Reinicia el sistema de disco o disquete, ya que restablece el disco duro o el controlador de disco o disquete, obligando a la recalibración de la cabeza para la lectura/escritura (19). Para ver la especificación de esta interrupción ver la Tabla 7.4.

Entrada	AH = 0x00
	DL = dispositivo asociado
Salida	AH = estatus
	CF = encendida si existe un error, sino apagada

Tabla 7.4 INT 0x13, AH = 0x00

```
movb dl, #0x80      !dl = 0x80 hace referencia al primer disco duro
movb ah, #0x00      !ah = 0x00
nt 0x13             !realiza la llamada a la interrupción, para reiniciar el controlador del primer disco duro
```

Figura 7.3 Código de INT 0x13, AH = 0x00

[15] "Interrupts Page" - http://sps.nus.edu.sg

7.5.2 INT 0x13, AH = 0x02

Lee uno o más sectores de un disco duro o disquete para cargarlos en la memoria principal (19).Para ver las especificaciones de esta interrupción ver la Tabla 7.5.

<table>
<tr><td rowspan="6">Entrada</td><td>AH = 0x02</td></tr>
<tr><td>AL = número de sectores a leer (debe ser distinto de cero)</td></tr>
<tr><td>CH = tiene los ocho bits menos significativos del número de cilindro, recuerde que en CHS el número de cilindro se representa con 10 bits.
CL = número de sector en los bits 0-5 (total de 6 bits), los dos bits más significativos del cilindro en los bits 6-7 (solo aplica en discos duros). Ejemplo del registro cx:
<table>
<tr><td>CX =</td><td>CH</td><td>CL</td></tr>
<tr><td>Cilindro [0-9] bits</td><td>76543210</td><td>98</td></tr>
<tr><td>sector [0-5] bits</td><td></td><td>543210</td></tr>
</table></td></tr>
<tr><td>DH = número de la cabecera</td></tr>
<tr><td>DL = dispositivo asociado</td></tr>
<tr><td>ES:BX = buffer, posición de memoria donde son copiados los sectores</td></tr>
<tr><td rowspan="3">Salida</td><td>AH = estatus de la operación</td></tr>
<tr><td>AL = número de sectores leídos</td></tr>
<tr><td>CF = encendida si existe un error, sino apagada</td></tr>
</table>

Tabla 7.5 INT 0x13, AH = 0x02

```
mov ax, #0x0201      !Código para leer, únicamente un sector; ah = 0x02 y al = 0x01
!el cilindro = 263 y el sector = 5
movb ch, #0x07       !ch[0..7] = bits menos significativos del cilindro
movb cl, #0x85       !cl[0..5] = sector, cl[6..7] = dos bits más significativos del cilindro
movb dh, #0x0A       !dh = 0x0A, indica que va a leer de la cabecera 10
movb dl, #0x80       !dl = 0x80 hace referencia al primer disco duro
movb es, #0x06
movb bx, #0x00       !buffer = 0x0600, posición de memoria donde son copiados los sectores

int 0x13             !realiza la llamada a la interrupción
```

Figura 7.4 Código de INT 0x13, AH = 0x02

7.5.3 INT 0x13, AH = 0x08

Obtiene los parámetros del dispositivo. Ofrece la información de los parámetros de la unidad de disco, tales como el número de cabezas, pistas y sectores por pista (19), vea la Tabla 7.6.

Entrada		AH = 0x08
		DL = dispositivo asociado
Salida	Ambos para ambos	AH = estatus
		CF = encendida si existe un error, sino apagada
		CH = tiene los ocho bits más bajo del número de cilindro
		CL = número de sector en los bits 0-5 (total de 6bits), los dos bits más altos del cilindro en los bits 6-7 (solo aplica en discos duros).
		DH = número de la cabecera
	disco	DL = número del disco asociado
	Solo aplica para disquete	BL = tipo de dispositivo
		DL = número del disquete asociado
		ES: DI = Apuntador a la tabla de parámetros de la unidad de disquete

Tabla 7.6 INT 0x13, AH = 0x08

```
movb dl, #0x80      !dl = 0x80 hace referencia al primer disco duro
ovb ah, #0x08       !ah = 0x08, indica la llamada
int 0x13            !realiza la llamada a la interrupción
```

Figura 7.5 Código de INT 0x13, AH = 0x08

7.5.4 INT 0x13, AH = 0x42

Lectura extendida de sectores del disco. Permite leer los sectores que se encuentran sobre los 8.45 GB que permite el direccionamiento lógico de CHS (19). Para ver las especificación de esta interrupción ver la Tabla 7.7.

Entrada	AH = 0x00
	DL = dispositivo asociado
	DS:SI = dirección que apunta al paquete de dirección de disco
Salida	CF = encendida si existe un error, sino apagada
	AH = código de error

Tabla 7.7 INT 0x13, AH = 0x42

```
movb dl, #0x80        !dl = 0x80 hace referencia al primer disco duro
movb ah, #0x42        !ah = 0x42, indica la llamada
movb ds, #0x08
movb si, #0x00        !buffer = 0x0600, apunta a la dirección del paquete de dirección de disco

int 0x13              !realiza la llamada a la interrupción
```

Figura 7.6 Código de INT 0x13, AH = 0x42

7.6 *Secuencia de arranque*

¿Cómo inicia un SO? Se utilizará un primer enfoque resumido para poder entender como inicia un SO y luego se explicará de forma detallada el proceso. En la mayoría de las computadoras modernas existen varios dispositivos a partir de los cuales se puede iniciar el proceso de arranque del SO, es por ello que existe una jerarquía de arranque. Típicamente, se intenta arrancar desde la unidad de disquete, si este intento no es exitoso, se intenta arrancar desde la unidad de CD-ROM. Si desde la unidad de CD-ROM es fallido, se intenta arrancar desde la unidad de disco duro. El orden de esta jerarquía puede ser configurable a través del *setup* de la BIOS, como se mencionó anteriormente. Adicionalmente otros dispositivos también brindan soporte a este proceso de inicio, especialmente los de almacenamiento removible (17).

Suponga que la computadora es encendida, si el dispositivo de arranque es un disquete, el hardware lee el primer sector de la primera pista del disco, lo carga en memoria y ejecuta el código encontrado allí. En el disquete, este primer sector contiene el programa *bootstrap*. Este programa es muy pequeño, ya que tiene que entrar en un sector (512 bytes). El *bootstrap* carga un programa más grande, llamado *boot*, posteriormente éste último es el encargado de cargar el SO. Para entender mejor este escenario vea la Figura 7.7 donde aparece el diseño mencionado donde el primer sector contiene el *bootblock* y el disco no está particionado.

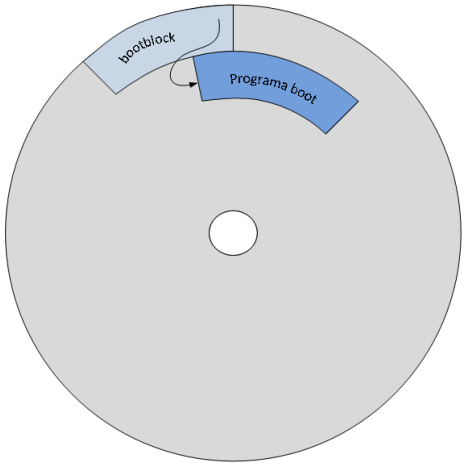


Figura 7.7 Estructura de un disquete.

En contraste, el disco duro requiere un paso intermedio. Un disco duro está dividido en particiones, y el primer sector de un disco duro contiene un programa pequeño y la tabla de particiones del disco. Colectivamente estas dos piezas son llamadas MBR (Master Boot Record). El hardware lee el primer sector de la primera pista del primer cilindro del disco duro y lo carga en memoria. Este programa es ejecutado para leer la tabla de particiones y seleccionar la partición activa (17). La partición activa tiene un programa *bootstrap* en su primer sector, el cual es cargado y ejecutado para encontrar e iniciar una copia del *boot* en la partición, exactamente como se hace cuando se arranca desde un disquete. Para entender mejor este escenario ver la Figura 7.1.

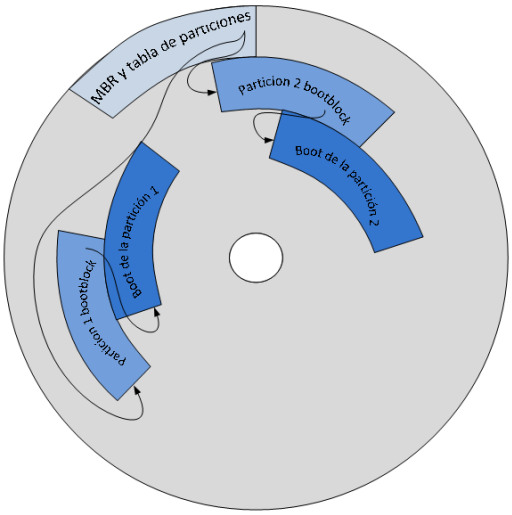


Figura 7.8 Estructura de un disco duro

En cualquier caso, una vez cargado el *boot* de Minix 3, éste busca un archivo multipartes ya sea en el disco (disquete) o partición (disco duro) y carga las partes individuales en las posiciones apropiadas de la memoria. Esta es la *boot image*, para visualizarla mejor ver la Figura 7.9. La parte más importante son el núcleo (el cual incluye el reloj y la tarea del sistema), el manejador de proceso y el sistema de archivo. Adicionalmente algunos driver deberían ser cargados en la *boot image*. Esto incluye el servidor reencarnación, el disco RAM, la consola e *init*.

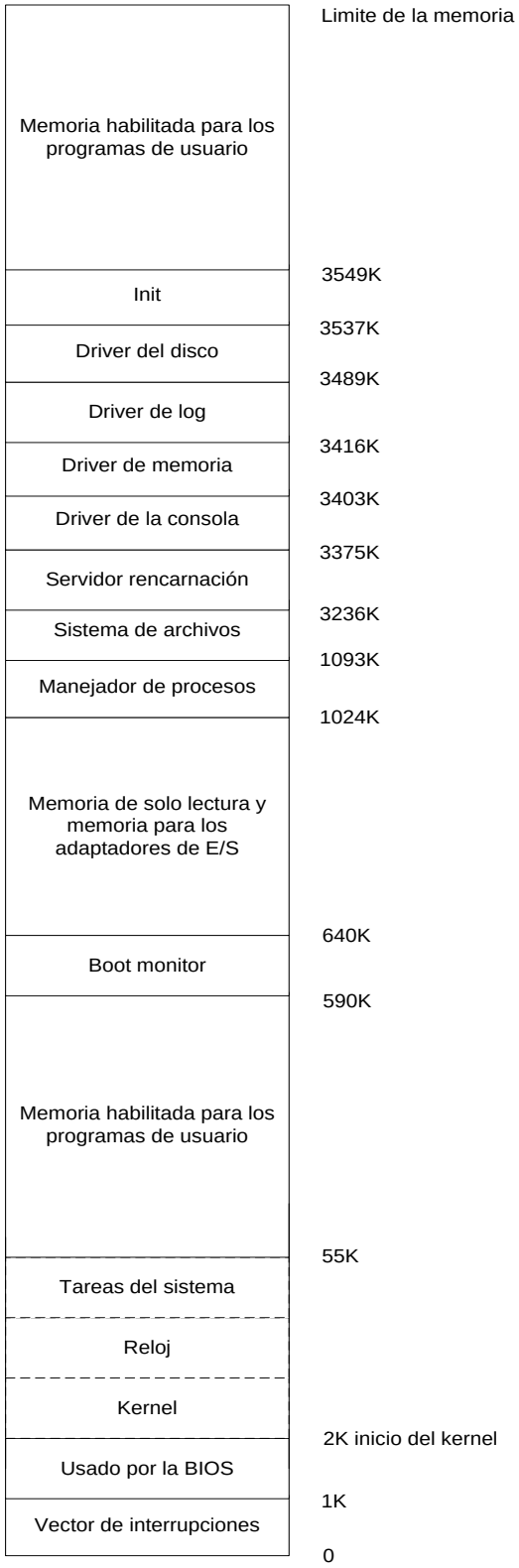


Figura 7.9 Diseño de la memoria RAM luego de que Minix ha sido cargado desde el disco

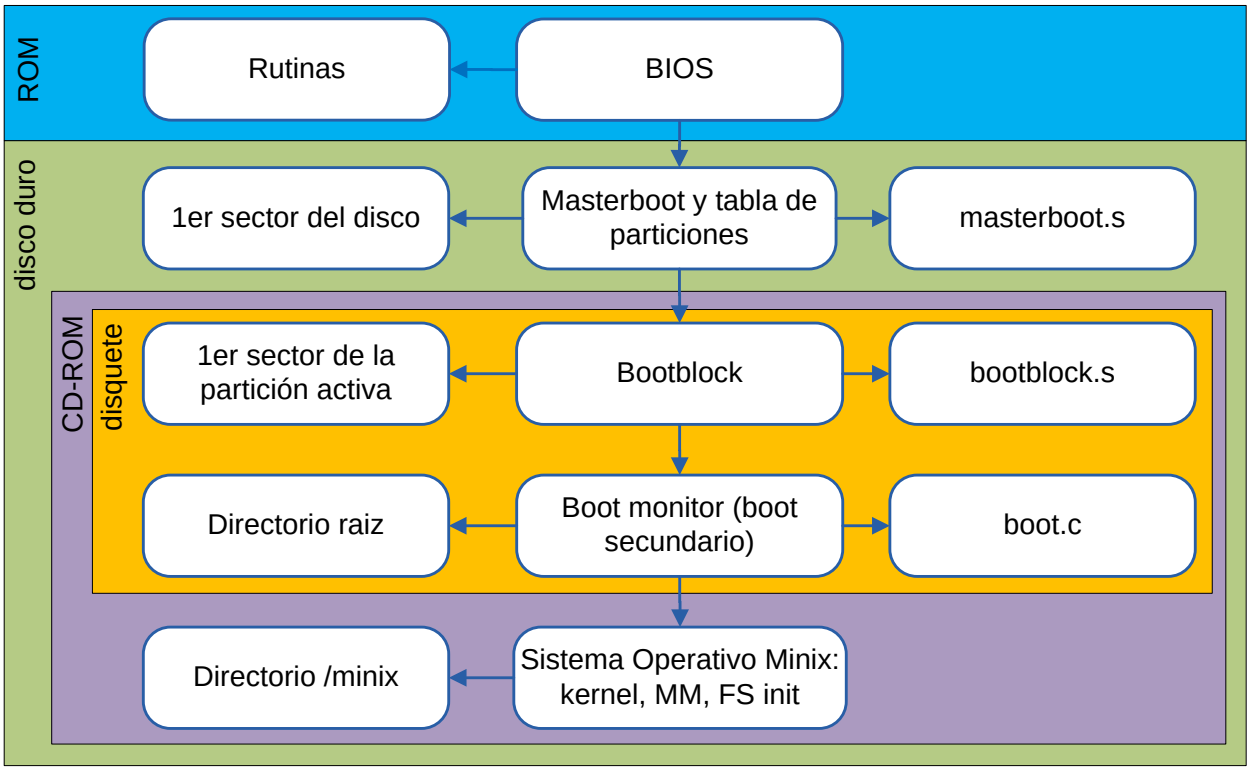


Figura 7.10 Proceso de arranque de Minix 3

La secuencia de este proceso de arranque puede ser fácilmente entendido viendo la Figura 7.10 donde además puede apreciarse donde son almacenados los ejecutables dependiendo del dispositivo de almacenamiento. Como se aprecia se empieza por la ROM que solo almacena a la BIOS, sus estructuras de datos y rutinas. Por otra parte, como puede observarse un disquete solo puede almacenar a *bootblock* y *boot*. El CD-ROM puede almacenar al *bootblock*, *boot* y SO. Por último el un disco duro puede almacenar a todos los programas mencionados anteriormente.

Ahora se explicará el proceso con una visión más detallada del mismo. Cuando el sistema esta inicializado, el hardware (realmente, un programa en la ROM) lee el primer sector del disco de arranque, lo copia para cargarlo en memoria y ejecuta el código encontrado allí. En un disco no particionado, como un disquete, en el primer sector está el *bootblock* el cual carga el programa *boot*, como lo muestra la Figura 7.7. El disco duro siempre está particionado, y el programa ubicado en el primer sector (llamado *masterboot* en el sistema Minix) primero se traslada asimismo a otra región de memoria, luego lee la tabla de particiones, recuerde que viene cargada con él desde el primer sector. Posteriormente, carga y ejecuta el primer sector de la partición activa, como se muestra en la Figura 7.8. Normalmente una y solo una partición está marcada como activa. Una partición de Minix 3 tiene la misma estructura de un disquete, con un código *bootblock* que carga el programa *boot*. El código *bootblock* es el mismo para un disco particionado o un disco sin partición, para entender mejor como un disco duro esta particionado en

Minix vea la Figura 7.11. Cuando el programa *masterboot* se traslada asimismo el *bootblock* puede ser escrito en memoria y ejecutado en la misma dirección de memoria donde originalmente el *masterboot* fue cargado, más adelante será explicado con mayor detalle (15).

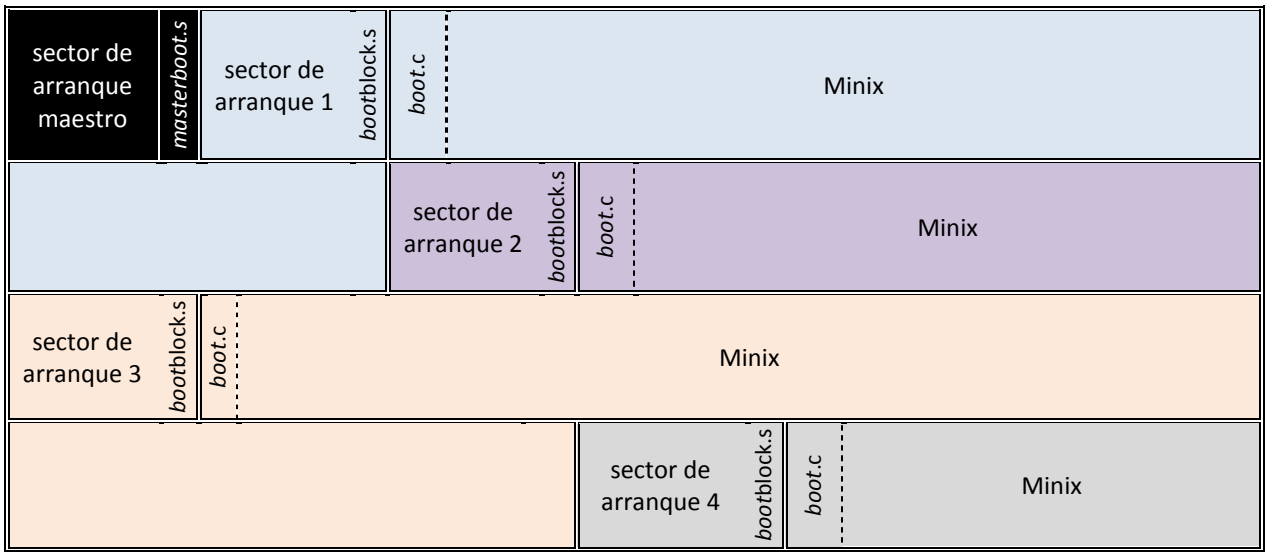


Figura 7.11 Estructura de un disco particionado

La situación real puede ser un poco más complicada de lo que muestra la Figura 7.11, en la cual se puede visualizar que en el primer sector físico del disco duro se encuentra el MBR, y el disco esta particionado alojando a cuatro sistemas operativos (20). Cada partición como tiene su *bootstrap* o cargador primario llamado *bootblock.s* en el caso de Minix. También, tiene su cargador secundario o *boot* monitor que puede estar disperso en varios bloques llamados *boot*.

Se dice que puede ser más complicada la situación debido a que una partición puede contener subparticiónes. En tal caso el primer sector de la partición será otro *masterboot record* conteniendo la tabla de particiones para las subparticiónes. No obstante, tarde o temprano se transferirá el control a un sector de arranque, el primer sector de un dispositivo que no tiene más subdivisiones. En un disquete el primer sector siempre es el sector de arranque. En un disquete el primer sector siempre es un sector de arranque. Minix permite una forma de subdivisión de un disquete, pero no se brindará esa información aquí.

El sector de arranque de Minix se modifica en el momento en que se escribe en el disco a través de un comando, el programa alojado en este sector (*bootblock*) es el encargado de encontrar un programa llamado *boot* en su partición o subpartición. Una vez encontrado debe cargar en memoria los sectores que comprenden a éste (*boot*) y darle el control, para que sea luego el *boot* pueda cargar al SO finalmente. Esta introducción es necesaria porque antes de que se cargue el SO no es posible usar el directorio y nombres de archivo para encontrar un archivo. Se utiliza un programa especial llamado

installboot para realizar la introducción y la escritura del sector de arranque. *Boot* es el cargador secundario para Minix, pero puede hacer más que simplemente cargar el SO, ya que es un programa monitor que permite al usuario modificar, establecer y guardar diversos parámetros.

Boot examina el segundo sector de su partición en busca de un conjunto de parámetros que usará. Minix, al igual que el UNIX estándar, reserva el primer bloque de 1 KB de todos los dispositivos de disco como bloque de arranque, pero el cargador de arranque en ROM o el sector maestro de arranque sólo carga un sector de 512 bytes, así que hay 512 bytes disponibles para guardar ajustes. Éstos controlan la operación de arranque, y también se pasan al SO mismo. Los ajustes por omisión presentan un menú con una sola opción, iniciar Minix, pero es posible modificarlos de modo que presenten un menú más complejo que permita la iniciación de otros sistemas operativos (cargando y ejecutando sectores de arranque de otras particiones), o iniciar Minix con diversas opciones. También podemos modificar los ajustes por omisión de modo que pasen por alto el menú e inicien Minix de inmediato.

Boot no es parte del SO, pero es lo suficientemente inteligente para usar la estructura de datos del sistema de archivo y encontrar la imagen real de SO. La imagen de Minix cargada por *boot* no es más que una concatenación de los archivos individuales producidos por el compilador cuando compila los programas del núcleo, el administrador de memoria y el sistema de archivos. A partir de la información contenida en la cabecera de cada parte, *boot* determina cuánto espacio debe reservar para datos no inicializados después de cargar el código ejecutable y los datos inicializados de cada parte, con objeto de que la siguiente parte pueda cargarse en la dirección correcta.

Para saber de forma detallada como se inicia un SO, se presenta a continuación los pasos que sigue este proceso. Cada fabricante de placa base o tarjetas madres disponen de su propia BIOS, a pesar de posibles diferencias entre ellas, existe un procedimientos o patrón de arranque común. La secuencia de arranque es la siguiente (20):

- La fuente de poder lleva a cabo una auto-prueba. Cuando los niveles de voltaje son aceptables, la fuente de poder envía la señal *Power Good* al procesador, hasta que el mismo comience a iniciar. El tiempo provisto para este paso es de 0,1 y 0,5 segundos.
- El procesador Intel ha sido diseñado para que siempre al iniciar ejecute el código que se encuentre en la posición de memoria 0xFFFF0. Sin embargo, antes de eso debe cagar en memoria las instrucciones que va a ejecutar. Es por ello que el procesador carga en la posición 0xFFFF0 el programa de arranque de la BIOS, que son 16 bytes y está en el tope de la memoria ROM o EPROM de la misma, específicamente en la posición de memoria 0xF000FFF0.
- Cualquier error que se producen en este punto del proceso de arranque se informará por medio de "códigos bip" porque el subsistema de vídeo aún no se ha inicializado.

- Se busca el adaptador de video, razón por la cual esta información se muestra por pantalla antes que la información de arranque. La rutina de inicio de la BIOS escanea las direcciones 0xC0000000 hasta la 0xC7800000 para encontrar la ROM de video.
- Para determinar si se trata de un inicio en caliente (cuando se reinicia la computadora) o en frío (cuando se enciende por primera vez) la rutina de inicio de la BIOS comprueba si el valor de dos bytes situados en posición de memoria 0x00000472. Cualquier valor distinto de 0x1234 indica que se trata de un arranque en frío.
- La BIOS busca y ejecuta otras ROM BIOS en los adaptadores de dispositivo. Normalmente los discos duros IDE/ATA tienen su BIOS en la dirección 0xC800.
- Se establece el ambiente Post-PC realizando un conjunto de pruebas POST (Power On Self Test). El POST se puede dividir en tres componentes:
    - La BIOS realiza un inventario de los dispositivos existentes, entre ellos el teclado y los puertos seriales y paralelo.
    - Prueba de memoria, prueba los chips de memoria y muestra una suma continua de memoria instalada.
    - La identificación del BIOS, muestra la versión del BIOS, el fabricante y la fecha.
- La BIOS muestra en forma de tabla un resumen de la configuración del sistema. Esta tabla indica los problemas surgidos en el arranque, si los hubiese. Los errores que se producen durante el POST se pueden clasificar como grave o no graves. En un error no grave generalmente se muestra un mensaje de error en la pantalla y permite que el sistema siga el proceso de arranque. En un error grave, en cambio, detiene el proceso de arranque del computador y es generalmente denotado por una serie de códigos de pitido.
- Si la BIOS soporta el estándar "Plug and Play (PnP)", detecta y asigna recursos a los dispositivos PnP. Muestra un mensaje por cada uno que encuentra.
- La BIOS comienza a buscar un dispositivo desde el que arrancar el sistema. Normalmente comenzando por los disquetes, los discos duros y los CD, dependiendo de cómo se ha parametrizado la cadena de arranque en el programa de configuración de la BIOS. Entonces carga en memoria el primer sector del dispositivo en la dirección 0x00007C000. A continuación, la BIOS comprueba que los últimos dos octetos del sector son 0xAA55. Si no lo son, significa que el primer sector del disquete o disco duro no es un sector de arranque.
- En un disco duro el MBR ocupa el sector de la primera en el cilindro 0, cabeza 0, sector 1. Es 512 bytes de tamaño. Si este sector se encuentra, se carga en memoria en la dirección 0x00007C00 y se prueba para identificar si posee la firma válida. Una firma válida sería el valor 0x55AA en los últimos dos bytes del sector. Al carecer de un MBR o una firma válida el proceso de arranque se detiene con un mensaje de error.

Luego de estos pasos el proceso de arranque de Minix 3 es el mismo descrito en la Figura 7.10. En la siguiente sección se muestra en detalle los programas involucrados en el proceso de arranque de Minix

7.7 *Masterboot*

Como se mencionó anteriormente el MBR (Master Boot Record) está compuesto por un programa pequeño contenido en el primer sector de un disco duro y la tabla de particiones. Este programa es el encargado de cargar la partición activa. Está estructurado de la siguiente forma los primeros 446 bytes están destinados al programa, y si es el caso un poco de relleno; a partir de allí se encuentra a la tabla de particiones la cual ocupa 64 bytes, conteniendo 4 registros de 16 bytes, los cuales definen entradas a la tabla de particiones. En ellos se almacena toda la información básica sobre la partición. Para finalizar, los 2 últimos bytes representan a la firma o número mágico, el cual indique si el dispositivo es de arranque. Para entender mejor la estructura ver la Figura 7.12 y la Figura 7.13.

<table>
<tr><td rowspan="6">512 bytes</td><td>446 bytes</td><td colspan="2">Código del gestor de arranque</td></tr>
<tr><td rowspan="4">64 bytes</td><td>16 bytes</td><td>Primera partición</td></tr>
<tr><td>16 bytes</td><td>Segunda partición</td></tr>
<tr><td>16 bytes</td><td>Tercera partición</td></tr>
<tr><td>16 bytes</td><td>Cuarta partición</td></tr>
<tr><td>2 bytes</td><td colspan="2">Firma de unidad arrancable = 0x55AA</td></tr>
</table>

Figura 7.12 Primer sector físico del disco duro

<table>
<tr><td rowspan="6">16 bytes</td><td>1 byte</td><td>Define si la partición está activa, verificando que el bit 7 tenga el valor 1. En otras palabras (0x00 = Inactiva; 80h = Activa)</td></tr>
<tr><td>3 bytes</td><td>CHS de inicio</td></tr>
<tr><td>1 byte</td><td>Tipo de partición</td></tr>
<tr><td>3 bytes</td><td>CHS final</td></tr>
<tr><td>4 bytes</td><td>LBA = en formato little-endian, indica el número del sector de arranque (contando desde 0)</td></tr>
<tr><td>4 bytes</td><td>Tamaño en sectores = en formato little-endian, indica el tamaño de la partición representado en sectores</td></tr>
</table>

Figura 7.13 Diseño de una entrada de la tabla de partición

Se muestra la forma en que se guarda las triplas de CHS de inicio y de CHS final, el primer bytes de los tres contiene el número de la cabecera. Los dos siguientes se muestran en la Figura 7.14.

Estructura de la tripla CHS								
Primer byte	7	6	5	4	3	2	1	0
	bits del 7-0 de cabecera							
Segundo byte	8	9	5	4	3	2	1	0
	bits 9-8 cilindro		bits 5-0 del sector					
Tercer byte	7	6	5	4	3	2	1	0
	bits 7-0 del cilindro							

Figura 7.14 Estructura de la tripla CHS

El tipo de partición define el formato que tiene una partición para que el SO pueda gestionar los datos contenidos en su interior. Una vez comprendida la secuencia de arranque y los principales programas de Minix involucrados en este proceso se presentará ahora el código de los mismos, empezando en esta sección con el *masterboot.s*. Primero se mostrará los pasos que sigue el *masterboot.s* y luego el código fuente del mismo.

- Una vez cargado el primer sector en la posición 0x7C00 (recuerde que la BIOS es quien lo carga), el *masterboot.s* se copia asimismo a la posición desde la posición 0x7C00 a la 0x0600 y saltar a esta última posición. Esto es realizado por seguridad, ya que el próximo código (*bootblock.s*) será cargado en la posición 0x7C00 nuevamente por la BIO0053.
- Busca el dispositivo donde está alojado el *bootstrap*, para esto verifica el dispositivo en el registro dl. De ser un disquete, carga el primer sector del mismo y le sede el control. Si en un disco duro, busca en la tabla de particiones la partición activa, y almacena la dirección en los registros de donde comienza dicha partición en el disco duro.
    - Se leen las características de la unidad elegida (cabeza, cilindro y sector si se trata de un disco duro).
    - Se carga el primer sector físico de la unidad.
        - Si existe error, se reintenta la lectura del primer sector tres veces. Si falla, se muestra un mensaje en pantalla: “Error de lectura”; quedándose en un bucle infinito.
        - Si la lectura tiene éxito, el siguiente paso consiste en comprobar si es de arranque:
            - Si no está la firma de arranque, se muestra un mensaje en pantalla: “Unidad no arrancable” y se queda en un bucle infinito.
            - Si está la firma, se continúa con el programa.
- Se salta al código cargado (*bootstrap*).

La primera instrucción de este código fuente es cargada en memoria principal con un desplazamiento de

0x7C00.La variable LOADOFF=0x7C00 indica la primera posición de memoria donde será cargado el programa *masterboot*. La variable BUFFER=0x0600 indica la siguiente posición donde el programa *masterboot* será copiado debido a que debe darle espacio en memoria para cargar el *bootblock*.

```
LOADOFF=0x7C00      !En el espacio de memoria 0x0000:0x7C00 el código será cargado.
BUFFER=0x0600       !Inicia el primer espacio de memoria libre.
PART_TABLE=446      !Ubicación de la tabla de particiones. (Revisar estructura del Master Boot Record).
PENTRYSIZE=16       !Cada entrada en la tabla de particiones es de 16Bytes de tamaño.
MAGIC=510           !Ubicación de la firma 0xAA55 identifica si un dispositivo es arrancable o no.
```

Para identificar a los discos duros se utiliza la siguiente notación: hd0 identifica al primer dispositivo y hd1 identifica a la primera partición primaria del primer dispositivo. Como solo se pueden manejar 4 particiones primarias entonces podemos decir que hd[1-4] identifican a las particiones primarias del primer dispositivo. hd5 identifica al segundo dispositivo. Si las particiones de un dispositivo han sido subparticionadas entonces hd1a sería la primera subpartición de la primera partición del primer disco.

```
bootind=0           !Boot indicator.
sysind=4            !System indicator. Indica el tipo de partición si es primaria o lógica.
lowsec=8            !Logical first sector. Dirección lógica del primer sector, es decir, LBA.
.text                       !Inicia el conjunto de instrucciones relacionadas a la ejecución del código fuente.
master:             !Creación del entorno de trabajo.
xor ax, ax          !Inicializa el registro ax=0.
mov ds, ax          !Asigna al registro ds el valor de ax=0.
mov es, ax           !Asigna al registro es el valor de ax=0.
cli                 !Esta instrucción me permite deshabilitar las interrupciones enel procesador.
mov ss, ax          !Asigna al registro ss el valor de ax=0.
mov sp, #LOADOFF    !Desplazamiento de la pila.
sti                 !Esta instrucción me permite habilitar las interrupciones en elprocesador.
mov si, sp          !Asigna al registro si el valor de sp = #LOADOFF = 0x7C00. Se llama a la instrucción ret después de bootstrap
push si             !Se apila el valor contenido en el registro si. Para almacenar la información de retorno.
mov di, #BUFFER     !Se asigna al registro di = #BUFFER = 0x0600.
mov cx, #512/2      !Se asigna al registro cx = 256.
cld                 !Limpia las banderas.
rep movs   !Esta instrucción se va encargar de copiar tantas palabras de2Bytes como lo indique el registro cx iniciando !desde la dirección
ds=0x0000:si=0x7C00 hacia es=0x0000:di=0x0600.
jmpf BUFFER+migrate, 0      !Esta instrucción es para al ya ser copiado esta sección de código a su nueva locación en memoria
!principal, este retoma su ejecución en la etiqueta migrate
```

```
migrate:
findactive:
testb dl, dl                !Activa la bandera si el parámetro es negativo.
jns nextdisk                !Verifica el estado de la bandera sign o signo, y realiza el salto si esta se encuentra activada.
mov si, #BUFFER+PART_TABLE  !Le asigna a si la ubicación de la tabla de particiones.
find:
cmpb sysind(si), #0         !Se verifica el tipo de partición es igual a cero, y si lo es entonces significa que la partición esta vacía.
jz nextpart                 !En caso de que la partición este vacía, este salta a la etiqueta nextpart para verificar la siguiente partición.
testb bootind(si), #0x80    !Al verificar que la partición no está vacía, ahora verifica si la partición se encuentra activa.
jz nextpart                 !En caso de que la partición no este activa, este salta a la etiqueta nextpart para verificar la siguiente partición.

loadpart:
call load                   !Se realiza la llamada a load para cargar el bootstrap (sea el bootblock o un nuevo masterboot).
jc error1                   !En caso de errores.

bootstrap:
ret                         !Cuando retorna de la llamada load, le cede el control al bootstrap.
nextpart:
add si, #PENTRYSIZE         !Se le añaden 16bytes a si para acceder ala siguiente entrada de la tabla de particiones.
cmp si, #BUFFER+PART_TABLE+4*PENTRYSIZE   !Verifica si ya se revisaron todas las entradas 4 de la tabla de particiones.
jb find                     !En caso de existir una entrada valida, salta a find.
call print                  !Se llama a print para comentar que no existe ninguna partición.
.ascii "No active partition\0"
jmp reboot
nextdisk:                   !No existen particiones activas en esta unidad arrancable entonces intenta con la siguiente
incb dl                     !Incrementa el valor del registro dl que contiene la siguiente unidad arrancable.
testb dl, dl                !Si el parámetro dl es negativo entonces se activa la bandera de signo.
js nexthd                   !Si dl es negativo entonces se realiza un salto a la etiqueta nexthd.
int 0x11                    !Lee la configuración de los dispositivos y el bit 6-7 tiene el número de unidades arrancables.
shl ax, #1                  !Realiza un desplazamiento a la izquierda.
shl ax, #1                  !Realiza un desplazamiento a la izquierda.
andb ah, #0x03              !Se aplica una máscara al registro ah para extraer los bits 6-7.
cmpb dl, ah                 !Si el valor de dl es menor o igual a los obtenido en ah por la interrupción 0x11 entonces la unidad existe.
ja nextdisk                 !En otro caso intenta con hd0
call load0                  !Si falló, próximo disco por favor
jc nextdisk                 !It failed, next disk please
ret                         !Jump to the next master bootstrap
nexthd:
call load0                  !Lee el bootstrap alojado en el disco duro.
error1:
jc error                    !En caso de error.¿No existe el disco?
ret
```

```
load0:          !Carga el sector 0 del dispositivo actual, sea un bootstrap de disquete o un master bootstrap de disco duro.
mov si, #BUFFER+zero-lowsec     !si = donde lowsec(si) es cero
load:           !Cargar el sector lowsec(si) desde el dispositivo actual. Los número de cabeza, sector y cilindro son ignorados
                !para favorecer de manera absoluta el comienzo de la partición.

mov di, #3      !Tres reintentos para comprobar si hay disquete
retry:
push dx         !Grabar código de la unidad
push es
push di         !La próxima llamada destruye es y di
movb ah, #0x08  !Código para los parámetros de la unidad
int 0x13
pop di
pop es
andb cl, #0x3F  !cl = max sector number (1-origin) se lee los últimos 6 bits que tiene el número de sectores incb dh
                !dh = 1 + max head number (0-origin) índice de la cabecera de la unidad, se le suma 1 porque empieza en 0
movb al, cl     !al = cl = sectors per track, asigna el número de sectores
mulb dh         !dh = cabeceras, ax = cabeceras* sectores
mov bx, ax      !bx = sectores por cilindros = cabeceras * sectores
mov ax, lowsec+0(si)  !ax = le asigna el LBA los dos primeros bytes
mov dx, lowsec+2(si)  !dx:ax = sector within drive, dx = le asigna el LBA los dos últimos bytes
cmp dx, #[1024*255*63-255]>>16          !Near 8G limit?
jae bigdisk
div bx          !ax = cilindro, dx = sector sin cilindro, ax = tiene el cilindro de la partición, dx = sector
xchg ax, dx     !ax = sector sin cilindro, dx = cilindro
movb ch, dl     !ch = ch = 8 bits inferiores de cilindro
divb cl         !al = cabeza, ah = sector (0-origen)
xorb dl, dl     !Desplazamiento de los bits 8-9 del cilindro en dl
shr dx, #1
shr dx, #1      !dl[6..7] = cilindro superior
orb dl, ah      !dl[0..5] = sector (0-origen)
movb cl, dl     !cl[0..5] = sector, cl[6..7] = cilindro superior
incb cl         !cl[0..5] = sector (1-origen)
pop dx          !Restablecer código de la unidad en dl
movb dh, al     !dh = al = cabecera
mov bx, #LOADOFF    !es:bx = donde se carga el sector
mov ax, #0x0201     !Código para leer, unicamente un sector
int 0x13            !Llamar a la BIOS para una lectura
 jmp rdeval                 !Evaluar los resultados de la lectura
bigdisk:
mov bx, dx      !bx:ax = dx:ax = Número de sectores a leer
pop dx          !Restaura el valor de la unidad en dl
```

```
push si                 !Guarda si
mov si, #BUFFER+ext_rw  !si = extendida lectura/escritura usando como parámetro el paquete
mov 8(si), ax           !Número de inicio del bloque = bx:ax
mov 10(si), bx
movb ah, #0x42          !Llamada a la lectura extendida
int 0x13
pop si                  !Restaura si para apuntar a la tabla de partición
!salta a rdeval
rdeval:
jnc rdok                !Si la lectura fue exitosa
cmpb ah, #0x80          !Tiempo de espera agotado? (dispositivo disquete vacío)
je rdbad
dec di
jl rdbad                !Número de reintentos vencidos
xorb ah, ah
int 0x13                !Reinicia
jnc retry               !Intentar de nuevo
rdbad:
stc                     !Establecer la bandera carry
ret
rdok:
cmp LOADOFF+MAGIC, #0xAA55
jne nosig               !Error si la firma es incorrecta
ret                     !Retorna con la bandera carry limpia
nosig:
call print
.ascii "Not bootable\0"
jmp reboot
!Un error de lectura se produjo
error:
mov si, #LOADOFF+errno+1
prnum:
movb al, ah             !Número de error en ah
andb al, #0x0F          !Inferiores 4 bits
cmpb al, #10 !A-F?
jb digit !0-9!
addb al, #7 !'A' - ':'
digit:
addb (si), al           !Modificar '0' en el string
dec si
movb cl, #4             !Próximos 4 bits
shrb ah, cl
```

```
. jnz prnum                    !De Nuevo si el digito es> 0
call print
.ascii "Read error "
errno:
.ascii "00\0"
!jmp reboot
reboot:
call print
.ascii ". Hit any key to reboot.\0"
xorb ah, ah          !Esperar a que se oprima una tecla
int 0x16
call print
.ascii "\r\n\0"
int 0x19
!Print a message.
print:
pop si               !si = siguiente String 'call print'
prnext:
lodsb                !al = *si++ es el carácter a imprimir
testb al, al         !La marca de nulo señala el fin
jz prdone
movb ah, #0x0E       !Impresión de caracteres en el modo de teletipo
mov bx, #0x0001      !Página 0, color de primer plano
int 0x10
jmp prnext
prdone:
jmp (si)             !Continuar después de la cadena
.data
!Extendida lectura/escritura usando como parámetro el paquete
ext_rw:
.data1 0x10          !Tamaño del paquete r/w
.data1 0             !Reservado
.data2 1             !Bloques para la transferencia (sólo uno)
.data2 LOADOFF       !Buffer dirección de desplazamiento
.data2 0             !Buffer dirección de desplazamiento
.data4 0             !Número inferior de inicio del bloque 32 bits
zero:
        .data4 0             !Número superiorde inicio del bloque 32 bits
```

7.8 *Bootblock*

En esta sección serán explicados los principales registros utilizados por el código fuente del programa *bootblock.s*:

Registro	Bits	Descripción
DS	16	Data Segment, número apunta a los datos activos del segmento.
BP	16	Base pointer, utilizado para pasar datos desde y hacia la pila.
SP	16	Stack pointer, número que indica el desplazamiento que está utilizando la pila.
ES	16	Extra Segment, número que apunta a la participación activa del segmento extra.
SI	16	Source index, utilizado por las operaciones de cadena en las fuentes.
DI	16	Destination index, utilizado por las operaciones de cadena como destinos.

Registro	Bits	Descripción
AX	16	Accumulator Register, utilizado para los cálculos y para la E/S.
BX	16	Base Register, registro que solo puede ser utilizado como un índice.
CX	16	Count Register, registro utilizado para instrucciones de ciclos.
DX	16	Data Register, utilizado para la E/S, multiplicar y dividir.

Algunos datos con los que vienen configurados los registros.

Registro	Bits	Descripción de la función o parámetros de inicio que ellos poseen
DL	16	Registro de datos, contiene el dispositivo donde será cargado el *boot* secundario. Discos duros = 0x80, 0x81, 0x82, 0x83; Disquete = 0x00, 0x01.
es:bx	32	Es utilizado como buffer, representa la dirección de memoria donde serán almacenados los datos que se están leyendo.
es:si	32	Representa la dirección donde empieza la entrada a la tabla de particiones, si es un disco duro

En la siguiente sección de código se definen un conjunto de variables globales que son de gran utilidad para el manejo del cargador.

Variable	Valor	Descripción
LOADOFF	0x7C00	Al ser leído el primer sector de la unidad arrancable, dicha información es cargada en la región 0x0000:0x7C00 de la memoria principal.
*BOOT*SEG	0x1000	Al obtenerse el *boot* secundario este será cargado a partir de dicha de dirección de memoria.
*BOOT*OFF	0x0030	Desplazamiento del *boot* secundario por encima de la cabecera.
BUFFER	0x0600	Dirección donde comienza los sectores de memoria libres.
LOWSEC	8	Desplazamiento del primer sector lógico en la tabla de particiones LBA
device	0	Información sobre el dispositivo de arranque.
lowsec	2	Desplazamiento de la partición dentro de la unidad, es la LBA del sector de inicio de la partición.
secpcyl	6	El número de sectores por cilindro, número de cabezas * sectores.

Al culminar la inicialización de variables se presenta la sección de código referente a las instrucciones en ensamblador para iniciar el procedimiento de arranque. En esta sección se prepara la memoria principal para que esta pueda alojar el código obtenido en la unidad arrancable, al final se realiza una verificación acorde a los valores del registro “dl” para verificar la naturaleza del dispositivo, en caso de ser un disquete este se dirigirá a la etiqueta floppy dentro del código sino seguirá su orden secuencial, es decir, accede a la etiquete winchester. Para entender de primer plano vea los pasos que sigue el código *bootblock.s*:

- Creación del entorno de trabajo: inicializa ds=ss=ax=0, sp=bp=0x7C00.
- Se comprueba si se va a cargar de disco duro o de floppy.
    - Si es un disco duro, se obtienen los parámetros de la unidad y salta a cargar el *boot* secundario.
    - Si es un floppy, hay que determinar qué tipo de unidad se trata. El proceso que se sigue es sencillo, se tiene una variable que contiene los parámetros de las distintas unidades posibles (3.5’’ alta densidad, 3.5’’ baja densidad, 5.25’’ alta densidad y 5.25’’ baja densidad) y se lee el último sector de la primera pista, si falla la prueba se sigue con la siguiente unidad y así sucesivamente. Cuando se determina el tipo de unidad, se salta a cargar el *boot* secundario (monitor).
- El monitor se empieza a cargar en la posición 0x1000:0x0000. Se entra en un proceso iterativo: Mientras queden sectores por leer del monitor:
    - Cargar el sector especificado en la posición de memoria es:bx.

- Se modifica la próxima posición de memoria: es:bx+512
- Si existe error, mostrarlo en pantalla y quedarse en un bucle infinito.

- Cuando están todos los sectores del monitor cargados en memoria, se salta a la posición de memoria adecuada (0x1000:0x0000) para ceder el control al monitor.

```
LOADOFF = 0x7C00    !0x0000:LOADOFFposición de memoria donde es cargado este código
BOOTSEG = 0x1000    !Donde el boot secundario (monitor) se empieza a cargar, la posición es 0x1000:0x0000.
BOOTOFF = 0x0030    !Desplazamiento del boot secundario por encima de la cabecera
BUFFER= 0x0600              !Dirección donde comienza la memoria libre
LOWSEC=8                    !Desplazamiento del primer sector lógico en la tabla de particiones = LBA

device=0            !El dispositivo de arranque
lowsec=2            !Desplazamiento de la partición dentro de la unidad, utilizado para obtener valor de la pila
secpcyl =6          !Sectores por cilindro = cabezas * sectores

.text

!Inicio del procedimiento de arranque
boot:                       !Creación del entorno de trabajo: inicializa ds=ss=ax=0, sp=bp=0x7C00
xorax, ax           !ax = 0x0000, El vector de segmento
movds, ax           !ds = Segmento de datos - apunta a los datos activos del segmento
cli                 !Ingnora las interrupciones mientras inicializa la pila
movss, ax           !ss = ds = vector segment (ss APUNTA AL SEGMENTO DE LA PILA)
movsp, #LOADOFF     !Lugar usual para la pila del bootstrap - sp = indica el desplazamiento de la pila = 0x7C00
sti                 !habilita las interrupciones

push ax
push ax             !Apila un cero en lowsec(bp), para almacenar luego a la dirección del primer sector de la partición
push dx             !Apila el dispositivo de arranque almacenado en dl sera = device(bp)
movbp, sp           !Actualiza el marco de la pila
push es
push si             !Apila a es:si = entrada a la tabla de particiones
movdi, #LOADOFF+sectors     !char *di = sectors; | di = Destination Index - utilizado por las operaciones de cadena como destino

testb dl, dl                !Si el dispositivo es un disco dl >= 0x80, activa la bandera sign
jgefloppy           !Si no es negativo, entonces el dispositivo es un diskette y salta a floppy
winchester:
!Obtiene el desplazamiento del primer sector de la partición de arranque desde la tabla de partición
!La tabla se encuentra en es:si, el parámetro lowsec en desplazamiento LOWSEC.
!Los 4 bytes en es:si+LOWSEC son copiados en la dirección bp+lowsec
```

```
eseg                    !eseg usa el registro es (en vez de ds) como el registro segmento
lesax, LOWSEC(si)       !es:ax = LOWSEC+2(si):LOWSEC(si) | LES para cargar el registro extra segment, lee a LBA
movlowsec+0(bp), ax             !Apilar los 16 bits inferiores del primer sector de la partición
movlowsec+2(bp), es             !Apilar los 16 bits superiores del primer sector de la partición
!Si el dispositivo es un disco duro, obtiene los paramertos de la unidad.
!Si es un diskette el número de sectores se conocen y estan escritos en un arreglo llamado 'sectors'

movb ah, #0x08          !El número de la función que lee los parámetros del disco es ah = 0x08
int 0x13                !dl contiene aun el dispositivo
!La función (int 0x13) retorna el máximo número de sectores en los bits 0-6 de cl y el máximo número de !cabeceras en dh. Sin embargo,
suma una confusión, debido a que el máximo número de cabeceras tiene el !siguiente formato 0-origen, en consecuencia se debe sumar 1
al resultado

andb cl, #0x3F          !cl = máximo número de sectores (1-origen)
movb (di), cl           !En (di) se almacena el número de sector por pistas
incb dh                 !dh = 1 + máximo número de cabeceras (0-origen)
jmpload boot

!Floppy:
!Ejecuta 3 tests de lectura para determinar el tipo de unidad. Prueba para cada tipo de disquete mediante la lectura del !último sector de la
primera pista. Si esto falla, intenta un tipo que tenga menos sectores. Por lo tanto comenzamos con !1.44M (18 sectores) luego ;con 1.2M
(15 sectores) y finaliza con 720K/360K (ambos con 9 sectores). Usa el arreglo sectors
next:
incdi       !Siguiente número de sectores por pista (si es necesario)

floppy:
xorb ah, ah             !Resetea el dispositivo especificado por dl con la llamada a int 0x13, ah=0x00
int 0x13
movb cl, (di)           !cl = número del último sector por pista
cmpb cl, #9             !No hay necesidad de hacer la prueba con los últimos tipos de diskette 720K/360K
je success
!Intenta leer el último sector en la pista 0

moves, lowsec(bp)       !es = vector de segmento (lowsec = 0)
movbx, #BUFFER          !es:bx buffer = 0x0000:0x0600
movax, #0x0201          !Lee un sector con la llamada a int 0x13 ah=0x02. dicha función lee sectores desde el dispositivo
!especificado y los copia a memoria
xorb ch, ch             !Pista 0, último sector
xorb dh, dh             !Unidad dl, cabecera 0
int0x13
jcnext                  !Error, intenta con el siguiente tipo de diskette
success:
```

```
movb dh, #2             !Carga número de cabezas para multiplicar(en diskette siempre es 2), el número de sectores está aún en cl
load boot:
!Carga el código del boot secundario desde el dispositivo de arranque
movb al, (di)           !al = (di) = sectores por pista
        mulb dh         !dh = cabezas, ax = cabezas * sectores
        movsecpcyl(bp), ax  !Sectores por cilindro = cabezas * sectores

movax, #BOOTSEG         !Segmento para cargar dentro el código del boot secundario
moves, ax
xorbx, bx               !Load first sector at es:bx = BOOTSEG:0x0000
movsi, #LOADOFF+addresses   !Comienzo de la dirección del código del boot


load:
movax, 1(si)            !Obtiene el próximo número de sector: 16 bits inferiores
movb dl, 3(si)          !Bits 16-23 para tu disco de 8GB
xorb dh, dh             !dx:ax = sector sin la partición
addax, lowsec+0(bp)
adcdx, lowsec+2(bp)     !dx:ax = sector dentro de la unidad | Suma con acarreo
cmpdx, #[1024*255*63-255]>>16       !Cerca del límite de 8G?
jaebigdisk              !El salto se realiza si cf esta desactivada
divsecpcyl(bp)          !ax = cilindro, dx = sector dentro del cilindro
xchg ax, dx             !ax = sector dentro del cilindro, dx = cilindro | xchg intercambia el contenido de los registros
movb ch, dl             !ch = 8 bits inferiores del cilindro
divb (di)               !al = cabecera, ah = sector (0-origin)
xorb dl, dl             !Desplazar los bits 8-9 del cilindro en dl
shrdx, #1               !desplaza todos los bits (tantas posiciones como lo indique el inmediato) hacia la derecha e inserta cero en la izquierda
shrdx, #1               !dl[6..7] = cilindro superior
orbdl, ah               !dl[0..5] = sector (0-origin)
movb cl, dl             !cl[0..5] = sector, cl[6..7] = cilindro superior
incb cl                 !cl[0..5] = sector (1-origin)
movb dh, al             !dh = al = cabecera
movb dl, device(bp)     !dl = dispositivo para leer
movb al, (di)           !Sectores por pista – Número de sector (0-origen)
subb al, ah             != Sectores que restan en la pista
cmpb al, (si)           !Compara con el número de sectores a leer
jberead                 !No puede leer después del final del cilindro?
movb al, (si)           !(si) < Sectores que restan en la pista
read:
push ax                 !Apila al = sectores para leer
movb ah, #0x02          !Código para leer del disco !ah = 0x02 Lee sectores de un dispositivo, al = contador de sectores a leer, ch = pista, !cl =
sector, dh = cabecera, dl = dispositivo, ES:BX = buffer
int 0x13                !Call the BIOS for a read
Popcx                   !Restore al in cl
```

```
jmprdeval
bigdisk:
movb cl, (si)            !Número de sectores a leer
push si                  !Apila a si
movsi, #LOADOFF+ext_rw   !si = si = extendida lectura/escritura usando como parámetro el paquete
movb 2(si), cl           !Rellena # bloques para la transferencia
mov4(si), bx             !Direction del Buffer
mov8(si), ax             !Número de inicio del bloque = bx:ax
mov10(si), dx
movb dl, device(bp)      !dl = dispositivo a leer
movb ah, #0x42           !Lectura extendida
int 0x13                 !AH = 0x42 número de la función para la lectura extendida
popsi                    !Restaurar si para que apunte a la dirección del arreglo
!jmp rdeval
rdeval:
jcerror                  !Si ocurrio un error salta a error
movb al, cl              !Restarura al = rectores a leer
addb bh, al              !bx += 2 * al * 256 (suma los bytes leidos)
addb bh, al              !es:bx = donde el siguiente sector debe ser copiado
                                 !Si se añade a 2 bh, es equivalente a la adición de 512 a bx (recuerde que un sector = 512 bytes).

add1(si), ax             !Actualizar la dirección del sector a leer
adcb 3(si), ah           !No olvidar los bits 16-23 (sumar ah = 0)
subb (si), al            !Decrementar el contador de sector por sectores leídos
jnzload                  !Sino todos los sectores han sido leídos
addsi, #4                !Siguiente par (dirección , contador)
cmpb ah, (si)            !Cuando no hay sectores a lee
jnzload                  !Lee el siguiente trozo del código del boot secundario

done:
!Llama al boot secundario, asumiendo una cabecera larga a.out (48 bytes).
!La cabecera a.out es normalmente pequeña (32 bytes), pero el boot
!secundario tiene dos puntos de entrada: Uno es el desplazamiento 0, para
!la cabecera larga, y el otro es el ;desplazamiento 16 para la cabeceracorta.

!dl=Dispositivo Boot.
!es:si= Entrada de la tabla de partición si es el disco duro.

popsi    !Restablecer es:si = entrada en la tabla de partición
popes    !dl está aún cargado
jmpf BOOTOFF, BOOTSEG    !Saltar al sector del boot (saltando a la cabecera), aqui es donde al fin le sede el control a boothead.s
!Read error: imprimir mensaje, bucle infinito | cuando ocurre algun error
error:
movsi, #LOADOFF+errno+1
```

```
prnum:
movb al, ah          !Número de error en ah
andb al, #0x0F !4 bits inferiores
cmpb al, #10!A-F?
jbdigit!0-9!
addb al, #7 !'A' - ':'
digit:
addb (si), al        !Modificar '0' en string
decsi
movb cl, #4          !Proximos 4 bits
shrb ah, cl
jnzprnum             !De nuevo si digit > 0
movsi, #LOADOFF+rderr!String a imprimir
print:
lodsb                !al = *si++ es el carácter a ser imprimido
testbal, al          !byte null marca que indica fin
hang:
jzhang               !Manejador siempre esperando CTRL-ALT-DEL
movb ah, #0x0E       !Imprimir caracter en modo teletype
movbx, #0x0001       !Pagina 0, color de primer plano
int0x10              !Llama a BIOS VIDEO_IO
jmpprint
.data
rderr:
.ascii"Read error "
errno:
.ascii"00 \0"
errend:
!Varios sectores por pista dependiendo del tipo de disquete 1.44M, 1.2M and 360K/720
sectors:
.data118, 15, 9!Número de sectores por disco
!Comandos extendicos de lectura/escritura que requiren un paquete de parametros
ext_rw:
.data10x10           !Tamaño del paquete r/w
.data10              !Reservado
.data20              !Bloques para la transferencia
.data20              !Buffer dirección de desplazamiento
.data2BOOTSEG        !Buffer dirección del segmento
.data40              !Número inferior de inicio del bloque 32 bits
.data40              !Número superior de inicio del bloque 32 bits
.align2
addresses:
                     !El espacio ocupado luego de este código es para las direcciones de disco para unprograma de boot
                     !secundario (en el peor de los casos, cuando el archivo está fragmentado). Esto debería ser suficiente.
```

8 Implementación de un intérprete de comandos

Un intérprete de comandos es un software que proporciona una interfaz para los usuarios de un sistema operativo, el cual provee acceso a los servicios del núcleo. El nombre intérprete de comandos viene dado por el hecho de ser una capa externa (interfaz) entre el usuario y sistema operativo (funcionamiento interno del núcleo).

El intérprete de Minix3 no es parte del SO, pero utiliza fuertemente muchas de las características del SO, por lo cual es un buen ejemplo de uso de las llamadas al sistema. Si el usuario escribe cualquier orden válida, el intérprete de comandos crea un proceso hijo y el cual es el que ejecuta el programa para satisfacer la orden recibida. Mientras el hijo está corriendo, el intérprete de comandos espera a que termine. Cuando el hijo termina, el intérprete de comandos pone el prompt y nuevamente queda esperando un mandato en la entrada estándar (la terminal en este caso).

Para completar este laboratorio los estudiantes deben implementar un programa intérprete de comandos. El programa resultante es muy parecido a los intérpretes de comando de Unix/Linux. Para la implementación de este laboratorio, se utilizara un código plantilla que consiste en tres archivos:

- Shell.l: ofrece un programa de captura por entrada estándar (la función getline()), que se puede utilizar para controlar el flujo de entrada del usuario.
- myshell.c: contiene un código esqueleto de un intérprete de comandos simple.
- Makefile: contiene todo lo necesario para compilar Shell.l y myshell.c.

La solución debe satisfacer los requerimientos mencionados en el capítulo Adecuación de Minix 3 a la UCV, como se allí se menciona la solución debe ser implementada en myshell.c, la cual puede observarse a continuación en la Figura 8.1. Cabe destacar que existe también para este laboratorio un video tutorial el cual utiliza como solución el código que se muestra a continuación. Para dicha implementación se usaron las siguientes llamadas al sistema:

- *Fork*
- *Execvp*
- *Wait*
- *Exit*
- *Close*
- *Dup*
- *Pipe*

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <string.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <sys/select.h>
/* Funcion que permite obtener la cadena introducida
 * en la consola
 */
extern char **getline(void);
char **args;
pid_t pid,childpid;
int logArch,logPipe,logFuente;
char *archivo,*archivo1;
int file;
int i,inicio,fin;
char *ejecutar[100];
int logError;

/* Dicha funcion solo le asigna
 * el valor de args a ejecutar
 */
void cambiar(void){
	int j,k;
	k=0;
	for(j = inicio; j < fin ; j++) {
		if(args[j]!=NULL){
			ejecutar[k++]=args[j];
		}
	}
	ejecutar[k++] = NULL ;
}

/* Una vez obtenida la cadena desde la consola
 * se busca el primer comando valido a ejecutar,
 * si activan las banderas si existe un caracter
 * '>' o '|' para poder trabajarlos de manera
 * especial
 */
int obtenerInstruccion(void){
	for(i = inicio; args[i] != NULL ; i++) {


		if(strcmp(args[i],">") == 0){
			if(args[i+1]!= NULL){
				archivo = args[i+1];
logArch=1;
				break;
			}else{
				printf("Error en sixtaxis: no se esperaba nueva linea\n");
				logError=1;
				break;
			}
}else if(strcmp(args[i],"<") == 0){
			if(args[i+1]!= NULL){
				archivo1 = args[i+1];
				logFuente=1;
				break;
```

```
}else{
                    printf("Error en sixtaxis: no se esperaba nueva linea\n");
                    logError=1;
                    break;
                }
            }else if(strcmp(args[i],"|") == 0){
                if(args[i+1]!= NULL){
                    logPipe=1;
                    break;
                }else{
                    printf("Error en sixtaxis: no se esperaba nueva linea\n");
                logError=1;
                    break;
                }
            }
        }

        if(logError){
            return 0;
        }else{
            fin=i;
            return 1;
        }
}

/* Ya obtenida la instruccion a ejecutar
 * se realiza un fork y luego un excec
 * para poder ejecutar el comando obtenido.
 * Esta funcion es recursiva para poder ejecutar
 * tantas instrucciones como el comando lo
 * amerite
 */
int Ejecutar(void){
  int status,hacer,j;
  pid_t pid;
  int fd[2];
  hacer=obtenerInstruccion();

  if(hacer){
      switch(pid = fork()){
                case -1:
                    perror("fork error");
                    break;
            case 0:
            cambiar();
            if(logPipe==1){
                    pipe(fd);
                if((childpid = fork()) == -1){
                    perror("fork");
                    exit(1);
                }
                if(childpid == 0){
                    close(fd[0]);
                    dup2(fd[1],1);
                    close(fd[1]);
                    execvp(ejecutar[0], ejecutar);
                }else{
                        close(fd[1]);
                    dup2(fd[0],0);
                    close(fd[0]);
                    inicio=fin;
                    inicio++;
                    hacer=obtenerInstruccion();
                    cambiar();
```

```
                        if(logArch){
                        file = open(archivo, O_CREAT | O_RDWR);
                        close(1);
                        dup(file);
                        close(file);
                    }
                    execvp(ejecutar[0], ejecutar);
                }

            }else{
                if(logArch){
                    file = open(archivo, O_CREAT | O_RDWR);
                    close(1);
                    dup(file);
                    close(file);
                }
                if(logFuente){
                    file = open(archivo1,O_RDONLY);
                    close(0);
                    dup2(file,0);
                    close(file);
                }
                            execvp(ejecutar[0], ejecutar);

                            /* Sino se puedo ejecutar el comando con
                             * la instuccion anterior se usa lo siguiente
                             */

                            execve( strcat ("/usr/bin/",ejecutar[0]), ejecutar, ejecutar);

                            if(execvp(ejecutar[0], ejecutar) == -1){
                                printf("******* Error: comando no encontrado *******\n");
                                exit(0);
                            }
        }

        default:

            if ((pid = wait(&status)) == -1){

                perror("wait error");
            }else{
                if(WIFSIGNALED(status) != 0){
                }else if(WIFEXITED(status) != 0){
                }else{
                    }
        }
    }
    inicio=fin;
    if(logPipe==1){
        logPipe=0;
        inicio++;
        hacer=obtenerInstruccion();
        inicio=fin;
        inicio++;
    }
if(logArch==1){

        logArch=0;
        inicio+=2;
    }
```

```
        if(logFuente==1){

                logArch=0;
                inicio+=2;
        }

        if(args[inicio]!=NULL){
                Ejecutar();
                return 1;
        }
        return 0;
 }
  return 0;
}
/* cilo infinito a la espera de comandos
 * a ejecutar, sale del ciclo si el usuario
 * intenta ejecutar "exit" o "quit"
 */
void main(void) {

        while(1) {
                logArch = 0;
                logPipe = 0;
                logError = 0;
                inicio = 0;
                logFuente = 0;
                fin = 0;

                printf("MiShell-$ ");

                args = getline();


                if(args[0]==NULL){
                        continue;
                }

                if(strcmp(args[0],"exit") == 0 || strcmp(args[0], "quit") == 0 ){
                        printf("Saliendo del Shell...\n");
                        sleep(1);
                        exit(0);
                }

                Ejecutar();

        }
}
```

Figura 8.1 Código fuente de un Shell simple

9 Implementación de llamadas al sistema

Una llamada al sistema es la forma en la cual un proceso requiere de un servicio al núcleo (al cual generalmente no tiene permisos para ejecutar). Las llamadas al sistema proporcionan la interfaz entre un proceso y el SO. La mayoría de las operaciones para interactuar con el sistema requieren permisos, los cuales no están disponibles para un proceso en la capa de usuario. Por ejemplo, realizar una operación de E/S (Entrada/Salida) con un dispositivo en el sistema, o cualquier otra forma de comunicación con otros procesos se requiere del uso de las llamadas al sistema (2).

Existe la posibilidad de que el uso inadecuado de la llamada al sistema pueda afectar la ejecución del SO. El diseño de la arquitectura del microprocesador en prácticamente todos los sistemas modernos (con excepción de algunos sistemas embebidos) ofrece modos de ejecución de la CPU (Unidad Central de Procesamiento). Esto ofrece niveles de privilegio, uno de ellos es el modo usuario, donde las aplicaciones tienen limitaciones en el espacio de direcciones debido a que no pueden acceder o modificar otras aplicaciones en ejecución, ni al propio SO. También impide el acceso directo de las aplicaciones hacia los dispositivos del computador. Pero las aplicaciones, obviamente, necesitan estas habilidades, por lo tanto, las llamadas al sistema ofrecen estos servicios a través del SO. El SO tiene más privilegios y se ejecuta en modo protegido, esto permite a las aplicaciones solicitar servicios mediante las llamadas al sistema, que a menudo se implementan a través de las interrupciones.

En general, se ofrece una biblioteca que se define entre los programas de usuarios y el SO. Existe una confusión entre los términos llamada al sistema y las funciones de la biblioteca estándar C. Hay que tener claro que en las llamadas al sistema se transfiere el control al núcleo.

9.1 *Llamadas al sistema en Minix 3*

El SO Minix 3 está basado la estructura micronúcleo (a diferencia de la estructura monolítica la cual es más común). Teniendo este tipo de núcleo significa que es menor el código que se ejecuta en el modo privilegiado de la CPU (también se conoce como modo núcleo). La mayoría del código se ejecuta con menos privilegios (en espacio de usuario). Además, Minix 3 está diseñado en capas, como se explico en el capítulo 2, estas capas pueden verse en la Figura 6.1.

El micronúcleo maneja las comunicaciones entre los procesos, realiza la planificación de los mismos, maneja interrupciones y provee algunos mecanismos básicos para la administración de procesos. El manejo de sistemas de archivo, funciones de red, administración de procesos y demás servicios a usuarios, son provistos por servidores especializados fuera del micronúcleo.

La comunicación entre los diferentes componentes es a través de pase de mensajes. Una ventaja de este enfoque es que la parte del SO que se ejecuta en modo privilegiado es mínima, por lo tanto, más fáciles de mantener libre de errores. Los errores en la parte del SO que se ejecuta en el espacio de usuario no tiene la capacidad afectar la estabilidad del sistema.

Una desventaja de este enfoque es que hay una reducción del rendimiento debido a que el pase de mensajes implica una sobrecarga, relacionada a la construcción, copia y envío. En caso de un núcleo monolítico sólo tiene un espacio de direcciones, esto implica que cualquier porción de código interna al núcleo puede obtener el control del SO, es decir, como estas porciones son ejecutadas en modo privilegiado no tienen ningún tipo de restricciones. Por ejemplo, en Minix 3, el FS (Servidor de Archivos), PM (Servidor Manejador de Procesos) y otros componentes del SO (incluyendo los controladores de dispositivo) se ejecutan como procesos en espacio de usuario, por lo tanto, ellos dependen de las llamadas al sistema para poder realizar operaciones privilegiadas.

Una de las funciones principales del micronúcleo es proveer un conjunto de funciones a los controladores de dispositivos y a los servidores que están en las capas inmediatamente superiores, estas funciones son denominadas llamadas al sistema. La encargada de atender y realizar esas llamadas es la Tarea de Sistema o System Task.

Desde el punto de vista del núcleo, todos los procesos de las capas superiores son tratados casi de la misma forma: todos son planificados por el núcleo, están limitados a usar instrucciones en modo usuario, ninguno puede acceder directamente a puertos de E/S y ninguno puede acceder a direcciones de memoria fuera del espacio asignado a sí mismo.

La diferencia entre los procesos que pertenecen a las distintas capas radica, principalmente, en la posibilidad de realizar llamadas al núcleo siendo los de la capa 2 los más privilegiados, seguidos en orden decreciente por la capa 3 y 4. Por ejemplo, los procesos de capa 2 (Controladores de Dispositivos) tienen permitido requerirle al System Task que lea y escriba datos en los puertos de E/S o que copie datos en el espacio de direcciones de otro proceso.

Es importante distinguir entre llamadas al núcleo y llamadas al sistema POSIX (POSIX System calls). Las primeras son llamadas de bajo nivel provistas por la System Task para permitirle hacer su trabajo a los controladores de dispositivos y a los Procesos Servidores. En contraste las llamadas al Sistema POSIX son llamadas de alto nivel definidas por el estándar POSIX y están disponible para los procesos de usuario en la capa 4 (1).

El controlador de dispositivo o proceso servidor tiene permitido intercambiar mensajes con un grupo acotado de otros procesos. Los mensajes pueden fluir entre procesos de la misma capa o entre procesos

de capas adyacentes. Los procesos de la capa 4 no pueden enviar mensajes a otros procesos de la capa 4, sino sólo a los de capa 3.

Para entender mejor como son las llamadas al sistema en Minix 3, vea la Figura 9.1. Las flechas indican los mensajes intercambiados entre las distintas capas de Minix 3. La idea, muy simplificada, es la siguiente (21):

- El proceso de usuario hace una llama al sistema. Desde su punto de vista, no es más que una función de biblioteca. En este ejemplo, la función forma parte de la libc.
- La función de la biblioteca es armar un mensaje con la petición y lo envía a uno de los procesos servidores quedando a la espera de una respuesta. En el ejemplo, se utilizó el Process Manager como destinatario del mensaje.
- El proceso servidor recibe el mensaje y, dependiendo del tipo de mensaje, ejecutará operaciones predefinidas. Si la llamada al sistema puede ser resuelta en el ámbito del proceso servidor, se genera un mensaje de respuesta d y se envía al proceso que realizó la petición. Si la llamada no puede ser resuelta por el proceso servidor, se generará un mensaje y se enviará al manejador de dispositivos correspondiente o, en su defecto a la SYSTASK. En el ejemplo, la petición es realizada a la SYSTASK vía el mensaje b.
- La SYSTASK realiza las acciones que correspondan ante un mensaje del tipo b y genera, eventualmente, un mensaje c con la respuesta a la petición. La respuesta, no está demás decirlo, es enviada al proceso servidor que emitió el mensaje b; no al proceso de usuario.
- El proceso servidor (el PM, en el ejemplo) toma el mensaje c, lo procesa y envía la respuesta al proceso de usuario vía el mensaje d

La función de biblioteca desempaqueta el contenido del mensaje y devuelve los resultados, posiblemente vía parámetros pasados por referencia, al proceso de usuario.

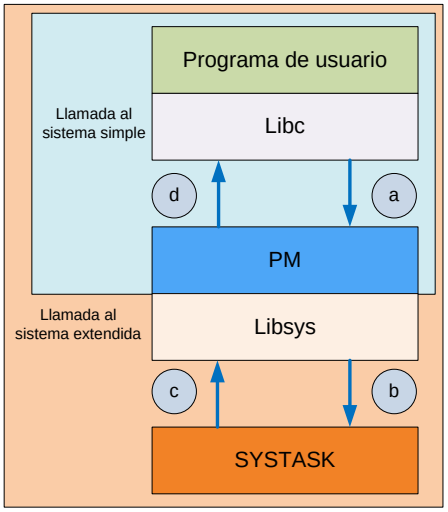


Figura 9.1 Flujo de información en la nueva llamada al Sistema

9.2 *Implementación de Llamadas al Sistema*

Minix 3 está basado en servidores, existen dos servidores importantes el FS y el PM, entre otros. El FS se encarga del manejo de archivos (creación, eliminación, etc.) y el PM se encarga de todo lo referente a los procesos del sistema. Estos servidores son los que van a permitir el desarrollo de la nueva llamada al sistema. Antes de implementar las llamadas al sistema se debe conocer algunas funciones definidas en Minix 3.

9.2.1 Funciones relacionadas con llamadas al sistema

Estas funciones ayudaran a una implementación rápida y eficaz de una llamada al sistema. La primera de ellas es la *_syscall*. Esta llamada al sistema, realiza el envío de un mensaje destinado a otro proceso y se bloquea hasta recibir una respuesta. Utilizándola es la forma más sencilla de realizar una llamada al sistema. *_syscall* debe recibir tres parámetros, vea su implementación en el */usr/src/lib/other/syscall.c* o en la Figura 9.2 y su sintaxis es la siguiente:

```
PUBLIC int _syscall(int who, int syscallnr, register message *msg);
```

- ✓ who: es el destinatario del mensaje, en este caso el servidor.
- ✓ syscallnr: es el número de la llamada al sistema
- ✓ msgptr: es un apuntador al mensaje a enviar. *message* es una estructura de datos definida por Minix 3 en el directorio */usr/src/include/minix/ipc.h.*

En _syscall si el valor de retorno es negativo será tratada como un error. Los valores estándares de error se define en */usr/src/include/errno.h.* De lo contrario el valor de retorno es debería ser 0.

syscall.c

```
1#include <lib.h>
2
3PUBLIC int _syscall(who, syscallnr, msgptr)
4int who;
5int syscallnr;
6register message *msgptr;
7{
8  int status;
9
10  msgptr->m_type = syscallnr;
11  status = _sendrec(who, msgptr);
12  if (status != 0) {
13    /* 'sendrec' itself failed. */
14    /* XXX - strerror doesn't know all the codes */
15    msgptr->m_type = status;
16  }
17  if (msgptr->m_type < 0) {
18    errno = -msgptr->m_type;
19    return(-1);
20  }
21  return(msgptr->m_type);
22}
```

Figura 9.2 Función _syscall

Como puede observarse *_syscall* utiliza a *sendrec* con el destinatario who (normalmente a PM o FS). *Sendrec* hace una petición la cual es recibida por *get_work*, y se encarga de responder a la misma a través de *reply*. Las funciones *get_work* y *reply* existe tanto en las PM como en FS. Para entender mejor a _syscall vea la siguiente llamada al sistema:

```
message m;

_syscall(MM,70,&m);
```

Donde MM es el número de proceso asignado al PM, vea la Figura 9.3 en la línea 26 del archivo */usr/src/include/lib.h*, donde se define dicha variable. El segundo parámetro 70, es el número de llamada al sistema y m es el mensaje a enviar.

```
/* The <lib.h> header is the master header used by the library.
 * All the C files in the lib subdirectories include it.
 */

#ifndef _LIB_H
#define _LIB_H

/* First come the defines. */
#define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
#define _MINIX             1    /* tell headers to include MINIX stuff */

/* The following are so basic, all the lib files get them automatically. */
#include <minix/config.h>   /* must be first */
#include <minix/types.h>
#include <limits.h>
#include <errno.h>
#include <ansi.h>

#include <minix/const.h>
#include <minix/com.h>
#include <minix/type.h>
#include <minix/callnr.h>

#include <minix/ipc.h>

#define MM                 PM_PROC_NR
#define FS                 FS_PROC_NR

_PROTOTYPE( int __execve, (const char *_path, char *const _argv[],
            char *const _envp[], int _nargs, int _nenvps)   );
_PROTOTYPE( int _syscall, (int _who, int _syscallnr, message *_msgptr)  );
_PROTOTYPE( void _loadname, (const char *_name, message *_msgptr)   );
_PROTOTYPE( int _len, (const char *_s)                  );
_PROTOTYPE( void  begsig, (int  dummy)                  );
```

Figura 9.3 Código fuente de lib.h

Otra función utilizada en llamadas al sistema es *_taskcall*. Tiene la misma funcionalidad que *_syscall* excepto que devuelve los códigos de error negativo directamente y no en *errno*. Esta es una mejor interfaz para que MM y FS se comuniquen con el núcleo, la implementación de esta función está en */usr/src/lib/syslib/taskcall.c*, puede observarse en la Figura 9.4.

taskcall.c

```
/* _taskcall() is the same as _syscall() except it returns negative error
 * codes directly and not in errno.  This is a better interface for MM and
 * FS.
 */

#include <lib.h>
#include <minix/syslib.h>

PUBLIC int _taskcall(who, syscallnr, msgptr)
endpoint_t who;
int syscallnr;
register message *msgptr;
{
  int status;

  msgptr->m_type = syscallnr;
  status = _sendrec(who, msgptr);
  if (status != 0) return(status);
  return(msgptr->m_type);
}
```

Figura 9.4 Función taskcall.c.

9.2.2 ¿Cómo se crea una llamada al sistema?

La llamada al sistema propuesta utilizará como soporte al servidor PM. La comunicación entre el proceso de usuario y el PM involucra dos tareas:

a) La creación, por parte del proceso de usuario, de un mensaje y el envío del mismo al proceso servidor.
b) La creación de un manejador, en el proceso servidor, que realice las acciones pertinentes cuando llega un mensaje de ese tipo.

En cuanto a la generación del mensaje por parte del proceso de usuario, existen dos alternativas. Se desea implementar ambas alternativas de llamadas al sistema, estás se definen por enfoques; el directo e indirecto (a través de una biblioteca). Las llamadas al sistema propuestas serán descritas a continuación, desglosadas por enfoque:

- Enfoque directo: Para este ejemplo el programa de usuario debe conocer, manipular y utilizar la función *_syscall*, la cual será explicada posteriormente.
- Enfoque indirecto: es necesario la creación de una función dentro de una biblioteca que oculte todo el manejo de envío y recepción de mensajes entre las capas.

En la siguiente sección se explica paso a paso el proceso para la creación de cada tipo de llamada al sistema mencionado anteriormente.

9.2.3 Pasos para crear una llamada al sistema (enfoque directo)

En esta implementación se utilizará el enfoque directo. Eso significa, como se mencionó anteriormente, que el programa de usuario va a utilizar la llamada al sistema _syscall, que realiza el envío de un mensaje destinado a otro proceso y se bloquea hasta recibir una respuesta. Esta es la forma más sencilla de realizar una llamada al sistema. A continuación se describe paso a paso el proceso de la creación de una llamada al sistema en Minix 3 la cual solo imprime por pantalla: "Esta es una llamada al sistema":

Paso 1 (Implementar *prueba_imprimirmsg.c*): debe implementar en primer lugar un programa de usuario que permita invocar la llamada al sistema, el cual será denominado *prueba_imprimirmsg.c*, como se indico en la tarea a). Este programa de estar alojado en el directorio */usr/src.* Para crear este programa debe crear primero el archivo a través de la consola de Minix 3 use el siguiente comando: "*vi prueba_imprimirmsg.c*". Posteriormente introduzca ":", luego cuando aparezca en pantalla los dos puntos teclee *wq* para guardarlo y salir del editor de texto. Para finalizar utilice el IDE de eclipse para implementar el programa que puede visualizar en la Figura 9.5.

```
#include <lib.h>
#include <stdio.h>

/* Programa de prueba de la llamada
 * al sistema imprimirmsg */

void main(int argc, char *argv[])
{
        int retorno;
        message m;
        retorno = _syscall(MM,69,&m);

        printf("Resultado imprimirmsg:[%d]\n", retorno);
}
```

Figura 9.5 prueba_imprimirmsg.c

Paso 2.1 (Creación de un manejador - modificar el *table.c*): ahora se procede a la tarea b), para esto se debe empezar por encontrar una ranura o entrada vacía en el archivo */usr/src/servers/pm/table.c*, vea la Figura 9.6. Para agregar una nueva llamada al sistema, se debe identificar una ranura o entrada sin usar. Por ejemplo, el índice 69 contiene una entrada no utilizada, fácil de identificar debido a que dice *unuse*. Se podría utilizar el número de ranura 69 para la llamada al sistema de *do_imprimirmsg*. Para utilizar la entrada 69, se reemplaza *no_sys* con *do_imprimirmsg*, vea la Figura 9.7 en la línea 83 del *archivo /usr/src/servers/pm/table.c.*

```
do_imprimirmsg,                 /*          69 = Llamada nueva = imprimirmsg          */
```

```
 1/* This file contains the table used to map system call numbers onto the
 2 * routines that perform them.
 3 */
 4
 5#define _TABLE
 6
 7#include "pm.h"
 8#include <minix/callnr.h>
 9#include <signal.h>
10#include "mproc.h"
11#include "param.h"
12
13_PROTOTYPE (int (*call_vec[]), (void) ) = {
14    no_sys,      /*  0 = unused  */
15    do_exit,     /*  1 = exit    */
16    do_fork,     /*  2 = fork    */
17    no_sys,      /*  3 = read    */
18    no_sys,      /*  4 = write   */
19    no_sys,      /*  5 = open    */
20    no_sys,      /*  6 = close   */
21    do_waitpid,  /*  7 = wait    */
```

Figura 9.6 Código fuente de table.c

```
78    do_itimer,  /* 64 = itimer  */
79    do_get,     /* 65 = getgroups */
80    do_set,     /* 66 = setgroups */
81    no_sys,     /* 67 = unused  */
82    no_sys,     /* 68 = unused  */
83    no_sys,     /* 69 = unused  */
84    no_sys,     /* 70 = unused  */
85    do_sigaction,   /* 71 = sigaction   */
86    do_sigsuspend,  /* 72 = sigsuspend  */
```

```
78    do_itimer,  /* 64 = itimer  */
79    do_get,     /* 65 = getgroups */
80    do_set,     /* 66 = setgroups */
81    no_sys,     /* 67 = unused  */
82    no_sys,     /* 68 = unused  */
83    do_imprimirmsg,     /* 69 = Nueva llamada al sistema (sencilla) */
84    no_sys,     /* 70 = unused  */
85    do_sigaction,   /* 71 = sigaction   */
86    do_sigsuspend,  /* 72 = sigsuspend  */
```

Figura 9.7 Código fuente de table.c (modificado)

Paso 2.2 (Creación de un manejador - modificar el *proto.h*): El siguiente paso es declarar un prototipo del manejador de sistema, esto se hace modificando el archivo */usr/src/servers/pm/proto.h.* Este archivo contiene los prototipos de todas las funciones manejadoras de las llamadas al sistema. Se debe añadir el prototipo de *do_imprimirmsg*, esto se muestra en la línea 10 de la Figura 9.8.

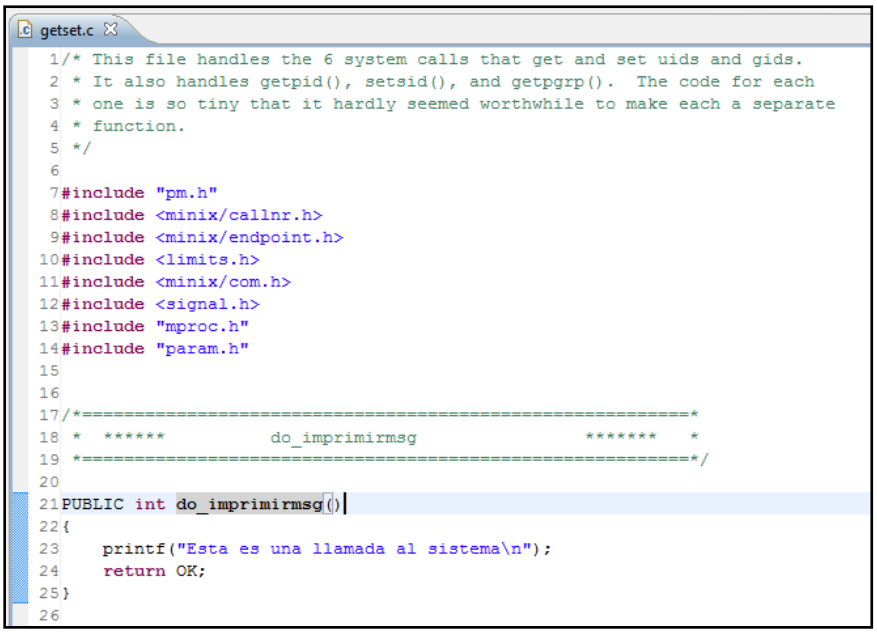
```
_PROTOTYPE( int do_imprimirmsg, (void)   );
```

proto.h

```
1 /* Function prototypes. */
2
3 struct mproc;
4 struct stat;
5 struct mem_map;
6 struct memory;
7
8 #include <timers.h>
9
10 _PROTOTYPE ( int do_imprimirmsg, (void) );
11
12 /* alarm.c */
13 _PROTOTYPE( int do_alarm, (void)                    );
14 _PROTOTYPE( int do_itimer, (void)                   );
15 _PROTOTYPE( void set_alarm, (struct mproc *rmp, clock_t ticks)      );
16 _PROTOTYPE( void check_vtimer, (int proc_nr, int sig)           );
```

Figura 9.8 Código fuente de proto.h

Paso 2.3 (Creación de un manejador - implementar *do_imprimirmsg*): Añadir la implementación de do_imprimirmsg a un archivo nuevo o un archivo existente en */usr/src/servers/pm/.* Si se agrega a un archivo existente, no es necesario cambiar *Makefile*, si prefiere crear un nuevo archivo, es necesario modificar el *Makefile*. En este ejemplo se utilizará un archivo existente en el directorio antes mencionado, dicho archivo es *getset.c*. Debido a que la llamada al sistema *do_imprimirmsg* solo va a imprimir datos a través de la salida estándar, la implementación de la misma solo tiene una llamada a la función *printf*. La implementación de esta llamada puede observarse en la Figura 9.9 desde la línea 17 a la 25.




```
/*=====================================================*
 *          ******          do_imprimirmsg      ****** *
 *=====================================================*/
PUBLIC int do_imprimirmsg()
{
  printf("Esta es una llamada al sistema\n");
  return OK;
}
```

Figura 9.9 Código fuente de getset.c

Paso 3 (Generar una versión): hay que recompliar el núcleo, para realizar este penúltimo paso se debe dirigir al directorio */usr/src/tools* y ejecutar los siguientes comandos:

```
# cd /usr/src/tools/
# make hdboot
```

Luego debe especificar la imagen desde donde se pretende iniciar, la misma puede verse una vez ejecutado el comando *make hdboot*, aparece una vez ejecutado el comando y tiene la siguiente sintaxis 3.1.5rX, donde X es un entero que hace referencia número de *release*. Debe ejecutar el comando:

```
d0p0s0> image=/boot/image/3.1.6rX
d0p0s0> boot
```

Paso 4 (compilación y ejecución del programa de usuario): Para concluir debe compilar el programa *prueba.c* y ejecutarlo. Para esto diríjase al directorio *(/usr/src/)* donde se encuentra *prueba_imprimirmsg.c* y ejecute los siguientes comandos:

```
# cd /usr/src/
# cc prueba_imprimirmsg.c -o prueba_imprimirmsg
# ./prueba_imprimirmsg
Esta es una llamada al sistema
Resultado imprimirmsg:[0]
```

9.2.4 Llamada al sistema (usando una biblioteca)

En esta implementación se utilizará el enfoque indirecto. Es necesaria la creación de una función dentro de una biblioteca que oculte todo el manejo de envío y recepción de mensajes entre las capas. Dicha comunicación ocurre entre el proceso de usuario (capa 4) y el PM (capa 3). De esta forma, se abstrae a los procesos de usuario del mecanismo de comunicación interprocesos. Esta llamada al sistema solo envía un entero y el resultado de la misma es el mismo entero multiplicado por dos, todo esto a través de pase de mensajes. Como se muestra en la Figura 9.10 donde a es la petición y d la respuesta.

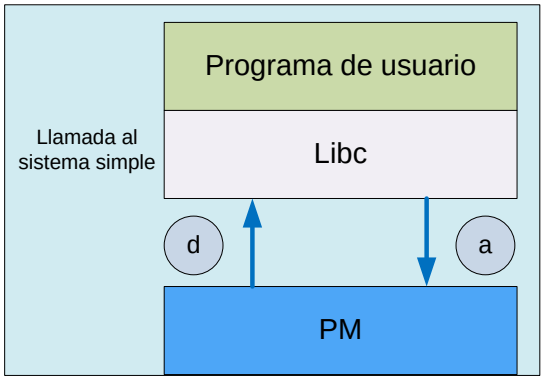


Figura 9.10 Llamada al sistema usando biblioteca (sencilla)

Antes de explicar los pasos de esta llamada es importante conocer la estructura de datos *message*. La cual puede observarse a continuación en la Figura 6.1. Luego será explicada esta llamada al sistema paso a paso (21).

```
29typedef struct {
30  endpoint_t m_source;       /* who sent the message */
31  int m_type;              /* what kind of message is it */
32  union {
33    mess_1 m_m1;
34    mess_2 m_m2;
35    mess_3 m_m3;
36    mess_4 m_m4;
37    mess_5 m_m5;
38    mess_7 m_m7;
39    mess_8 m_m8;
40    mess_6 m_m6;
41    mess_9 m_m9;
42  } m_u;
43} message;
44
```

Figura 9.11 Estructura message

Paso 1 (Construcción de la biblioteca): En */usr/src/lib/posix* se crea un archivo llamado *_newcall.c.* En este archivo se creará la función *newcall*, encargada de armar un mensaje y enviarlo al proceso servidor, en este caso al manejador de procesos. Si el proceso de usuario necesitara enviar información al proceso servidor, debería pasarla a *newcall* vía argumentos. De la misma forma, si el proceso servidor devuelve un mensaje con información. Luego, *newcall* debería desempaquetar el mensaje y enviar la información al proceso de usuario vía argumentos. La implementación del programa obsérvela en la Figura 9.12:

```
#include <lib.h>
#include <unistd.h>

PUBLIC long newcall (int entrada, int *salida)
{
 message m;
 int retorno;

 /* Se establece los campos del mensaje a enviar */
 m.m1_i1 = entrada;

 /* La forma de pasar un mensaje a MM es a través de:
   _syscall(MM,NEWCALL,&m);
 El mensaje de respuesta permanece en m.
 */
 retorno = _syscall(MM, NEWCALL, &m);

 /* Retornar la información contenida en el mensaje vía parametros*/
 /* *salida = m.m1_i2; */

 *salida = 1234;
 return(retorno);
}
```

Figura 9.12 Código fuente de newcall.c (versión 1)

Note que la función de ejemplo no hace mucho. Solo carga el campo m1_i1 del mensaje m que se enviará al PM y, luego de la llamada a _syscall, coloca el valor 1234 en la dirección de memoria suministrada como segundo parámetro. Por último, retorna el valor devuelto como resultado de la llamada a _syscall. Nota, si se quiere pasar otro tipo de dato que no es entero utilizar la estructura de datos message y buscar el campo requerido, el procedimiento sigue siendo el mismo. Para que esta función forme parte de la librería, se debe editar el archivo Makefile.in en el mismo directorio y agregar el nombre del nuevo archivo a compilar como parte de la libc.

```
# Makefile for lib/posix.

CFLAGS="-O -D_MINIX -D_POSIX_SOURCE"

LIBRARIES=libc

libc_FILES=" \
        __exit.c \
        _newcall.c \
        _access.c \
}
```

Luego ejecutar, en ese directorio:

```
# /usr/src/lib/posix

# Make Makefile
```

Esto generará un nuevo Makefile que incluye las reglas para el nuevo archivo. Como se puede observar en la llamada a _syscall, se ha usado la macro `NEWCALL` en lugar de utilizar el número de llamada al sistema. Entonces, en /usr/src/include/minix se debe editar el archivo *callnr.h* para agregar dicha macro:

```
#define NEWCALL 70 /* El 70 esta libre en la sección posix*/
```

Esto compilará e instalará la nueva librería POSIX. Ahora sólo queda agregar el prototipo de la función de la biblioteca newcall al archivo de que corresponda (en este caso a *unistd.h*) para que el compilador de C no de errores cuando deba generar el código para su llamada. En virtud de lo antes mencionado se edita */usr/src/include/unistd.h:*

```
_PROTOTYPE( long newcall, (int entrada, int *salida) );
```

Posteriormente se instala los nuevos archivos de cabecera en /usr/include/ ejecutando:

```
# cd /usr/src
# make includes
```

También se deben compilar las bibliotecas, para esto ejecute:

```
# cd /usr/src
# make libraries
```

Paso 2 (Construcción del programa de prueba): En este punto, solo a los efectos de prueba, se construirá un programa llamado */usr/src/prueba_newcall.c* con el siguiente código:

```
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
        int retorno, entrada=5, salida=7;

        retorno = newcall(entrada, &salida);

        printf("Resultado newcall:[%d], valor salida:[%d] \n", retorno,salida);

        return 0;
}
```

Al compilar este programa y ejecutar *prueba_newcall*, se llama a la función de la librería *newcall* que envía, a su vez, un mensaje al PM. Es importante notar que no hace falta recompilar el núcleo para poder probar a la nueva llamada al sistema, hasta ahora. Como el PM no sabe qué hacer con un mensaje de ese tipo, el resultado de la ejecución será:

```
# cc prueba_newcall.c -o prueba_newcall
# ./prueba_newcall
PM: in no_sys, call nr 70 from 36691
Resultado newcall: [-1], valor salida: [1234]
```

Pues la llamada a *_syscall()* retornará -1 (error). El 1234 es cargado por la función de la librería directamente. Lo siguiente, entonces, es aportar la funcionalidad requerida al PM.

Paso 3 (Modificación del proceso servidor): El programa principal del Process Manager *(/usr/src/servers/pm/main.c*) es un ciclo infinito en el que el servidor queda a la espera de mensajes, cuando recibe un mensaje lo atiende y luego, eventualmente, envía los resultados a quien realizó la petición. En este procedimiento los privilegios que tienen las tareas Clock y System ya que son los primeros atendidos por este proceso servidor (para denotar esto vea el código completo). La forma en que se realiza la acción a cada tipo de mensaje véala en la Figura 9.13:

```
99    switch(call_nr)
100   {
101   case PM_SETUID_REPLY:
102   case PM_SETGID_REPLY:
103   case PM_SETSID_REPLY:
104   case PM_EXEC_REPLY:
105   case PM_EXIT_REPLY:
106   case PM_CORE_REPLY:
107   case PM_FORK_REPLY:
108   case PM_FORK_NB_REPLY:
109   case PM_UNPAUSE_REPLY:
110   case PM_REBOOT_REPLY:
111   case PM_SETGROUPS_REPLY:
112       if (who_e == FS_PROC_NR)
113       {
114           handle_fs_reply();
115           result= SUSPEND;        /* don't reply */
116       }
117       else
118           result= ENOSYS;
119       break;
120   default:
121       /* Else, if the system call number is valid, perform the
122        * call.
123        */
124       if ((unsigned) call_nr >= NCALLS) {
125           result = ENOSYS;
126       } else {
127#if ENABLE_SYSCALL_STATS
128           calls_stats[call_nr]++;
129#endif
130
131           result = (*call_vec[call_nr])();
132
133       }
134       break;
135   }
```

Figura 9.13 Código fuente de main.c

Como se ve, la función que se ejecuta es aquella que se encuentra en la posición *call_nr* del vector call_vec[]. La variable call_nr, en este contexto, contiene el número de System Call solicitada. *call_vec*

es, entonces, un vector de apuntadores a una función que contiene en la posición n-ésima que debe ejecutarse para atender la System Call número n y se encuentra definido en */usr/src/servers/pm/table.c.* Ahora para que el PM ejecute determinada función ante la llegada de un mensaje requiriendo la System Call 70 hay que cambiar *"no_sys, /* 70 = unused */",* como se ve mostro en la Figura 9.6, por:

```
do_newcall, /* 70 = newcall Nueva llamada al sistema (biblioteca)*/
```

Mediante este cambio, se indica al Process Manager que ejecute la función *do_newcall*. Para programar *do_newcall* debe, en primera instancia, modificarse */usr/src/servers/pm/proto.h* para agregar el prototipo de la función, como se realizó en el ejemplo de la llamada al sistema anterior:

```
_PROTOTYPE( int do_newcall, (void) );
```

Al escribir la función *do_newcall* se debe tener en cuenta que:

- El mensaje de entrada está en *m_in*
- El mensaje de respuesta está en *mp->mp_reply* (mp es un apuntador a la posición de la tabla de procesos ocupada por el proceso que originó la llamada. ver *setreply* en *main.c*)
- Si la *system call* puede ser atendida con los recursos del proceso servidor, *do_newcall* debe ser dotada de toda la funcionalidad necesaria y completar los campos correspondientes del mensaje que se devolverá al proceso que realizó la llamada.
- Si debe requerirse algo a la SYSTASK, debe armarse un nuevo mensaje y enviárselo vía *sys_newcall*, este proceso será explicado posteriormente.
- En */usr/include/minix/ipc.h* se definen los tipos relativos a mensajes.

A continuación, para no generar otro archivo, podemos escribir en la función *do_newcall()* en */usr/src/servers/pm/getset.c:*

```
/*===========================================================*
 *          ******              do_newcall          ******* *
 *===========================================================*/
PUBLIC int do_newcall()
{
  mp->mp_reply.m1_i2 = m_in.m1_i1 * 2;
  return OK;
}
```

Una vez programada la función *do_newcall* es conveniente, a los efectos de simplificar la depuración probar lo hasta ahora lo que se ha realizado. Note que en el ejemplo, *do_newcall* no hace prácticamente nada. Solo retorna en un campo del mensaje de salida, el doble de lo que se le informa en el campo *m1_i1* del mensaje de entrada *m_in*. Una ligera modificación a la función *newcall* (en la línea que esta resaltada puede verse dicha modificación) en */usr/src/lib/posix/_newcall.c* permitirá probar el escenario:

```
#include <lib.h>
#include <unistd.h>

PUBLIC long newcall(int entrada, int * salida)
{
 message m;
 int retorno;

 /* Se establece los campos del mensaje a enviar */
 m.m1_i1 = entrada;

 /* La forma de pasar un mensaje a MM es a través de:
  _syscall(MM,NEWCALL,&m);
 El mensaje de respuesta queda en m.
 */
 retorno=_syscall(MM,NEWCALL,&m);

 /* Retornar la información contenida en el mensaje vía parametros*/
 *salida = m.m1_i2;

 return(retorno);
}
```

Se tiene hasta aquí una llamada al sistema que permite calcular en el PM el doble del primer valor pasado como argumento. Ahora debe compilar las librerías vía, el PM y el núcleo:

```
# cd /usr/src
# make libraries
# cd /usr/src/servers/pm
# make
# cd /usr/src/tools
# hdboot
```

Subsiguientemente, realice el paso explicado como Generar una versión. Luego de reiniciar se ejecuta nuevamente prueba_newcall que dará:

```
# cc prueba_newcall.c -o prueba_newcall
# ./prueba_newcall
Resultado newcall: [0], valor salida: [10]
```

Para este punto, se habrá probado que:

- El proceso de usuario puede enviar información al PM vía un mensaje.
- Que el PM puede desempaquetar la información contenida en el mensaje y hacer la tarea necesaria para realizar la nueva system call.
- Que el PM puede enviar información de respuesta al proceso que realizó la llamada al sistema.

Es decir, se ha logrado construir una nueva llamada al sistema y se ha trasmitido información entre las capas 3 y 4 de MINIX. La funcionalidad de la llamada creada es nula, pero el mecanismo a utilizar para escribir cualquier nueva llamada al sistema, será muy similar al descrito.

9.2.5 Llamada al sistema (extendida)

En el caso en que el proceso servidor no tenga todos los recursos necesarios para satisfacer la llamada al sistema, por ejemplo, porque se necesita acceder a información que está en el espacio de direcciones del núcleo, deberá solicitar a un ente externe que realice la tarea por él.

En este ejemplo, se asume que la SYSTASK realizará la tarea, siendo especifico, la multiplicación por tres del primer valor pasado como parámetro a *newcall*, se usará otro factor para distinguir los resultados de la llamada al sistema sencilla. Este esquema de llamada al sistema puede observarse en Figura 9.14. Es importante notar que esta función hace una invoca a *_taskcall*, para poder enviar un mensaje a núcleo. También es importante señalar que un proceso a nivel de usuario no puede realizar este tipo de llamadas debido a que el SYSTASK se encuentra en la capa 1, y los procesos de usuario solo envían mensajes a la capa 3. Hay que resaltar que una llamada al sistema que implique un cambio de modo es más costosa a nivel de recursos que las demás.

Además, implica otros temas como seguridad, si una llamada a nivel de usuario no está semánticamente bien hecha puede afectar los Procesos Servidores. Sin embargo, para evitar la caída de los mismos existe un proceso especial Proceso Servidor Reencarnación. Dicho servidor de forma periódica envía consultas a cada controlador de dispositivo y a los procesos servidores, como el PM, FS, etc. Si el controlador de dispositivo o proceso servidor muere o no responde correctamente a las consultas, el servidor de reencarnación automáticamente los sustituye por una copia nueva. La detección y el reemplazo de los controladores de dispositivo procesos servidores que no funcionan son de forma automática, sin intervención del usuario. Este mecanismo intenta brindarle a Minix 3 tolerancia a una caída de los controladores o procesos servidores (15).

A diferencia de una llamada al sistema que implique un cambio de modo, el servidor reencarnación no tiene el alcance ni el mecanismo para poder brindar la tolerancia o robustez al sistema operativo. Esta es una razón para tener precaución a la hora de implementar una llamada al sistema que implique un cambio de modo.

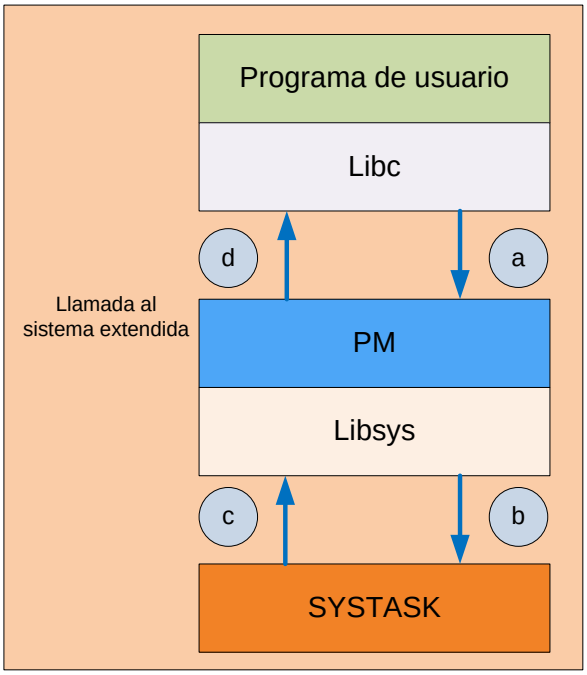


Figura 9.14 Llamada al sistema (extendida)

La función *do_newcall*, mencionada anteriormente, debe ahora armar un nuevo mensaje y enviarlo a la SYSTASK. Como en el caso del programa de usuario, puede programarse al PM para:

- Enviar directamente el mensaje utilizando la llamada a *_syscall*
- Crear una función de biblioteca que oculte todo el manejo de mensajes.

Se optará por la segunda opción agregando una función a la biblioteca *libsys*. Estos pasos se explicaran a continuación.

Paso 1 (Construcción de la función de biblioteca): En */usr/src/lib/syslib/* se creará el archivo *sys_newcall.c* con el código necesario para invocar a la SYSTASK por medio de un mensaje. Para lograrlo utiliza la función *_taskcall* (en la línea que esta resaltada), la cual fue explicada con anterioridad. El cual tendrá el siguiente contenido:

```
#include "syslib.h"

int sys_newcall(int entrada, int *salida)
{
 message m;
 int retorno;

 /* Se establece los campos del mensaje a enviar */
 m.m1_i1 = entrada;

            /* La forma de pasar un mensaje a la SYSTASK es a través de:
  _taskcall(SYSTASK,SYS_NEWCALL,&m);
  El mensaje de respuesta queda en m.
  */
  retorno = _taskcall(SYSTASK,SYS_NEWCALL,&m);

 /* Retorna la información contenida en el mensaje vía parametros*/

 *salida=m.m1_i2;
 return(retorno);
}
```

Editar el archivo Makefile.in en el mismo directorio y agregar a sys_newcall.c en la lista de archivos que componen la biblioteca:

```
# Makefile for lib/syslib.

CFLAGS="-O -D_MINIX -D_POSIX_SOURCE"

LIBRARIES=libsys

libsys_FILES=" \
            alloc_util.c \
            assert.c \
            sys_newcall.c \
            panic.c \
            pci_attr_r16.c \
            pci_attr_r32.c \
```

Hacer el Makefile ejecutando:

```
# cd /usr/src/lib/syslib/
# make Makefile
```

Luego, en */usr/src/include/minix/syslib.h* agregar el prototipo de la función:

```
_PROTOTYPE( int sys_newcall, (int entrada, int *salida) );
```

Activar los nuevos archivos de cabecera:

```
# cd /usr/src/
# make includes
```

Ahora se procederá a modificar el funcionamiento de la SYSTASK para que responda a la nueva petición.

Paso 2 (Modificación de la SYSTASK): Debe en primera instancia, agregar un procedimiento más al vector de apuntadores a funciones que utiliza la SYSTASK para ejecutar las funciones que atienden a cada system call. Se comenzará cambiando en */usr/src/include/minix/com.h* la cantidad máxima de llamadas soportada por el núcleo (NR_SYS_CALLS) y definiendo la macro SYS_NEWCALL que se utilizó en la llamada a _taskcall. Vea primero parte del archivo:

```
354#  define SYS_RUNCTL      (KERNEL_CALL + 46) /* sys_runctl() */
355#  define SYS_SAFEMAP     (KERNEL_CALL + 47) /* sys_safemap() */
356#  define SYS_SAFEREVMAP  (KERNEL_CALL + 48) /* sys_saferevmap() sys_saferevmap2() */
357#  define SYS_SAFEUNMAP   (KERNEL_CALL + 49) /* sys_safeunmap() */
358
359
360#define NR_SYS_CALLS    50  /* number of system calls */
361#define SYS_CALL_MASK_SIZE BITMAP_CHUNKS(NR_SYS_CALLS)
362
```

Lo que se debe hacer es agregar syscall al final del vector, y actualizar el valor de NR_SYS_CALL. Las modificaciones están resaltadas en el siguiente código. El archivo debe quedar de la siguiente forma:

```
# define SYS_RUNCTL (KERNEL_CALL + 46)  /* sys_runctl() */
# define SYS_SAFEMAP (KERNEL_CALL + 47)          /* sys_safemap() */
# define SYS_SAFEREVMAP (KERNEL_CALL + 48)       /* sys_safere sys_saferevmap2*/
# define SYS_SAFEUNMAP (KERNEL_CALL + 49)        /* sys_safeunmap() */
# define SYS_NEWCALL  (KERNEL_CALL + 50)         /* sys_newcall() */


#define NR_SYS_CALLS            51          /* number of system calls */
#define SYS_CALL_MASK_SIZE BITMAP_CHUNKS(NR_SYS_CALLS
```

En la función *initialize* en */usr/src/kernel/system.c* mediante llamadas a *map* carga en cada posición del vector call_vec[] un apuntador a la función que debe ejecutarse ante un mensaje de tipo n (n es el número de llamadas). Entonces, debe agregarse una llamada a *map* para cargar la dirección correspondiente a la nueva llamada:

```
map(SYS_NEWCALL, do_newcall);  /* nueva llamada al sistema */
```

Los cambios se muestran en la siguiente figura, específicamente en la línea que esta resaltada:

```
 /*===================================*
 *                initialize                 *
 *===================================*/
PRIVATE void initialize(void)
{
 register struct priv *sp;
 int i;

 /* Initialize IRQ handler hooks. Mark all hooks available. */
 for (i=0; i<NR_IRQ_HOOKS; i++) {
 irq_hooks[i].proc_nr_e = NONE;
 }

 /* Initialize all alarm timers for all processes. */
 for (sp=BEG_PRIV_ADDR; sp < END_PRIV_ADDR; sp++) {
  tmr_inittimer(&(sp->s_alarm_timer));
 }
/* Initialize the call vector to a safe default handler. Some system calls
  * may be disabled or nonexistant. Then explicitely map known calls to their
  * handler functions. This is done with a macro that gives a compile error
  * if an illegal call number is used. The ordering is not important here.
  */
 for (i=0; i<NR_SYS_CALLS; i++) {
 call_vec[i] = do_unused;
 callnames[i] = "unused";
 }

 map(SYS_NEWCALL, do_newcall);  /* nueva llamada al sistema */
```

En /usr/src/kernel/system.c, puede escribirse la función que hace el trabajo solicitado, en este caso retorna el mensaje obtenido multiplicado por 3 (tres). Esta función es denominada do_newcall, vea su implementación a continuación:

```
/*=========================================*
 *              do_newcall                 *
 *=========================================*/
int do_newcall(m_ptr)
register message *m_ptr;
{
 m_ptr->m1_i2= m_ptr->m1_i1 * 3;
 return(0);
}
```

Ahora debe escribirse el prototipo al comienzo de /usr/src/kernel/system.c, antes del map():

```
_PROTOTYPE( int do_newcall, (message *m_ptr) );
```

Luego de esto, solo falta compilar las bibliotecas y el nuevo núcleo. Para ello:

```
# cd /usr/src/
# make includes
# make libraries
# cd /usr/src/tools
# make hdboot
```

Paso 3 (Prueba 1 - Programa de usuario llama a system call): Se crea un nuevo archivo llamado *prueba_newcall_task.c,* el cual utiliza la función sys_newcall ¿Qué pasa si invocamos a la llamada al sistema desde el programa de prueba, que corre con permisos de usuario? Compilamos el programa de pruebas, esta vez su código en *prueba_newcall_task.c* será (21):

```
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <minix/syslib.h>

int main(int argc, char *argv[])
{
 int retorno, entrada=5, salida=7;

 retorno = sys_newcall(entrada, &salida);

 printf("Resultado newcall:[%d], valor salida:[%d]\n", retorno, salida);

 return(0);
}
```

Se compila y se enlaza con la biblioteca del sistema. Luego podemos correr a prueba_newcall y ver el resultado en pantalla. Lo que sucede es que se obtiene un mensaje de error en la consola, y el proceso es terminado por el SO. El mensaje es:

```
# cc -o prueba_newcall_task prueba_newcall_task.c -lsys
# ./prueba_newcall_task
sys_call: ipc mask denied trap 3 from 36727 to -2
sys_call: ipc mask denied trap 3 from 36727 to -2
```

El 36730 es el PID del proceso creado al ejecutar el programa prueba_newcall. El mensaje dice que se denegó el permiso para hacer IPC desde el proceso de usuario al servidor, arrojando el número de error -2 (el RS_PROC_NR).

Paso 2 (Prueba 2 - Programa de usuario llama a system call): Para poder utilizar realmente lo implementado debe modificar la función do_newcall en el archivo /usr/src/servers/pm/getset.c de la siguiente manera para sea el PM el que se comunique con el SYSTASK (21).

```
PUBLIC int do_newcall()
{
        int retorno, input, output;
        input = m_in.m1_i1;


        retorno = sys_newcall(input, &output);
        mp->mp_reply.m1_i2 = output;

 return OK;
}
```

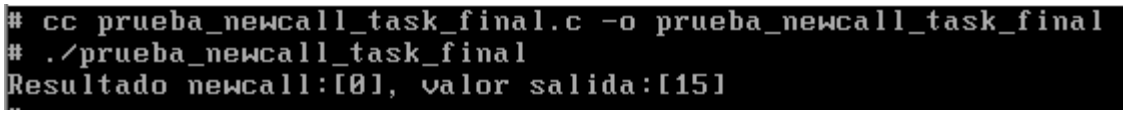
Luego de esto, solo falta compilar las bibliotecas y el nuevo núcleo. Para ello:

```
# cd /usr/src/
# make libraries

# cd /usr/src/tools
# make hdboot
```

Luego implementamos el programa de pruebas, recordemos su código en *prueba_newcall_task_final.c:*

```
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
 int retorno, entrada=5, salida=7;

 retorno = newcall(entrada, &salida);

 printf("Resultado newcall:[%d], valor salida:[%d] \n", retorno, salida);

 return 0;
}
```

Para finalizar compilamos y ejecutamos el código y obtenemos los valores esperados, como se muestra a continuación.



10 Implementación de semáforos

El concepto de proceso es fundamental en la estructura de los SO. Cada proceso tiene asociado un espacio de direcciones, una lista de posiciones de memoria desde algún mínimo hasta algún máximo, que el proceso puede leer y escribir. El espacio de direcciones contiene el programa ejecutable, los datos del programa, y su pila. A cada proceso también se asocia un conjunto de registros, que incluyen el contador del programa, el apuntador de la pila y otros registros de hardware, así como todo la demás información necesaria para ejecutar el programa. Este término de proceso tiene muchas definiciones en las cuales tenemos (2):

- Un programa en ejecución, que conceptualmente tiene su CPU virtual.
- Una instancia de un programa ejecutándose en un procesador.
- La entidad que se puede asignar o ejecutar en un procesador.
- Una unidad de actividad caracterizada por un solo hilo secuencial de ejecución, un estado actual, y un conjunto de recursos del sistema asociados.

Cada proceso tiene las siguientes dos características (2):

- Propiedad de recursos: Un proceso incluye un espacio de direcciones virtuales para el manejo de la imagen del proceso; la imagen de un proceso es la colección de programa, datos, pila y atributos definidos en el bloque de control del proceso. En ciertas ocasiones un proceso se le puede asignar control o propiedad de recursos tales como la memoria principal, dispositivos E/S y archivos. El sistema operativo realiza la función de protección para evitar interferencias no deseadas entre procesos en relación con los recursos.
- Planificación/ejecución: Un proceso tiene un estado de ejecución y una prioridad de activación.

En la mayor parte de los sistemas operativos tradicionales, estas dos características son, realmente, la esencia de un proceso. Sin embargo, debe quedar muy claro que estas dos características son independientes y podrían ser tratadas como tales por el sistema operativo.

10.1 *Secuencia de inicialización del árbol de procesos en Minix 3*

Los procesos en MINIX siguen el modelo general de procesos que se describió con anterioridad, también procesos pueden crear subprocesos, que a su vez pueden crear más subprocesos, produciendo un árbol de procesos. De hecho, todos los procesos de usuario del sistema forman parte de un solo árbol con init ver la. Los servidores y controladores son un caso especial, por supuesto, ellos deberían ser inicializados antes que cualquier proceso de usuario, incluyendo init (15).

¿Cómo se forma este árbol? Una vez cargado el sistema operativo, como se explico en el capítulo 2, siguiendo la secuencia de ejecución del *masterboot*, *bootblock* y *boot* monitor. Este último busca un

archivo multiparte llamado *boot* image. El cual contiene la parte más importante del núcleo (la tarea del reloj y la tarea del sistema), el manejador de procesos y el sistema de archivos. Adicionalmente, también deberían estar incluidos algunos controladores. Además, hay varios programas incluidos en la *boot* image, estos son: el servidor reencarnación, la consola, el disco RAM e init.

Durante su fase de inicialización, el núcleo inicia las tarea del reloj y la tarea del sistema, luego el manejador de procesos y el sistema de archivos. Posteriormente, el manejador de procesos y el sistema de archivos cooperan para inicializar cualquier otro servidor o controlador que son parte de la *boot* image. Una vez que todos éstos se han ejecutado e inicializado a sí mismos, se bloquean, esperando algo que hacer. Cuando todas las tareas y servidores están bloqueados, se ejecuta init, el cual es el primer proceso de usuario. Este proceso ya está en la memoria principal, pero desde luego podría haberse cargado del disco como programa aparte, ya que todo está funcionando para cuando se inicia. Sin embargo, dado que init se inicia sólo esta única vez y nunca se vuelve a cargar del disco, lo más fácil es incluirlo en el archivo de imagen del sistema junto con el núcleo, las tareas y los servidores. Los componentes del sistema cargados en la *boot* image o durante la inicialización se muestran en la Tabla 10.1 (15).

Componente	Descripción	Cargado por
núcleo	Núcleo + las tareas del reloj y del sistema	En la *boot* image
pm	Manejador de procesos	En la *boot* image
fs	Sistema de archivos	En la *boot* image
rs	Servidor reencarnación (inicio y controladores)	En la *boot* image
memory	RAM controlador de disco	En la *boot* image
log	Registro de salida del bufer	En la *boot* image
tty	Controlador de consola y teclado	En la *boot* image
driver	Controlador de disco	En la *boot* image
init	Padre de todos los procesos de usuario	En la *boot* image
floppy	Controlador del disquete	/etc/rc
is	Servidor de información (la depuración)	/etc/rc
cmos	lee reloj CMOS para ajustar la hora	/etc/rc
random	Generador de números aleatorios	/etc/rc
printer	Controlador de impresora	/etc/rc
Nota: otros componentes como el controlador de Ethernet y el servidor inet pueden estar presentes en la boot image.		

Tabla 10.1 Componentes de Minix 3

10.2 *Comunicación entre proceso en Minix 3*

Minix 3 utiliza el paso de mensajes para la comunicación, en general, el paso de mensajes proporciona un par de primitivas. Las cuales permiten la comunicación y sincronización entre procesos. Las primitivas puede observarlas a continuación:

- send(destino, mensaje)
- receive(origen, mensaje)

Este es el conjunto mínimo de operaciones necesarias para que los procesos puedan entablar paso de mensajes. Un proceso envía información en forma de un mensaje a otro proceso designado por destino. El proceso recibe información ejecutando la primitiva receive indicando la fuente y el mensaje. Esta llamada podría ser bloqueante, esto quiere decir que el proceso puede bloquearse hasta recibir un mensaje.

10.2.1 Mecanismo de paso de mensajes en Minix 3

Debe recordar, primero que el diseño de Minix 3 es de micronúcleo. Segundo que Minix 3 está diseñado en capas, y que la principal diferencia entre las mismas radica en la posibilidad de realizar llamadas al núcleo siendo los de la capa 2 los más privilegiados, seguidos en orden decreciente por la capa 3 y 4. En cuanto al paso de mensajes, las capas 2 y 3 son las únicas que pueden comunicarse con el núcleo. También el paso de mensajes entre procesos de usuarios (capa 4) no es posible. En esta figura puede observarse el flujo de mensajes entre las diferentes capas y actores de Minix 3. Los procesos de usuario (capa 4) solo pueden enviar y recibir mensajes de la capa 3. Los procesos servidores (capa 3) pueden comunicarse con las capas 2 y 1. Los controladores (capa 2) se comunican directamente con el núcleo como los procesos servidores, sin embargo, los de esta capa tienen más prioridad (15). Para entender mejor la situación vea la Figura 10.1.

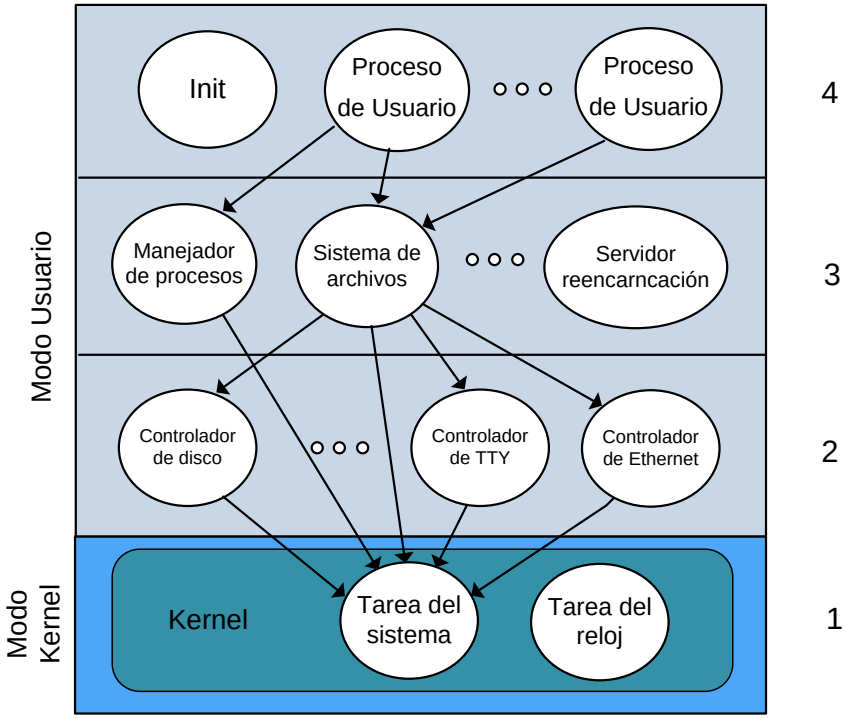


Figura 10.1 Flujo de mensajes en Minix 3

10.3 *Sincronización de procesos de usuario en Minix 3*

El único método para sincronizar los procesos en Minix 3 es a través de paso de mensajes. Sin embargo, debido al mecanismo de paso de mensajes surge un problema para sincronizar procesos de la capa 4. Se requiere proporcionar una solución que permita la sincronización entre procesos de usuarios. Para ello se propone la implementación de semáforos en Minix 3, usando como soporte el pase de mensajes. Antes de describir esta solución conozca la estructura y todo lo referente a un semáforo.

El primer avance fundamental en el tratamiento de los problemas de programación concurrente ocurre en 1965 con el tratado de Dijkstra. Dijkstra estaba involucrado en el diseño de un sistema operativo como una colección de procesos secuenciales cooperantes y con el desarrollo de mecanismos eficientes y fiables para dar soporte a la cooperación. Estos mecanismos podrían ser usados fácilmente por los procesos de usuario si el procesador y el sistema operativo colaborasen en hacerlos disponibles.

El principio fundamental es éste: dos o más procesos pueden cooperar por medio de simples señales, tales que un proceso pueda ser obligado a parar en un lugar específico hasta que haya recibido una señal específica. Cualquier requisito complejo de coordinación puede ser satisfecho con la estructura de señales apropiada. Para la señalización, se utilizan unas variables especiales llamadas semáforos. Para transmitir una señal vía el semáforo s, el proceso ejecutará la primitiva semSignal(s). Para recibir una señal vía el semáforo s, el proceso ejecutará la primitiva semWait(s); si la correspondiente señal no se ha

transmitido todavía, el proceso se suspenderá hasta que la transmisión tenga lugar. Para conseguir el efecto deseado, el semáforo puede ser visto como una variable que contiene un valor entero sobre el cual sólo están definidas tres operaciones (2):

- Un semáforo puede ser inicializado a un valor no negativo.
- La operación semWait decrementa el valor del semáforo. Si el valor pasa a ser negativo, entonces el proceso que está ejecutando semWait se bloquea. En otro caso, el proceso continúa su ejecución.
- La operación semSignal incrementa el valor del semáforo. Si el valor es menor o igual que cero, entonces se desbloquea uno de los procesos bloqueados en la operación semWait.

10.3.1 Semáforos en Minix 3

Lo anterior descrito es lo que se desea implementar en Minix 3. Pero ¿cómo la funcionalidad del semáforo se puede agregar a MINIX 3? Los semáforos son números enteros, los cuales se inicializan igual o mayor que cero. Y son modificados mediante dos operaciones, semSignal y semWait, para sincronizar múltiples procesos intentan que acceder a un recurso compartido. Una operación semWait(S) decrementa el semáforo S. Si S es menor que cero, el proceso se bloquea la llamada hasta que algún otro proceso incrementa a S a través de una operación semSignal(S). Esta funcionalidad es normalmente parte del núcleo de un sistema monolítico, pero puede ser realizado como un servidor independiente del espacio de usuario en MINIX 3.

Para la implementación de la solución se requiere de un proceso servidor. La estructura del servidor semáforo MINIX 3 se muestra en la Figura 10.2. Después de la inicialización, el servidor entra en un bucle principal sin fin. En cada iteración el servidor se bloquea y espera hasta que llega un mensaje de solicitud. Una vez que un mensaje ha sido recibido, el servidor examina la solicitud. Si el tipo es conocido, la función manejadora asociada a está llamada procesa la solicitud, y el resultado se devuelve; a menos que el proceso que llama deba ser bloqueado. Si se recibe tipos ilegales de solicitud directamente el resultado debe indicar que es una solicitud errónea.

Un punto importante es que los procesos de usuario en Minix 3 se limitan al paso de mensajes síncrono. Esto quiere decir que, luego de realizar la solicitud el proceso invocador se bloqueará hasta que la respuesta haya llegado. De esta característica se puede obtener mucha ventaja a la hora de implementar semáforos en Minix 3, sobre todo cuando se construye el servidor de semáforos. Para las operaciones semSignal, el servidor simplemente incrementa el semáforo y directamente envía una respuesta para que el proceso que realiza la solicitud pueda continuar. Para las operaciones semWait, por el contrario, la respuesta es retenida, el semáforo se decrementa, bloqueando efectivamente al proceso que llama hasta que se retorne el mensaje de respuesta (22).

```
void semaphore_server() {
        message m;
        int result;
        /* Inicializa al Servidor Semáforo. */
        initialize( );
        /* Ciclo principal del servidor. Obtiene trabajo y lo procesa. */
        while(TRUE) {
                /* Se bloquea hasta que un mensaje de petición llega. */
                ipc_receive(&m);
                /* El proceso que envío el mensaje esta bloqueado.
                 * Despacho según el tipo de mensaje. */
                switch(m.m_type) {
                        case semSignal: result = do_semSignal (&m); break;
                        case semWait: result = do_semWait (&m); break;
                        default: result = ERROR;
                }
                /* Enviar respuesta, a menos que el solicitante debe estar bloqueado. */
                if (result != EDONTREPLY) {
                        m.m_type = result;
                        ipc_reply(m.m_source, &m);
                }
        }
}
```

Figura 10.2 Estructura del servidor semáforo

El semáforo tiene asociado una cola FIFO (First In, First Out) de procesos para realizar un seguimiento de los procesos que están bloqueados. Después de una operación semSignal, la cola se comprueba para ver si un proceso espera por ser desbloqueado (22).

Todos los servidores y los controladores tienen un bucle principal similar. La función *initialize*() se llama sólo una vez y antes de entrar en el bucle principal, dicha función no será explicada aquí. Las funciones manejadoras *do_semSignal* y *do_semWait* se muestran a continuación.

Con la estructura del servidor de semáforos implementada, es necesario proveer a los procesos de usuario la posibilidad de comunicarse con él. Una vez que el servidor se ha iniciado está listo para recibir peticiones. En principio, el programador puede construir mensajes de solicitud y enviarlo al servidor nuevo mediante *ipc_request*, pero esos detalles suelen ser convenientemente escondidos en las bibliotecas del sistema, junto con las otras funciones POSIX, como se explicó en el capitula anterior.

```
int do_semWait (message *m_ptr) {
        /* Decremento al semáforo y retorna la respuesta. */
        s = s – 1; /* take a resource */

        if (s > 0) {
                return(OK); /* let the caller continue */
        }
        /* Encola al proceso solicitante y lo bloquea. */
        enqueue(m_ptr->m_source); /* lo agrega a la cola */
        return(DONTREPLY); /* indica que no retorne respuesta */
}
```

Figura 10.3 Implementación de do_semWait

Como puede observarse en la Figura 10.3 está la implementación de la función semWait, la cual decrementa el valor del semáforo s y verifica si esta variable queda negativa encola al proceso invocante y lo bloquea. En la Figura 10.4 se muestra la implementación de la función semSignal, la cual incrementa el semáforo s; después verifica si existen procesos bloqueados, en caso afirmativo bloquea al proceso que lleva más tiempo bloqueado.

```
int do_semSignal (message *m_ptr) {
        message m; /* Declaracion del mensaje a reponder */
        /* Agregar respuesta y desbloquear a un proceso
         * si es necesario. */
        s = s + 1; /* incrementa al semaforo */
        /* Chequea si un proceso está bloqueado por el semaforo. */
        if (queue_size() > 0) { /* pregunta si hay procesos encolados? */
                m.m_type = OK;
                m.m_source = dequeue(); /* lo elimina de la cola */
                ipc_reply(m.m_source, m); /* coloca la respueta */
        }
        return(OK);
}
```

Figura 10.4 Implementación de do_semSignal

Por lo general, las llamadas a semSignal y semWait están declaradas en una nueva biblioteca que permitiría manejar estas llamadas. Esto se explicó en el capítulo anterior. La estructura modular de MINIX 3 ayuda a acelerar el desarrollo la implementación de semáforos de varias maneras. En primer lugar, se puede implementar de forma independiente del resto del sistema operativo, al igual que las aplicaciones de usuario normal. Cuando haya finalizado, puede ser compilado como una aplicación independiente y de forma dinámica comenzó a formar parte del sistema operativo. No es necesario construir un nuevo núcleo (22).

10.4 *Servidor PM*

Antes de implementar la solución se debe conocer el funcionamiento del servidor PM, debido a que el mismo se va a utilizar como soporte a la misma. El servidor PM tiene similar a la ilustrada como *semaphore_server* en la Figura 10.2. La función *main* del servidor PM se muestra en la Figura 10.5 y empieza con una inicialización principal *sef_local_startup*(). Después, el servidor entra en un bucle principal sin fin.

En cada iteración el servidor se bloquea y espera hasta que llega un mensaje de solicitud, para esto utiliza la función *get_work().* Una vez que un mensaje ha sido recibido, el servidor examina la solicitud, a través del *switch(call_nr)* para determina el tipo de llamada. Si el tipo es conocido y la llamada está definida en /usr/src/servers/pm/table.c, el servidor busca la función mapeada a la llamada a través de la instrucción *result = (*call_vec[call_nr])().* La función manejadora asociada a está llamada procesa la solicitud, y devuelve el resultado. Si se recibe tipos ilegales de solicitud directamente el resultado debe

indicar que es una solicitud errónea, para este caso el resultado como se explicó en el capítulo anterior retorna -1. La porción de código que se encarga de retornar la respuesta la función main del servidor PM comienza con la etiqueta *send_reply*, y se puede observar en la Figura 10.6, la cual es una continuación de la función main. Para poder retornar el mensaje el servidor PM verifica si *result != SUSPEND*, cuando se cumpla esta condición el mensaje es devuelto al proceso invocador, para que el mismo se desbloquee.

El proceso antes mencionado ocurre siempre que el sistema operativo este funcionado. En caso de que ocurra un error, existe el servidor RS (Servidor Reencarnación). El RS trabaja para reiniciar los controladores y procesos servidores. Este servidor inicia bloqueado y espera a la llegada de un mensaje que indica le indique que va a crear. En el proceso de arranque el proceso init ejecuta un script que emite un comando al RS para que inicie los controladores y servidores que no están presentes en la *boot image*, entonces inician como procesos hijos des servidor RS. En consecuencia, si alguno de estos procesos falla o terminan el RS será informado y se encargará de restaurarlos.

Este mecanismo es un intento para permitirle a Minix 3 la tolerancia a fallos de los controladores y servidores debido a que uno nuevo de estos siempre se crearan.

```
/*===========================================================*
 *                          main                             *
 *===========================================================*/
PUBLIC int main()
{
/* Main routine of the process manager. */
 int result, s, proc_nr;
 struct mproc *rmp;
 sigset_t sigset;
 /* SEF local startup. */
 sef_local_startup();

 /* This is PM's main loop-get work and do it, forever and forever. */
 while (TRUE) {
                get_work();                     /* wait for an PM system call */
                switch(call_nr)
                {
                case PM_SETUID_REPLY:
                case PM_SETGID_REPLY:
                case PM_SETSID_REPLY:
                case PM_EXEC_REPLY:
                case PM_EXIT_REPLY:
                case PM_CORE_REPLY:
                case PM_FORK_REPLY:
                case PM_FORK_NB_REPLY:
                case PM_UNPAUSE_REPLY:
                case PM_REBOOT_REPLY:
                case PM_SETGROUPS_REPLY:
                                if (who_e == FS_PROC_NR)
                                {
                                                handle_fs_reply();
                                                result= SUSPEND;                /* don't reply */
                                }
                                else
                                                result= ENOSYS;
                                break;
                default:
                /* Else, if the system call number is valid, perform the call. */
ir_a_call_vec:
                                if ((unsigned) call_nr >= NCALLS) {
                                                result = ENOSYS;
                                } else {
#if ENABLE_SYSCALL_STATS
                                                calls_stats[call_nr]++;
#endif

                                                result = (*call_vec[call_nr])();

                                }
                                break;
                }

send_reply:
```

Figura 10.5 Código fuente de main.c

```
send_reply:
        /* Send the results back to the user to indicate completion. */
        if (result != SUSPEND)setreply(who_p, result);

        /* Send out all pending reply messages, including the answer to
         * the call just made above.
         */
        for (proc_nr=0, rmp=mproc; proc_nr < NR_PROCS; proc_nr++, rmp++) {
                /* In the meantime, the process may have been killed by a
                 * signal (e.g. if a lethal pending signal was unblocked)
                 * without the PM realizing it. If the slot is no longer in
                 * use or the process is exiting, don't try to reply.
                 */
                if ((rmp->mp_flags & (REPLY | IN_USE | EXITING)) ==
                 (REPLY | IN_USE)) {
                                s=sendnb(rmp->mp_endpoint, &rmp->mp_reply);
                        if (s != OK) {
                                printf("PM can't reply to %d (%s): %d\n",
                                        rmp->mp_endpoint, rmp->mp_name, s);
                        }
                        rmp->mp_flags &= ~REPLY;
                }
        }
 }
 return(OK);
}
```

Figura 10.6 Código fuente de main.c (continuación)

10.5 *Implementación de semáforos Minix 3*

Una vez trazado un bosquejo de la solución para la implementación de semáforos en Minix 3, se procederá a la explicación de la misma. Esta solución sigue el patrón de la solución explicada en el punto anterior, sin embargo, no se implemento un nuevo servidor para el manejo de los mensajes. En contraste, se utilizo el proceso servidor PM, para la recepción y transmisión de los mensajes con los procesos de usuario. Cabe destacar que para la implementación de semáforos se implementaron las llamadas *sem_signal, sem_wait* y *sem_init* para la comunicación entre los procesos de usuario y el servidor PM. En este momento se procederá a describir la solución de la implementación de semáforos en Minix 3.

Paso 1 (Implementación de las llamadas al sistema): se debe implementar las llamadas al sistema que permiten la comunicación entre los procesos de usuario y el servidor PM, tal y como se explicó el capítulo anterior. Dicha implementación se llevo a cabo en el archivo */usr/src/lib/posix/_semcall.c.*

Para esto se implementará como se menciono anteriormente las funciones sem_signal, sem_wait y *sem_init*, las cuales reciben un entero semáforo que indica cual es el semáforo referenciado. En el caso de *sem_init* también recibe un entero valor que indica el valor con el cual será inicializado el semáforo, como se sabe el valor debe ser positivo. Todos estos parámetros se transfieren a cada llamada en particular a través del mensaje m declarado.

```
#include <lib.h>
#include <unistd.h>
#include <stdio.h>


PUBLIC long sem_signal (int semaforo)
{
 message m;

 m.m1_i1=semaforo; /* pasa por parametro el semaforo asociado */
 return _syscall(MM, SEM_SIGNAL, &m);
}

PUBLIC long sem_wait (int semaforo)
{
 message m;

 m.m1_i1=semaforo;  /* pasa por parametro el semaforo asociado */
 return _syscall(MM, SEM_WAIT, &m);
}

PUBLIC long sem_init (int semaforo, int valor)
{
 message m;

 if(valor >= 0){

                m.m1_i1=semaforo;  /* pasa por parametro el semaforo asociado */
                m.m1_i2=valor;        /* pasa el valor de inicio del semaforo */
                return _syscall(MM, SEM_INIT, &m);
 }else{
                printf("el valor de inicio debe ser mayor o igual a 0\n");
 }
}
```

Evidentemente, se modifico el archivo /usr/src/include/minix/callnr.h con las entradas a los macros como se muestra, No se hará mucho énfasis entre punto debido a que se explico en el capítulo anterior.

```
/********* SEMAFOROS **********/
#define SEM_SIGNAL 67
#define SEM_WAIT 68
#define SEM_INIT 69
```

Paso 2 (Modificación de PM): Como se mencionó con anterioridad para la implementación de semáforos en Minix 3 no se va a crear un nuevo proceso servidor, en cambio se va a modificar el servidor PM. Es un sutil cambio pero vital, el cual será mostrado a continuación. Solamente se agregó una entrada al switch con el case 68. Para que a la hora de bloquear a un proceso, pueda ser determinado mediante el parámetro que se está pasando a través de la variable m_in.m1_i2. Luego de esto se indica que debe realizar un salto a la nueva etiqueta creada como ir_a_call_vec. Los cambios pueden verse resaltados con el color amarillo.

```
	switch(call_nr)
	{
	case 68:
		m_in.m1_i2=who_p;
		goto ir_a_call_vec;

	case PM_SETUID_REPLY:
	case PM_SETGID_REPLY:
	case PM_SETSID_REPLY:
	case PM_EXEC_REPLY:
	case PM_EXIT_REPLY:
	case PM_CORE_REPLY:
	case PM_FORK_REPLY:
	case PM_FORK_NB_REPLY:
	case PM_UNPAUSE_REPLY:
	case PM_REBOOT_REPLY:
	case PM_SETGROUPS_REPLY:
		if (who_e == FS_PROC_NR)
		{
			handle_fs_reply();
			result= SUSPEND;		/* don't reply */
		}
		else
			result= ENOSYS;
		break;
	default:
		/* Else, if the system call number is valid, perform the call */
ir_a_call_vec:
		if ((unsigned) call_nr >= NCALLS) {
			result = ENOSYS;
		} else {
#if ENABLE_SYSCALL_STATS
			calls_stats[call_nr]++;
#endif

			result = (*call_vec[call_nr])();

		}
		break;
	}
```

Paso 3 (Implementación de semáforo): Para la implementación se modificó el archivo /usr/src/servers/pm /getset.c. En primer lugar se implemento la estructura cola para el manejo de procesos. Esta estructura de datos permite simular una cola FIFO, la cual es de valiosa importancia para la implementación de semáforos. Además, se implemento las funciones encolar y desencolar para manipulación de la cola. La cual puede observarse en la Figura 10.7.

```
/*=======================================================*
 *                    Parte de semaforos / cola                    *
 ========================================================*/

/************** Varibles ****************************/

#define Max_Sem 10 /* numero de semaforos */
#define N 10 /* capacidad de la cola fifo */

int Sem[Max_Sem];

struct cola{
		/* el elemento 0 n es usado */
		int fifo[1+N];
		int w_idx;
		int r_idx;
}FIFO[10];

int next_index(int idx){ return --idx ? idx: N; }

int fifo_empty(struct cola *c_fifo)
{
		return c_fifo->w_idx == c_fifo->r_idx;
}

int fifo_full(struct cola *c_fifo)
{
		return next_index( c_fifo->w_idx) == c_fifo->r_idx;
}
int fifo_encolar(struct cola *c_fifo, int ch)
{
 if( fifo_full(c_fifo) ) return 0;
 c_fifo->w_idx = next_index( c_fifo->w_idx);
 c_fifo->fifo[c_fifo->w_idx] = ch;

 return 1;
}

int fifo_desencolar(struct cola *c_fifo, int *ch)
{
 if( fifo_empty(c_fifo) ) return 0;

 c_fifo->r_idx = next_index(c_fifo->r_idx);
 *ch = c_fifo->fifo[c_fifo->r_idx];

 return 1;
}
```

Figura 10.7 Implementación de la estructura cola

Tal y como se describió con anterioridad se implementaron las funciones *semSignal*, *semWait* y *semInit*, sin embargo para que no se confundan estas funciones con las llamadas al sistema se denominaron: *do_incrementar*, *do_decrementar* y *do_iniciar_sem*. Las cuales pueden observarse a continuación:

```
/*=========================================================*
 *                                do_incrementar                                        *
 *=========================================================*/
PUBLIC int do_incrementar()
{
        int Nsem,who;
        Nsem = m_in.m1_i1;
        Sem[Nsem]++;          /* incrementa el semaforo */

        if(Sem[Nsem] <= 0){ /* hay un proceso bloqueado? */
                if(fifo_desencolar(&FIFO[Nsem],&who)){/* desencola el primero */
                        setreply(who, ENOSYS);/* habilita para recibir mensaje */
                }else{
                        return -1; /* si hay error retorna -1 */
                }
        }
        return OK; /* no hay, retorna ok */
}
```

```
/*=========================================================*
 *                                do_decrementar                                        *
 *=========================================================*/
PUBLIC int do_decrementar()
{
        int Nsem,who;
        Nsem = m_in.m1_i1;
        who = m_in.m1_i2;
        Sem[Nsem]--;          /* decrementa al semaforo */

        if(Sem[Nsem]>0){ /* si queda positivo */
                return OK;
        }else{                                  /* si queda negativo */
                if(fifo_encolar(&FIFO[Nsem],who)){ /* encola al proceso */
                        return SUSPEND;       /* inhabilira para recibir mensaje */
                }else{
                        return -1;
                }
        }
}
```

```
/*=========================================================*
 *                                do_decrementar                                        *
 *=========================================================*/
PUBLIC int do_decrementar()
{
        int Nsem,who;
        Nsem = m_in.m1_i1;
        who = m_in.m1_i2;
        Sem[Nsem]--;          /* decrementa al semaforo */

        if(Sem[Nsem]>0){ /* si queda positivo */
                return OK;
        }else{                                  /* si queda negativo */
                if(fifo_encolar(&FIFO[Nsem],who)){ /* encola al proceso */
                        return SUSPEND;       /* inhabilira para recibir mensaje */
                }else{
                        return -1;
                }
        }
}
```

```
/*===========================================*
 *            do_iniciar_sem                 *
*============================================*/
PUBLIC int do_iniciar_sem()
{
        int Nsem,valor;
        int auxiliar;

        Nsem = m_in.m1_i1;
        valor = m_in.m1_i2;

        Sem[Nsem] = valor; /* se inicializa el semaforo con el valor */
        /* se debe inicilizar la cola, r_idx = w_idx = N */
        FIFO[Nsem].w_idx = N;
        FIFO[Nsem].r_idx = N;

        return OK;
}
```

Una vez modificado todos los archivos, introduzca los siguientes comandos e inicie desde la nueva imagen creada.

```
# cd /usr/src
# make libraries
# cd /usr/src/servers/pm
# make
# cd /usr/src/tools
# hdboot
```

Paso 4 (Implementación de programas de prueba): Para la implementación de los programas de pruebas se utilizaran dos códigos que se muestran a continuación en la Figura 10.8y la Figura 10.9. El primero sólo se encarga de inicializar el semáforo indicado por parámetro en cero y posteriormente de hacer un wait del semáforo indicado por parámetro para que el programa se bloquee. El otro código sólo hace un *signal* del semáforo pasado por parámetro. Ambos imprimen por salida estándar el PID del proceso. Estos programas están ubicados en la el directorio /usr/src. Y tienen la siguiente estructura:

```c
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <lib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
        int PID;
        message m;
        /* argumentos:
           *       argv[0] => puede ser [0,1] = indica el semaforo a iniciar
           *       argv[1] => puede ser [0,1] = indica el semaforo a hacer sem_wait
           */
        _syscall(MM,MINIX_GETPID,&m);
        PID = m.m2_i1;
        printf("soy el proceso == %d \n",PID);
        if(atoi(argv[1])==0){
                        printf("voy a iniciar el semaforo[0] en == 0\n");
                        sem_init(atoi(argv[1]),0);
        }else if(atoi(argv[1])==1){
                printf("voy a iniciar el semaforo[1] en == 0\n");
                sem_init(atoi(argv[1]),0);
        }
        if(atoi(argv[2])==0){
                printf("voy a hacer un sem_wait del semaforo[0]\n");
                sem_wait (0);
                printf("soy el proceso == %d y me debloquee\n",PID);
        }else if(atoi(argv[2])==1){
                printf("voy a hacer un sem_wait del semaforo[1]\n");
                sem_wait (1);
                printf("soy el proceso == %d y me debloquee\n",PID);
        }
        return 0;
}
```

Figura 10.8 prueba_sem_wait.c

```c
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <lib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
        int PID;
        message m;

        /* argumentos:
         *      argv[0] => [0,1] = indica el semaforo a iniciar
         *      argv[1] => [0,1] = indica a que semaforo se aplica sem_wait
         */

        _syscall(MM,MINIX_GETPID,&m);
        PID = m.m2_i1;
        printf("soy el proceso == %d \n",PID);
        printf("voy a hacer un signal del semaforo[%d]\n",atoi(argv[1]));
        sem_signal (atoi(argv[1]));

        return 0;

}
```

Para comprobar la correcta implementación de los semáforos implementados habilite las cuatro consolas de minix y ubíquese en el directorio */usr/src/.* Luego ejecute los programas de la siguiente forma:

- En primera instancia compile el código *prueba_sem_wait.c* y ejecútelo con los parámetros mostrados en la siguiente figura. Esto hace que el programa inicialice el semáforo 0 con un valor igual a 0. Posteriormente el programa hace un wait del semáforo y como es de esperarse el mismo se queda bloqueado, el semáforo queda con valor igual a -1. Como puede observarse el proceso al ejecutarse el programa tiene PID igual a 110.

```
# cc prueba_sem_wait.c -o prueba_sem_wait
# ./prueba_sem_wait 0 0
soy el proceso == 110
voy a iniciar el semaforo[0] en == 0
voy a hacer un sem_wait del semaforo[0]
_
```

- El segundo paso es ejecutar el mismo código como se muestra debajo. Esta ejecución sólo un wait del semáforo 0, quedando con valor igual a -2. Y desde luego el proceso con PID igual a 111 queda bloqueado.

Figura 10.9 prueba_sem_signal.c

```
# ./prueba_sem_wait -1 0
soy el proceso == 111
voy a hacer un sem_wait del semaforo[0]
_
```

- El tercer programa a ejecutarse sigue siendo el código de prueba_sem_wait.c, sin embargo, en este caso se hace una inicialización del semáforo 1 con un valor igual a 0. Después, el proceso hace un wait del mismo semáforo para quedar bloqueado. El PID del proceso es igual a 112.

```
# ./prueba_sem_wait 1 1
soy el proceso == 112
voy a iniciar el semaforo[1] en == 0
voy a hacer un sem_wait del semaforo[1]
_
```

- Luego se ejecuta el programa *prueba_sem_signal.c*. En ese caso se hace un signal del semáforo 1, y como es de esperarse el proceso con PID 112 es desbloqueado. Puede observarse a continuación. El proceso que ejecuta el signal tiene un PID igual a 113.

```
# cc prueba_sem_signal.c -o prueba_sem_signal
# ./prueba_sem_signal 1
soy el proceso == 113
voy a hacer un signal del semaforo[1]
# _
```

Se puede observar como el proceso con PID 112 es desbloqueado.

```
# ./prueba_sem_wait 1 1
soy el proceso == 112
voy a iniciar el semaforo[1] en == 0
voy a hacer un sem_wait del semaforo[1]
soy el proceso == 112 y me debloquee
# _
```

- Ahora se ejecutará nuevamente el programa prueba_sem_signal.c. Pero en esta oportunidad se hace un signal del semáforo 0. Se obtiene el resultado que el proceso con PID 110 es desbloqueado. Puede observarse a continuación. El proceso que ejecuta el signal tiene un PID igual a 113.

```
# cc prueba_sem_signal.c -o prueba_sem_signal
# ./prueba_sem_signal 1
soy el proceso == 113
voy a hacer un signal del semaforo[1]
# ./prueba_sem_signal 0
soy el proceso == 113
voy a hacer un signal del semaforo[0]
# _
```

Se puede observar como el proceso con PID 112 es desbloqueado.

```
# cc prueba_sem_wait.c -o prueba_sem_wait
# ./prueba_sem_wait 0 0
soy el proceso == 110
voy a iniciar el semaforo[0] en == 0
voy a hacer un sem_wait del semaforo[0]
soy el proceso == 110 y me debloquee
# _
```

- Para culminar con la prueba se ejecutará nuevamente el programa prueba_sem_signal.c. Se hace otro un signal del semáforo 0. Se obtiene el resultado que el proceso con PID 111 es desbloqueado. Puede observarse a continuación. El proceso que ejecuta el signal tiene un PID igual a 113.

```
# cc prueba_sem_signal.c -o prueba_sem_signal
# ./prueba_sem_signal 1
soy el proceso == 113
voy a hacer un signal del semaforo[1]
# ./prueba_sem_signal 0
soy el proceso == 113
voy a hacer un signal del semaforo[0]
# ./prueba_sem_signal 0
soy el proceso == 113
voy a hacer un signal del semaforo[0]
# _
```

Se puede observar como el proceso con PID 111 es desbloqueado.

```
# ./prueba_sem_wait -1 0
soy el proceso == 111
voy a hacer un sem_wait del semaforo[0]
soy el proceso == 111 y me debloquee
# _
```

11 Modificación del planificador de procesos

La planificación de procesos es una característica provista por los sistemas operativos modernos que le permite garantizar un comportamiento multiprogramado al sistema. Regularmente esta funcionalidad es provista por una pieza de software conocida como el “planificador” encargado de asignar los recursos de un sistema entre los múltiples procesos que lo solicitan. Siempre que exista la necesidad de tomar una decisión referente a la asignación de recursos entrara en ejecución el planificador para definir qué proceso recibirá el recurso. Se debe hacer hincapié que el procesador es la pieza más importante del computador, pero sigue siendo un recurso del mismo (23).

Hay dos tipos fundamentales de planificadores [Referencia libro] que coexisten dentro del sistema operativo, y existe un tercero que ha surgido para manejar nuevos estados de los procesos, estos son:

- **Planificador de largo plazo:** este determina que trabajos se admiten en el sistema para su procesamiento y cual(es) serán alojados en la memoria principal. Además, es el principal responsable que se cumplan las condiciones definidas referentes al manejo del procesador y los dispositivos de entrada/salida.
- **Planificador de corto plazo:** este determina que proceso del sistema que se encuentre en un estado activo (en espera de procesador) lo selecciona y lo lleva al procesador, regularmente este es un código bastante corto debido a que es un programa que se ejecuta mucho en el sistema y para garantizar un mayor rapidez del sistema operativo se exige eso. Durante la ejecución del sistema operativo cada vez que se presenta algún evento que implica un cambio en el proceso que debe pasar al estado activo. Los eventos que regularmente disparan la ejecución de este planificador son los siguientes:
    - Las señales de reloj del sistema.
    - Las interrupciones.
    - La finalización de las operaciones de entrada/salida.
    - Las llamadas al sistema operativo.
    - El envió y la recepción de señales.
    - La activación de programas interactivos.

    En otros casos, se divide al planificador en dos partes, el primero conocido como Schedule que se encarga solo de manejar la cola de procesos en espera por el CPU y el dispatcher que es aquel que lleva a cargo en si la tarea de asignar el procesador al proceso seleccionado en la cola.

- **Planificador a medio plazo:** este planificador surge en el hecho de que en algunos casos es conveniente llevar a la memoria secundaria algún proceso que se encuentre en un estado de

suspendido lo que permite liberar espacio en la memoria principal para albergar a un proceso que requiera el procesador.

Para tener un mejor entendimiento de estos conceptos vea la Figura 11.1 en la donde se puede observar los escenarios de cada uno de los planificadores son ejecutados (23).

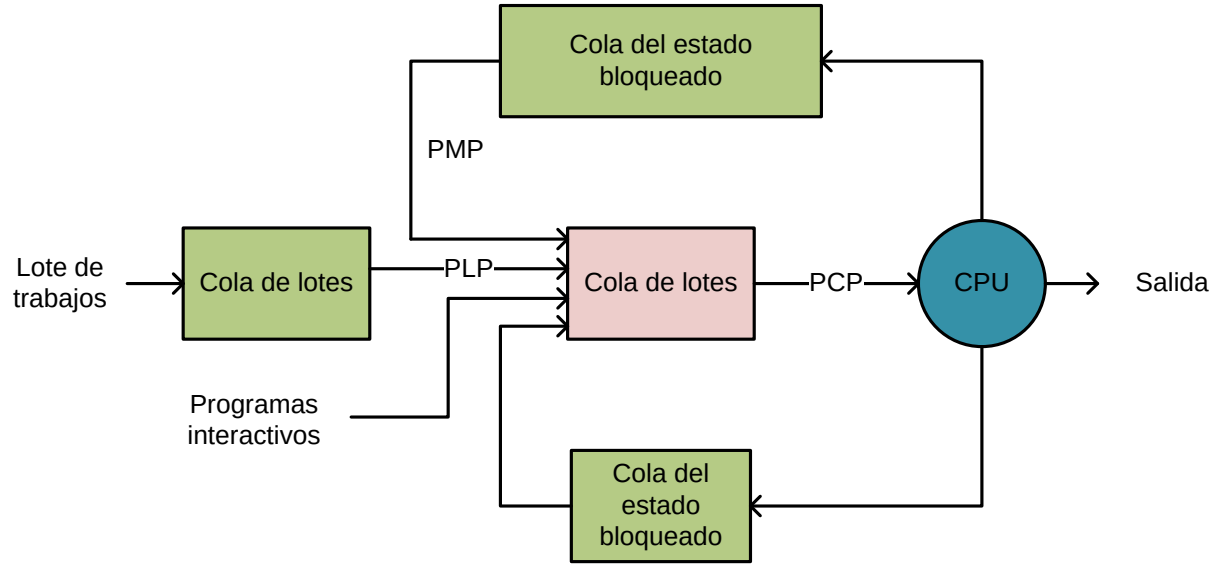


Figura 11.1 Diagrama de planificación

11.1 *Criterios para la planificación:*

Un algoritmo de planificación de corto plazo tiene distintas propiedades dependiendo de los criterios o fundamentos con los que este haya sido diseñado, lo que conlleva a que siempre existirá un tipo de proceso que se verá discriminado porque este no cumple en su totalidad con la característica impuesta en la implementación del algoritmo para su escogencia.

Así que antes de mostrar cual(es) son los algoritmos de planificación debe revisarse los criterios más importantes referentes al diseño o escogencia de un algoritmo para la gestión de la planificación. Alguno de estos criterios son los siguientes:

- **Eficacia**: se expresa como el porcentaje o la media de utilización del procesador.
- **Rendimiento:** es una medida que expresa el número de procesos culminados por unidad de tiempo.
- **Tiempo de retorno:** es el intervalo de tiempo delimitado por el inicio del proceso al sistema y su finalización y salida del mismo.
- **Tiempo de espera:** es el tiempo que espera el proceso para poder usar el procesador.

- **Tiempo de respuesta:** se denomina así al intervalo de tiempo que transcurre desde que se señala un evento hasta que se ejecuta la primera instrucción de la rutina de servicio de dicho evento.

11.2 *Algoritmos de planificación*

Existen un número elevado de algoritmos propuestos para llevar a cabo la tarea de planificación en los Sistemas Operativos cuya adecuación depende precisamente del tipo de planificación que se desee manejar y de los objetivos que se persigan con la misma. Hoy día debido a la diversidad de sistemas podríamos conseguir muchos de estos en herramientas modernas. Enfocado a este estudio el enfoque va dedicado a analizar aquellos algoritmos orientados directamente al comportamiento del planificador a corto plazo.

Antes de mostrar algún algoritmo debemos definir un concepto importantísimo en dichos algoritmos referentes a lo que se llama un algoritmo apreciativo.

Un algoritmo de planificación es catalogado como apreciativo cuando el proceso que se encuentra en ejecución o mejor dicho que posee al recurso procesador este puede ser interrumpido por el sistema operativo y colocarlo en un estado de listo o preparado nuevamente. A diferencia que aquellos de naturaleza no apropiativa donde el proceso activo permanece con el recurso procesador hasta que culmine toda su labor en el sistema o hasta que el proceso en si lo libere, la ejecución del planificador a corto plazo en este tipo de algoritmos no es tan frecuente como el anterior debido a que la cantidad de veces que se seleccionara a algún proceso del estado preparado a llevarlo al procesador dependerá exclusivamente del tiempo de ejecución del proceso activo.

11.3 *Planificación por prioridades*

Este algoritmo consiste en asignarle prioridades a los procesos y el de mayor prioridad que se encuentra en la cola de procesos preparados o disponibles será el que tome el uso del procesador. El valor inicial de esta prioridad puede ser asignada por el usuario a través de algún utilitario como lo es el comando *nice* en los Sistemas Operativos Unix-like o es asignada directamente por el sistema. La asignación de este valor puede ser (23):

- **Estática:** en este caso la prioridad asignada al proceso no cambia a lo largo de su estadía en el sistema.
- **Dinámica:** en este caso la prioridad asignada al proceso puede cambiar a lo largo de su estadía en el sistema, este cambio puede venir de parte del sistema o del usuario, un ejemplo seria el uso del utilitario *renice* de los Sistemas Operativos Unix-like que le permite al usuario modificar la prioridad de un proceso residente en el sistema.

Los algoritmos por prioridades pueden ser apropiativos o no apropiativos, es decir, en el primer caso si llega a la cola de procesos preparados un proceso que posee mayor prioridad que aquel que se encuentra ejecutándose en el procesador entonces el planificador a corto plazo toma la decisión de quitarle el recurso procesador y asignárselo al proceso de mayor prioridad que ha llegado al sistema.

Regularmente los algoritmos por prioridades que manejen este enfoca deben tener sumo cuidado en no relegar a los procesos de menor prioridad a lo que se llama muerte por inanición, es decir, los procesos de menor prioridad se ejecutaran muy poco o incluso no llegaran nunca a ejecutarse, por lo tanto, sus tareas no podrán ser realizadas y además estos consumen recursos del sistema. Para minimizar esta consecuencia se plantea el uso de prioridades dinámicas en el algoritmo para que aquellos procesos que tienen mucho tiempo en el sistema puedan ir aumentando su prioridad para poder acceder al procesador o por el contrario, para aquellos procesos que han utilizado continuamente el procesador se le decrementa su prioridad para que la competencia con el procesador sea más equitativa.

11.4 *Planificación FIFO (First In First Out)*

Es el algoritmo de planificación más sencillo de implementar es aquel que la cola de procesos preparados es evaluada siempre seleccionando al proceso que se encuentre en la primera posición de la misma y los procesos se irán encolando acorde a su tiempo de llegada.

Este método es rara vez utilizado aunque existen algunas implementaciones como las colas multinivel donde cada cola representa una prioridad en el sistema, y como todos los procesos que se encuentren en una cola particular comparten características similares entonces utilizar FIFO sería una forma sencilla y equitativa de atenderlos.

11.5 *Planificación SJF (Shortest Job First)*

Es un algoritmo no apropiativo en la que cada proceso se le asocia una estimación del tiempo que le resta para finalizar su ejecución y su selección en la cola de procesos preparados es llevada a cabo con dicho parámetro. En caso de que existan dos procesos cuyo tiempo restante sean iguales entonces se procede la escogencia por el parámetro de tiempo de llegada a la cola.

Este algoritmo de planificación podría ser optimo e incluso podríamos catalogarla como de los mejores diseños el problema recurre en como determinar o mejor dicho estimar el tiempo restante de ejecución de un proceso lo que requiere algún proceso que se encargue de calcular estos tiempos ,por lo tanto, eso cuesta tiempo de ejecución para una tarea que luego me va a permitir seleccionar al proceso que realmente llevara a cabo una tarea específica en el sistema y además si se desea manejar un histórico para agilizar los cálculos entonces la problemática se presentara para aquellos procesos sean nuevos en el sistema ,por lo tanto, es muy complicada de implementar y la misma podría ser muy costosa.

11.6 *Planificación SRT (Shortest Remaining Time)*

Este algoritmo es muy similar al presentado anteriormente su principal diferencia es que este método de planificación es apropiativo. El algoritmo de igual manera que el anterior selecciona a los procesos de la cola de preparados o activo de igual manera, pero si a esta cola llega un proceso cuyo tiempo de finalización sea menor que el del proceso que se está ejecutando en el procesador entonces el planificador toma la decisión de sacar a dicha tarea del procesador y asignárselo al proceso que ha llegado a la cola de preparados o activos.

La limitante de este algoritmo sigue siendo la misma que la del anterior, calcular el tiempo restante de ejecución del proceso sería sumamente costoso.

11.7 *Planificación RR (Round Robin)*

Este algoritmo le asigna a todos los procesos ubicados en la cola de procesos listos un quantum de tiempo donde este define el tiempo que dicho proceso podrá utilizar el procesador y la asignación del mismo se va realizando de manera secuencial. Si algún proceso requiere de la asignación de un nuevo quantum de tiempo entonces el planificador a corto plazo se lo asigna y lo coloca al final de la cola de procesos preparados o listos véase la Figura 11.2

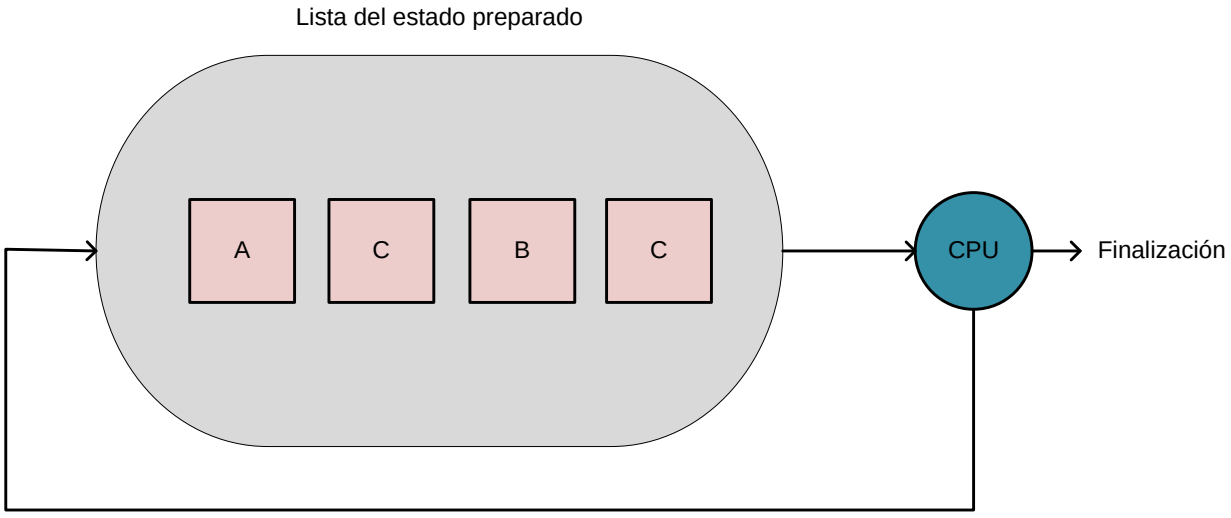


Figura 11.2 Diagrama de planificación RR (Round Robin)

El diseño de este algoritmo exige la existencia de un temporizador que sea capaz de llevar el control de los quantum de tiempos asignados y además de generar la interrupción en el sistema para que se pueda indicar la finalización del quantum de tiempo para que el procesador sea asignado al siguiente proceso en la cola de espera o listos.

La principal problemática que presenta este algoritmo es referente a de que tamaño será el quantum de tiempo a asignar a los procesos ya que acorde a la definición de esta variable se podrá observar el buen o no uso del procesador.

11.8 *Planificación MLQ (Multi-level Queues)*

En este algoritmo se plantea una estrategia para llevar a cabo alguna clasificación de los procesos se encuentran en el sistema y dependiendo de ella los procesos ingresaran a la cola correspondiente, a su vez cada una de estas colas puede ejecutar un algoritmo de planificación diferente.

El diseño de esta estrategia de planificación se basa principalmente en como categorizar a los procesos y manejar cual será el orden o prioridad en el que se van a manejar la elección de alguna de estas colas para luego decidir acorde al algoritmo de planificación particular de dicha cola como se seleccionan a los procesos que allí se encuentren.

Regularmente, la elección de la cola se lleva a cabo por prioridades donde se examinan cada una de las colas de forma secuencial y en caso de existir en alguna de ellas procesos en espera por el procesador entonces esa será la cola que se va a manejar. Esto puede presentar un problema con aquellos procesos que residan en las colas de menor prioridad ocasionando las consecuencias de los algoritmos de planificación por prioridades donde estos procesos tendrán la posibilidad de morir por inanición en el sistema.

11.9 *Planificación MLFQ (Multi-level Feedback Queues)*

En este algoritmo al igual que el anterior se debe plantear una estrategia para llevar a cabo alguna clasificación de los procesos para que estos sean asignados a las colas que les corresponden. El algoritmo anterior no trae la limitante de que los procesos siempre irán a la misma cola y nos trae la consecuencia de la muerte por inanición, por lo tanto, este algoritmo plantea la necesidad de que los procesos dependiendo de un parámetro puedan ir siendo asignados a diferentes colas para que el uso del procesador sea lo más equitativo posible entre todos los procesos del sistema.

Un ejemplo bastante sencillo sería el siguiente, se tiene un algoritmo de planificación MLFQ que está definido por tres colas, la primera se gestiona a través de RR con un quantum de 10ms, la segunda cola al igual que la anterior, pero con un quantum de 20ms y por último la tercera cola es manejada por planificación FIFO. El planificador a corto plazo ira chequeando por cada una de las colas por la existencia de procesos en espera entonces supongamos un proceso de la primera cola siempre tendrá mayor prioridad que el anterior y en caso de que su tiempo de vida en el sistema es extremadamente largo entonces los procesos de las otras colas se verán discriminados en el uso del procesador, por lo tanto, cuando este proceso se le acaba su quantum de tiempo y resulta que requiere de un nuevo

quantum para seguir su ejecución este en vez de ser asignado en la primera cola este es asignado a la segunda cola al final y se le da acceso al procesador ahora a los procesos de esta cola, y nuevamente ocurre el mismo escenario el proceso llega al procesador aun no logra culminar su ejecución en el sistema, por lo tanto, el planificador a corto plazo lo asigna a una cola FIFO donde este tendrá la posibilidad de culminar su tarea y no discriminara a ningún proceso en el sistema, véase la Figura 11.3 donde se muestra un diagrama de lo anterior planteado (23).

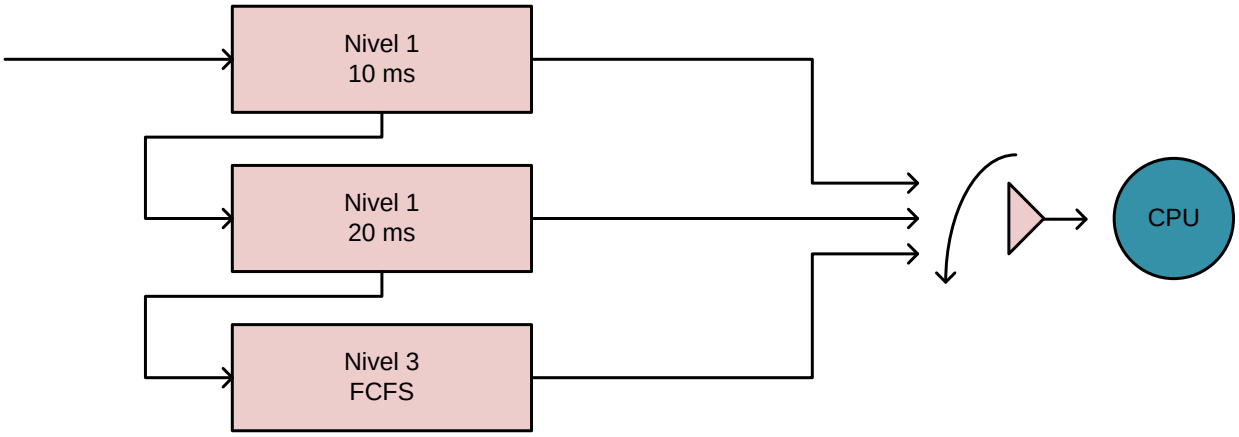


Figura 11.3 Diagrama de planificación MLFQ

Existen muchos algoritmos posibles de planificación MLFQ que pueden definirse de forma más general por los siguientes parámetros:

- El numero de colas a implementar.
- El algoritmo de planificación de cada una de las colas.
- Los métodos o condiciones que determinan el movimiento de los procesos entre las distintas colas.

11.10 *Planificación de procesos en Minix*

Dentro de la amplitud de la gestión de procesos en un sistema operativo, la parte que se lleva a cabo dentro del micronúcleo es la tarea de planificación. Un sistema multiprogramado se basa en las interrupciones, que permite al núcleo gestionar las peticiones de entrada/salida de los procesos y además controlar los tiempos de ejecución de cada proceso.

Los procesos se bloquean cuando hacen peticiones de entrada/salida, permitiendo la ejecución de otros procesos. Cuando la petición ha sido resuelta, el proceso en ejecución es interrumpido por el disco, el teclado o cualquier otra pieza de hardware, y deja de estar activo mientras el dispositivo atiende su petición. El reloj también genera interrupciones, utilizadas para garantizar que un proceso que no ha

realizado entrada/salida libere la CPU en algún momento y permita la ejecución de otros procesos. Es tarea de las capas más bajas de Minix 3 ocultar esas interrupciones transformándolas en mensajes. Desde el punto de vista de los procesos, cuando una operación de E/S termina, envía un mensaje a algún proceso, despertándolo y marcándolo como listo para ejecutar.

Las interrupciones también pueden ser generadas por software, caso en el que suelen ser llamadas *traps*. Las operaciones send y receive que son traducidas por la librería del sistema como interrupciones software, que tienen exactamente el mismo efecto que las interrupciones generadas por hardware -el proceso que lanza una interrupción software se bloquea inmediatamente y el núcleo se activa para procesar la interrupción. Los programas de usuario no invocan directamente send o receive, pero las llamadas al sistema implicadas ejecutan sendrec y generan una interrupción software.

Cada vez que un proceso es interrumpido (ya sea por un dispositivo E/S convencional o por el reloj) o debido a la ejecución de una interrupción software, existe una oportunidad para determinar nuevamente el proceso que tiene más derecho a ejecutarse. Por supuesto, esto también debe realizarse cada vez

Que un proceso termina, pero en un sistema como Minix 3 las interrupciones debidas a E/S, el reloj o el paso de mensajes ocurren de manera más frecuente que la finalización de un proceso.

11.10.1 Algoritmo de planificación en Minix v3.1.6

El platicador de Minix 3 utiliza un sistema de varios niveles de colas, cada una con distinta prioridad. Se deñen dieciséis colas, aunque puede recompilarse para usar más o menos colas de manera sencilla. La cola de menor prioridad es utilizada únicamente por el proceso IDLE, que se ejecuta cuando no hay nada más que hacer. Los procesos de usuario comienzan por defecto en una cola varios niveles de prioridad por encima de la más baja.

Los servidores normalmente se planifican en colas con prioridades más altas que las permitidas a los procesos. Los controladores de dispositivos en colas con prioridades mayores que los servidores y Clock Task y System Task se planifican en las colas de máxima prioridad. No tienen por qué estar en uso las dieciséis colas en un momento determinado.

Los procesos se inician únicamente en algunas de ellas. Un proceso puede ser movido a una cola de prioridad diferente por el sistema o por un usuario que invoque la orden nice. Además de la prioridad determinada por la cola en la que se coloca un proceso, se utiliza otro mecanismo para dar ventaja a unos procesos sobre otros: el quantum, un intervalo de tiempo mínimo que puede ejecutar un proceso antes de ser expropiado, aunque no es idéntico para todos los procesos. Los procesos de usuario tienen un quantum relativamente bajo, mientras que los controladores y los servidores normalmente se ejecutan hasta que ellos mismos se bloquean.

Sin embargo, como medida contra el funcionamiento incorrecto de los mismos, se han programado de manera en que puedan ser expropiados, pero se les asigna un quantum mayor. Tienen permitido ejecutar por un periodo largo, pero finito, de tiempo, pero si utilizan todo su quantum son expropiados para evitar que el sistema se bloquee. En estos casos, el proceso se considera preparado para ejecutar y se coloca al final de su cola. Sin embargo, si un proceso que ha utilizado todo su quantum fue el mismo que se ejecutó por última vez, se interpreta que puede estar bloqueado en un bucle y puede estar evitando que otros procesos se ejecuten. En estos casos, su prioridad se ve reducida, colocándolo al final de una cola de prioridad inferior. Si el proceso se quedase de nuevo sin tiempo, su prioridad se vería reducida de nuevo. Tarde o temprano, algún otro proceso tendrá una oportunidad para ejecutarse.

Un proceso al que se le ha reducido su prioridad puede recuperarla. Si un proceso utiliza todo su quantum, pero no impide que otros procesos se ejecuten, se asciende ese proceso a una cola de prioridad superior, hasta la prioridad máxima que el proceso tenga permitida.

Los procesos son planificados utilizando un RR ligeramente modificado (24). Si un proceso no ha utilizado todo su quantum cuando pasa a estar en estado no ejecutable, se interpreta que el proceso se ha bloqueado esperando a E/S, y cuando vuelva a estar listo para ejecutar, se colocará en la cabeza de su cola, pero tan solo con la cantidad de quantum que le quedaba cuando se bloqueó. La idea es proporcionar a los procesos de usuario una respuesta rápida a la E/S. Un proceso que es expropiado porque ha terminado su quantum se coloca al final de su cola, siguiendo el esquema RR.

Con las tareas situadas en la prioridad más alta, los controladores después, los servidores detrás de los controladores, y los procesos de usuario al final, un proceso de usuario no se ejecutará hasta que ningún proceso de sistema tenga nada que hacer, y un proceso de usuario no puede evitar que un proceso de sistema se ejecute.

Cuando se selecciona un proceso para ejecutar, el planificador comprueba si hay algún proceso esperando en la cola de mayor prioridad. Si hay alguno listo, el que se encuentre en la cabeza de la cola es ejecutado. Si no hay ninguno listo, se comprueba la cola de prioridad inmediatamente inferior, y así repetida mente. Puesto que los controladores responden a peticiones de los servidores y los servidores a peticiones de procesos de usuario, en algún momento todos los procesos de prioridad alta completarán la tarea que estén realizando. Entonces se bloquearán sin nada que hacer hasta que algún proceso de usuario tenga oportunidad de ejecutarse y realice más peticiones. Si no existe ningún proceso preparado, entonces el proceso IDLE es elegido. Esto coloca a la CPU en un estado de bajo consumo hasta la siguiente interrupción.

En cada tick de reloj, se realiza una comprobación para ver si el proceso actual ha agotado su quantum. Si lo ha hecho, el planificador lo mueve al final de su cola. Entonces, se elige el siguiente proceso para

ser ejecutado, como se ha descrito anteriormente. Sólo si no hay procesos en las colas de mayor prioridad y el proceso anterior está solo en su cola, será seleccionado para ejecutar inmediatamente.

En otro caso, el proceso en la cabeza de la cola de mayor prioridad será ejecutado. Los controladores esenciales y los servidores tienen un quantum tan largo que normalmente no son interrumpidos jamás por el reloj. Pero si algo va mal, su prioridad puede ser temporalmente reducida para evitar que el sistema llegue a un bloqueo completo. Probablemente no se pueda hacer nada útil si esto sucede con un controlador imprescindible, pero quizá sea posible apagar el sistema correctamente, evitando pérdida de datos y recolectando información que puede ayudar en la depuración del problema.

11.10.2 Desarrollo de ambiente de pruebas sobre el planificador en Minix

En las secciones anteriores de este capítulo se ha descrito el procedimiento de planificación de forma general, los distintos algoritmos de planificación existentes y además se ha mostrado de manera detallada cual es el funcionamiento de todo este proceso en el Sistema Operativo Minix versión 3.1.6, entonces ahora se procede a realizar algunas pruebas de desempeño.

Las pruebas consisten en un primer lugar en la ejecución de dos programas uno que tenga alta carga de uso del CPU y otro que tenga una alta carga de peticiones de Entrada/Salida, véase la Figura 5.4. Estos dos programas deberán ejecutarse en las distintas capas en las que está diseñado el Sistema Operativo Minix v3.1.6:

- Ejecución de ambos programas en la capa 4 de Minix v3.1.6 que corresponde con el espacio de procesos de usuario.
- Ejecución de ambos programas en la capa 3 de Minix v3.1.6 que corresponde con el espacio de procesos servidores, véase el Capítulo 3 de este documento de investigación donde podrá ver los pasos que debe seguir para desarrollar una llamada al sistema en la capa 3.
- Ejecución de ambos programas en la capa 1 de Minix v3.1.6 que corresponde con el espacio del núcleo donde operan los programas Clock y SystemTask, véase el Capítulo 3 de este documento de investigación donde podrá ver los pasos a seguir para desarrollar una llamada al sistema que le permita ejecutar su código en la SystemTask.

Para el ambiente de pruebas se han desarrollado los siguientes programas para comprobar el funcionamiento del planificador de Minix v3.1.6 y observar el desempeño del mismo, estos son los siguientes:

- **altaiouser.c:** este programa se ejecuta en la capa de procesos de usuario realizando muchas peticiones de Entrada/Salida (véase Figura 11.4)

```c
#include <lib.h>
#include <stdio.h>

void main(int argc, char *argv[]){
        int i,j;
        printf("Programa con alta carga de Entrada/Salida ejecutandose a nivel de usuario.\n");
        for(i=0;i<1000;i++){
                for(j=0;j<1000;j++){
                        printf("Imprimir por pantalla como proceso en el nivel de usuario.\n");
                }
        }
}
```

Figura 11.4 Código fuente de altaiouser.c

altaioserver.c: este programa hace una llamada al sistema que se encuentra implementada en la capa de procesos servidores donde se realizan muchas peticiones de Entrada/Salida (véase Figura 11.5)

```c
#include <lib.h>
#include <stdio.h>

void main(int argc, char *argv[]){
        int retorno;
        message m;
        retorno = _syscall(MM,68,&m);
        printf("Resultado de altaioserver:[%d]\n",retorno);
}
```

Figura 11.5 Código fuente de altaioserver.c

- **altaiokernel.c:** este programa hace una llamada al sistema que se encuentra implementada en la capa de procesos servidores, donde esta hace una llamada luego al SystemTask y luego allí se ejecuta el programa que realiza muchas peticiones de Entrada/Salida (véase Figura 11.6).

```c
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]){
        int retorno, entrada=5, salida=7;
        retorno = altaiokernel(entrada, &salida);
        printf("Resultado de la llamada es:[%d], valor salida:[%d]\n",retorno,salida);
        return(0);
}
```

Figura 11.6 Código fuente de altaiokernel.c

- **altacpuuser.c:** este programa se ejecuta en la capa de procesos de usuario generando alta carga de CPU (véase Figura 11.7)

```
#include <lib.h>
#include <stdio.h>

void main(int argc, char *argv[]){
        int i,j;
        printf("Programa con alta carga de CPU ejecutandose a nivel de usuario.\n");
        for(i=0;i<1000;i++){
                for(j=0;j<1000;j++){

                }
        }
}
```

Figura 11.7 Código fuente de altacpuuser.c

- **altacpuserver.c:** este programa hace una llamada al sistema que se encuentra implementada en la capa de procesos servidores generando alta carga de CPU (véase Figura 11.8).

```
#include <lib.h>
#include <stdio.h>

void main(int argc, char *argv[]){
        int retorno;
        message m;
        retorno = _syscall(MM,69,&m);
        printf("Resultado de altacpuserver:[%d]\n",retorno);
}
```

Figura 11.8 Código fuente de altacpuserver.c

- **altacpukernel.c:** este programa hace una llamada al sistema que se encuentra implementada en la capa de procesos servidores, donde esta hace una llamada luego al SystemTask donde allí reside el programa que genera alta carga de CPU (véase Figura 11.9).

```c
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]){
        int retorno, entrada=5, salida=7;
        retorno = altacpukernel(entrada, &salida);
        printf("Resultado de la llamada es:[%d], valor salida:[%d]\n",retorno,salida);
        return(0);
}
```

Figura 11.9 Codigo fuente de altacpukernel.c

Estos código fuentes serán ejecutados manejando el numero de colas predefinidas por el Sistema Operativo Minix v3.1.6 y luego serán modificados el numero de colas a su mitad.

11.10.3 Manejo de colas de planificación en Minix v3.1.6.

Anteriormente se realizo una explicación bien detallada sobre el algoritmo utilizado por el planificador a corto de plazo del Sistema Operativo Minix versión 3.1.6 y en esta sección tan solo mostraremos la ubicación de aquellos archivos fuentes donde son definidas dichas características.

En el archivo */usr/src/kernel/proc.h* podemos conseguir casi al finalizar el archivo la definición del numero de colas que manejara el planificador, la definición de las prioridades o asignaciones de colas para los procesos del núcleo, se define la cola para el procesos IDLE ,pero principalmente se define el comportamiento de los procesos de usuario esto debido a que existe un archivo donde se definen las asignaciones de las colas para los procesos de capa 3 hacia abajo durante el procedimiento de arranque del sistema operativo, véase Figura 11.10

```
234 /* Scheduling priorities for p_priority. Values must start at zero (highest
235  * priority) and increment.  Priorities of the processes in the boot image
236  * can be set in table.c. IDLE must have a queue for itself, to prevent low
237  * priority user processes to run round-robin with IDLE.
238  */
239 #define NR_SCHED_QUEUES   16    /* MUST equal minimum priority + 1 */
240 #define TASK_Q             0    /* highest, used for kernel tasks */
241 #define MAX_USER_Q         0    /* highest priority for user processes */
242 #define USER_Q            (NR_SCHED_QUEUES / 2) /* default (should correspond to
243                                                    nice 0) */
244 #define MIN_USER_Q        (NR_SCHED_QUEUES - 1) /* minimum priority for user
245                                                    processes */
246
```

Figura 11.10 Código fuente de /usr/src/kernel/proc.h

En el archivo /usr/src/kernel/table.c se encuentran definidos aquellos procesos que son importantes para

el arranque del Sistema Operativo, este archivo debe ser modificado cuando se desee disminuir el número de colas significativamente, véase Figura 11.11

```
62 PUBLIC struct boot_image image[] = {
63 /* process nr, pc, flags, qs,  queue, stack,   name */
64 {IDLE,       NULL,     0,  0,      0, IDL_S, "idle"  },
65 {CLOCK,clock_task,     0,  8, TASK_Q, TSK_S, "clock" },
66 {SYSTEM, sys_task,     0,  8, TASK_Q, TSK_S, "system"},
67 {HARDWARE,      0,     0,  8, TASK_Q, HRD_S, "kernel"},
68 {PM_PROC_NR,    0,     0, 32,      3, 0,     "pm"    },
69 {FS_PROC_NR,    0,     0, 32,      3, 0,     "vfs"   },
70 {RS_PROC_NR,    0,     0,  4,      3, 0,     "rs"    },
71 {MEM_PROC_NR,   0, BVM_F,  4,      3, 0,     "memory"},
72 {LOG_PROC_NR,   0, BVM_F,  4,      2, 0,     "log"   },
73 {TTY_PROC_NR,   0, BVM_F,  4,      1, 0,     "tty"   },
74 {DS_PROC_NR,    0, BVM_F,  4,      3, 0,     "ds"    },
75 {MFS_PROC_NR,   0, BVM_F, 32,      3, 0,     "mfs"   },
76 {VM_PROC_NR,    0,     0, 32,      2, 0,     "vm"    },
77 {PFS_PROC_NR,   0, BVM_F, 32,      3, 0,     "pfs"   },
78 {INIT_PROC_NR,  0, BVM_F,  8, USER_Q, 0,     "init"  },
79 };
```

Figura 11.11 Código fuente de /usr/src/kernel/table.c

11.11*Análisis de resultados.*

Los resultados obtenidos son manejados a través del comando time proporcionado en el Sistema Operativo Minix v3.1.6 donde se pueden observar los tiempo de vida del proceso dentro del sistema.

Los resultados debemos dividirlos en dos etapas, la primera como se comento en secciones anteriores será ejecutar cada uno de dicho programas con la definición predeterminada por el Sistema Operativo Minix versión 3.1.6 donde se manejan 16 colas de planificación y un segundo escenario donde se va a reducir el número de colas a ocho.

Los siguientes son los resultados obtenidos:

altaiouser.c: este programa es ejecutado en la capa 4 del sistema operativo, correspondiente a la sección de procesos de usuario. En ambos casos tanto para la prueba manejando 16 colas de planificación véase la Figura 11.12 y 8 colas véase Figura 11.13 el tiempo de vida del proceso es sumamente largo debido a que las peticiones de Entrada/Salida que este realiza son suficientes para que su resultado sea el esperado. Se observo un decremento del tiempo de ejecución al disminuir las colas.

```
Imprimir por pantalla como proceso en el nivel de usuario.
   12:24.33 real      20.16 user    12:04.16 sys
#
```

Figura 11.12 Resultados altaiouser.c (16 Colas de planificación)

Figura 11.13 Resultados altaiouser.c (8 Colas de planificación)

altaioserver.c: este programa es ejecutado en la capa 3 del sistema operativo, correspondiente a la sección de procesos servidores. En ambos casos tanto para la prueba manejando 16 colas de planificación véase la Figura 11.14 y 8 colas Figura 11.15 el tiempo de vida del proceso es considerablemente menor al ejecutado en espacio de usuario debido a que estos procesos poseen mayor prioridad en el sistema. Se observo un decremento del tiempo de ejecución al disminuir las colas del procesador.

Figura 11.14 Resultados altaioserver.c (16 Colas de planificación)

Figura 11.15 Resultados altaioserver.c (8 Colas de planificación).

altaiokernel.c: este programa es ejecutado en la capa 1 del sistema operativo, correspondiente a la sección de procesos del núcleo. En ambos casos tanto para la prueba manejando 16 colas de planificación véase la Figura 11.16 y 8 colas véase Figura 11.17 el tiempo de vida es aun más corto que las pruebas anteriores debido a que estos procesos son los que poseen mayor prioridad dentro del sistema y además acorde a lo planteado en secciones anteriores dichos procesos no pueden ser sacados del procesador mientras lo están utilizando razón por la cual su ejecución es extremadamente veloz.

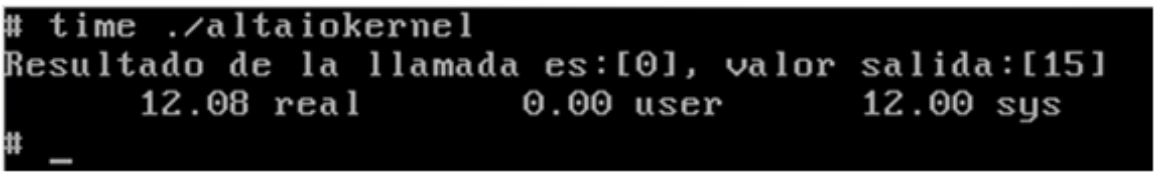


Figura 11.16 Resultados altaiokernel.c (16 Colas de planificación)

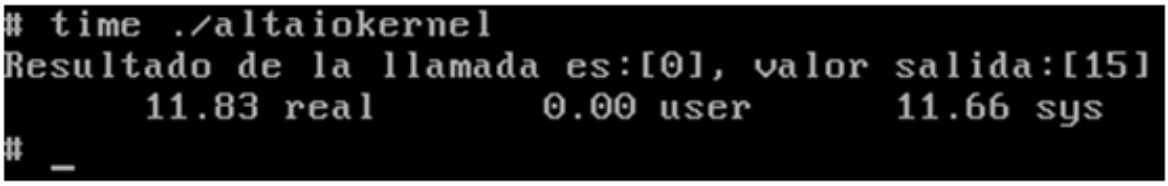


Figura 11.17 Resultados altaiokernel.c (8 Colas de planificación)

altacpuuser.c: este programa es ejecutado en la capa 4 del sistema operativo correspondiente a la sección de procesos de usuario. En ambos casos tanto para la prueba manejando 16 colas de planificación véase la Figura 11.18 y 8 colas véase Figura 11.19 el tiempo de vida es bastante corto debido a que este tipo de operaciones son manejadas con bastante velocidad por el procesador.

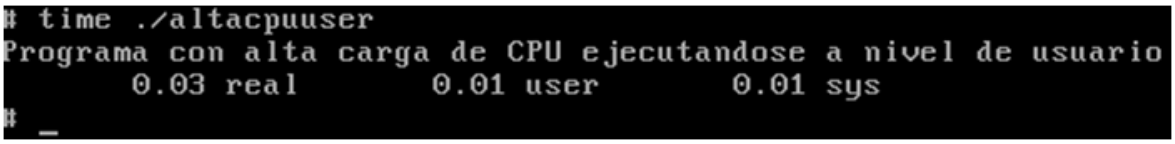


Figura 11.18 Resultados altacpuuser.c (16 Colas de planificación)

Figura 11.19 Resultados altacpuuser.c (8 Colas de planificación)

altacpuserver.c: este programa es ejecutado en la capa 3 del sistema operativo, correspondiente a la sección de procesos servidores. En ambos caso tanto para la prueba manejando 16 colas de planificación véase la Figura 11.20 y 8 colas véase Figura 11.21 el tiempo de vida es igual al anterior podemos deducir dicho comportamiento debido a que el tiempo necesario por el proceso servidor para culminar la ejecución de dicha tarea debe ser igual al tiempo necesario por un proceso de usuario, por lo tanto, el proceso servidor no consume su quantum de tiempo asignado para poder culminar la tarea.

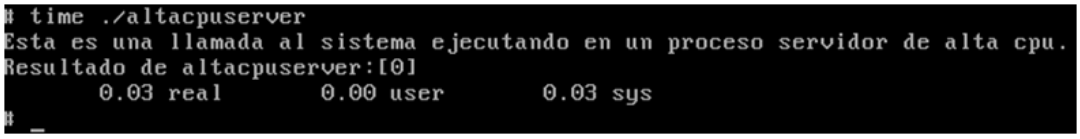


Figura 11.20 Resultados altacpuserver.c (16 Colas de planificación)

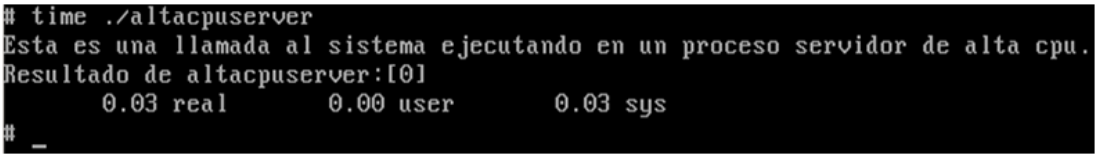


Figura 11.21 Resultados altacpuserver.c (8 Colas de planificación)

altacpukernel.c: este programa es ejecutado en la capa 1 del sistema operativo, correspondiente a la sección de procesos del núcleo. En ambos casos tanto para la prueba manejando 16 colas de planificación véase Figura 11.22 y 8 colas véase Figura 11.23 el tiempo de vida de es ligeramente mayor a los programas anteriores y podemos explicar este comportamiento debido a la sobrecarga que se debe añadir por el manejo de pase de mensajes, se debe recordar que la comunicación de este programa inicia en la capa 4 enviando un mensaje al proceso servidor que se encuentra en la capa 3 y luego es este último proceso que tiene la permisología suficiente para comunicarse con los procesos del núcleo que están alojados en la capa 1, todo este proceso ocurre desde el inicio del programa y luego se repite ,pero de manera ascendente para presentar los resultados.

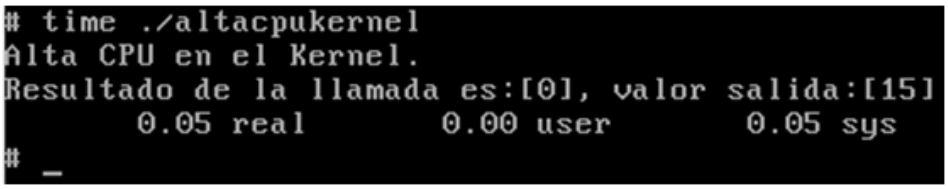


Figura 11.22 Resultados altacpukernel.c (16 Colas de planificación)

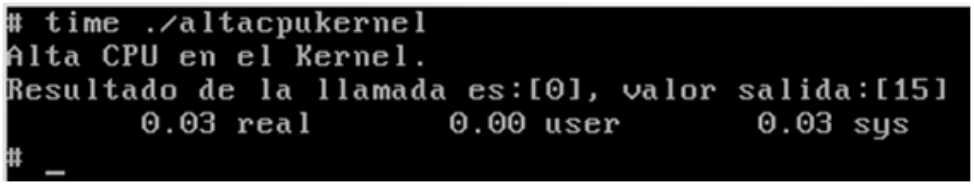


Figura 11.23 Resultados altacpukernel.c (8 Colas de planificación).

12 Conclusiones

Los sistemas operativos instruccionales son herramientas educativas extensas que pueden englobar una amplia gama de tópicos sobre sistemas operativos, desde el proceso de inicio hasta la seguridad; es por esto que permite ahilar la parte práctica de cualquier curso de sistemas operativos. Además, es una herramienta pedagógica que ofrece un entorno de desarrollo probado. Y lo más importante aún permite modificar realmente un sistema operativo, cosa que pocos estudiantes en ciencias de la computación han podido realizar.

Este Trabajo Especial de Grado pudo constatar que Minix versión 3.1.6 se adapta al curso de Sistemas Operativos vigente de la Escuela de Computación, así como también al pensum de estudios propuesto por la ACM (Association for Computing Machinery) e IEEE (Institute of Electrical and Electronics Engineers) para un curso de pregrado (25), a través de los siguientes puntos:

- Se comprobó a través del uso de Minix versión 3.1.6 como un Sistema Operativo estable, sencillo y de fácil manejo que provee como añadido la interacción de los estudiantes con un Sistema Operativo Unix-like.

- Se verificó la total compatibilidad de los laboratorios propuestos con el entorno de desarrollo, el cual está basado en el IDE eclipse, aplicación usada por los estudiantes en materias de semestres anteriores.

- Se realizo un análisis y estudio detallado sobre la estructura y algoritmo del gestor de arranque de Minix versión 3.1.6 ofreciendo los conocimientos tanto básicos como avanzados del funcionamiento de estos programas en un Sistema Operativo, dejando la documentación pertinente.

- Se hizo la implementación de un intérprete de comandos sencillo y básico que permite la ejecución de comandos y ordenes para convertirse en una interfaz humano computador similar a la ofertada por el intérprete de comandos de Minix.

- Se realizó una investigación sobre las llamadas al sistema en Minix versión 3.1.6, obteniendo los conocimientos de la codificación y función de las mismas. Además, se explica la implementación de las llamadas al sistema a los procesos servidores (capa 3) y las llamadas al núcleo (capa 1), el cual se logra a través del mecanismo de comunicación inter-procesos (pase de mensajes) planteado por Minix 3.

- Se analizó la posibilidad incorporar a Minix 3 la estructura de datos semáforos para la sincronización entre procesos. Se diseño e implemento la solución, para posteriormente comprobar la funcionalidad de los cambios realizados. Se realizo toda la documentación.

- Se realizo una investigación y estudio detallado sobre la programación y funcionamiento del algoritmo de planificación de procesos en Minix. En el cual se propone la modificación del mismo.

El Trabajo Especial de Grado desarrollado es de vital importancia dado que es el primer trabajo formal dentro de la Universidad Central de Venezuela donde se va ofertar el uso de un Sistema Operativo Instruccional para el curso de Sistemas Operativos de la Escuela de Computación. Además ofrece una herramienta colaborativa para la interacción entre estudiantes, docentes e investigadores para el continuo desarrollo de Minix versión 3.1.6. Sentando un precedente para las demás generaciones de estudiantes que quieran seguir con este tema y enfoque investigativo.

Esta investigación permitió la elaboración de un conjunto de laboratorios docentes para ser utilizados como plantillas para la elaboración de los espacios prácticos del curso de Sistemas Operativos. Además de proveer toda la documentación necesaria para la formulación y elaboración de los laboratorios. Asimismo la documentación para familiarizar al docente con el trabajo realizado. La documentación se elaboro acorde a la dinámica de los laboratorios con esto se hace la referencia sobre si el laboratorio es de desarrollo o es de análisis y estudio de alguna sección particular.

12.1 *Limitaciones*

A pesar que Minix versión 3 es un Sistema Operativo comercial de código abierto, su comunidad de desarrollo y/o soporte no es tan amplia cuando la comparamos con otros sistemas operativos comerciales de código abierto como las comunidades de GNU/Linux o BSD. A pesar que parte de la investigación se apoyo en dicha comunidad para la resolución de problemas, se presentaron situaciones donde la falta de documentación se convirtió en un obstáculo para que el proceso de adecuación fuese más sencillo. Durante el proceso de adecuación nos enfrentamos a las siguientes situaciones:

- Minix 3 posee unas versiones “netamente educativas” que refuerza el estudio de la herramienta, ya que el código de estas versiones es en esencia el mismo al que aparece en la publicación bibliográfica “Systems: Design and Implementation 3rd Edition”; y son todas aquellas versiones de Minix inferior a las versión 3.1.4. El inconveniente con estas versiones se produjo en el proceso de acoplamiento con IDE Eclipse, ya que el servicio de conexión remota SSH para la integración con aplicación no funcionaba correctamente, es por esta razón que se escogió la versión 3.1.6.

- Durante el proceso de diseño de los laboratorios se pensó generar un laboratorio docente donde los estudiantes pudiesen estudiar y modificar el manejador de memoria de Minix versión 3.1.6, al generar las plantillas relacionadas al laboratorio se presentaron los siguientes inconvenientes:

    - Minix versión 3.1.6 posee una técnicas de gestión de memoria diferente con respecto al encontrado en la documentación bibliográfica oficial de la herramienta. El libro menciona

que Minix 3 implementa la técnica de gestión de memoria segmentación sencilla usando el primer ajuste. Sin embargo, Minix versión 3.1.6 implementa la técnicas de gestión de memoria paginación sencilla.

  - Se propuso modificar el manejador de memoria de Minix versión 3.1.6 para que este utilizara el modulo de memoria desarrollado para la versión 3.1.4 y esta solicitud se presento a su comunidad de desarrollo donde inmediatamente se menciono que dicho cambio es bastante complejo; ya que entre cada una de estas versiones la estructura de los procesos servidores de Minix había sido cambiada. Por lo que no se recomendaba realizar el cambio sino utilizar una versión inferior a Minix, pero esto no era una opción para los laboratorios ya que se perdería todo el entorno de programación.

- Durante el proceso de adecuación se propuso generar un laboratorio para el estudio de los sistemas de archivos a pesar que este tópico no se encuentra en las recomendaciones de la ACM para los cursos básicos de Sistemas Operativos (25). En este laboratorio se propuso modificar el sistema de archivos de Minix para agregarle la funcionalidad de cifrado/descifrado del mismo lo que genero una problemática similar a la expuesta el punto anterior.

12.2 *Trabajos futuros*

Tomando como base el presente Trabajo Especial de Grado se propone establecer una comunidad o grupo de trabajo dedicado a esta área de estudios cuya motivación principal sea velar por la actualización, adecuación y desarrollo del sistema operativo instruccional Minix versión 3.1.6, integrada por el grupo docente de la materia Sistemas Operativos y los estudiantes interesados en seguir investigando sobre este punto. También se plantea como futuros trabajos de investigación la generación de los laboratorios de manejo de memoria y sistemas de archivos, sin olvidar la actualización de los laboratorios propuestos.

12.3 *Recomendaciones*

A lo largo de este Trabajo Especial de Grado y la investigación anterior realizada sobre los Sistemas Operativos Instruccionales documento previo a este trabajo en la modalidad de seminario, debe revisarse los siguientes puntos:

- Actualización del curso con respecto a la propuesta desarrollada por la ACM sobre los cursos de sistemas operativos.

- Elaborar una nueva dinámica pedagógica del curso Sistemas Operativos que contemple los siguientes aspectos:

- Agregar un espacio de discusión presencial donde estudiantes y docentes puedan reunirse para discutir los desarrollos propuestos a lo largo del curso, lo que permita aclarar dudas, solventar fallas o problemas de las herramientas utilizadas, discusiones para la generación de nuevos proyectos.

- Actualizar el material didáctico para que este utilice como caso de estudio al sistema operativo instruccional Minix versión 3.1.6, para que esta manera el estudiante siempre tenga la posibilidad de reforzar sus conocimientos a través de esta herramienta instruccional.

- Generar charlas y conferencias relacionadas al proyecto donde cualquier miembro de la Escuela de Computación, es decir, estudiantes y docentes puedan discutir y elaborar nuevos temas de investigación relacionada a los sistemas operativos.

- Modificar el plan de evaluación del curso para generar mayor equidad entre las evaluaciones teóricas y prácticas. Este punto particular es importante ya que se pudo constatar en su revisión de los cursos de Sistemas Operativos ofertados por las principales casas de estudios superiores a nivel mundial ofrecen una calificación de aproximadamente un 50% para la teoría y un 50% para la práctica. Se considera que la revisión debe ser pertinente para alcanzar mejores resultados académicos tanto de docentes y estudiantes (13).

- Se propone la elaboración de un laboratorio general de desarrollo de sistemas operativos en la Escuela de Computación basado en Minix versión 3.1.6 donde los estudiantes puedan estudiar y desarrollar un más a fondo el sistema operativo, basado en los conceptos ya forjados por esta investigación.

- Se propone expandir el Sistemas Operativo Instruccional Minix versión 3.1.6 para que se desarrollen herramientas tales como, un manejador de ventanas y escritorio, desarrollo de aplicaciones para el nivel de desarrollo y de usuarios entre otros.

- Para finalizar, se propone ambiciosamente la generación de una nueva mención profesional para la Escuela de Computación cuyo principal objeto de estudios sean los Sistemas Operativos donde se tomen cursos avanzados referente a esta área; donde se puedan estudiar, proponer y revisar como casos de estudios de sistemas operativos comerciales tales como: Microsoft Windows, GNU/Linux, OS X, Unix entre otros. Esta mención debería incluso agregar a su curriculum académico el desarrollo de sistemas operativos embebidos y móviles.

13 Referencias

(1. **Anderson, Charles y Nguyen, Minh.** *A Survey Of Contemporary Instructional Operating Systems For Use in Undergraduate Courses.* Oregon : Wester Oregon University, 2005.

2. **Stallings, William.** *Sistemas Operativos.* Madrid : Prentice Hall, 2005.

3. **Tanenbaum, Andrew.** *Sistemas Operativos Modernos.* México : Prentice Hall, 2003.

4. **Tanenbaum, Andrew y Woodhull, Albert.** *Sistemas operativos - Diseño e Implementacion.* Mexico : Prentice Hall, 1999.

5. **Silberchatz, Abraham, Galvin, Peter y Gagne, Greg.** *Fundamentos de Sistemas Operativos.* Madrid : McGraw Hill, 2006.

6. **Holland, David, Lim, Ada y Nguyen, Minh.** *A New Instructional Operating System.* Massachusetts : Harvard University, 2002.

7. **Cartereau, Michel.** *A Tool For Operating System Teaching.* Paris : Department of Mathematics and Computer Scienc e, 1993.

8. **Hovemeyer, Howard, Hollingsworth, Jeffrey y Bhattacharjee, Bobby.** *Running on the Bare Metal with GeekOS.* Maryland : University of Maryland, 2003.

9. **Universidad de Harvard.** OS/161. [En línea] Mayo de 2010. http://www.eecs.harvard.edu.

10. **Procter, Christopher y Anderson, Thomas.** *The NachOS Instructional Operating System.* California : University of California, Berkeley, Diciembre - 2002.

11. **Gary, James.** *Using NachOS in a Upper Division Operating Systems Course.* Western Kentucky University : Western Kentucky University, 2001.

12. **Instituto Tecnológico de Massachusetts.** xv6, A Teaching Operating System. [En línea] http://pdos.csail.mit.edu.

13. **De León Dumar, España José.** *Diseño, implementación y adecuación de una herramienta educativa para cursos de Sistemas Operativos.* Caracas : s.n., 2010.

14. **MINIX 3.** [En línea] Septiembre de 2010. http://www.minix3.org/.

15. **Andrew S Tanenbaum, Albert S Woodhull.** *Operating Systems Design and Implementation.* s.l. : Prentice Hall, 2006. 3 edition.

16. **Croucher, Phil.** *The Bios Companion: The Book That Doesn't Come With Your Motherboard.* s.l. : Electrocution Technical Publishers, 2004.

17. **Stallings, William.** *Computer Organization and Architecture.* s.l. : Prentice Hall, 2009. 8 edition.

18. **Randal E. Bryant, David R. O'Hallaron.** *Computer Systems, A Programmer's Perspective.* s.l. : Prentice Hall, 2003. First edition.

19. **Universität Passau.** Universität Passau. [En línea] Octubre de 2010. http://www.uni-passau.de/.

20. **Universidad de Extremadura.** Universidad de Extremadura. [En línea] Septiembre de 2010. http://www.unex.es/.

21. **Universidad Tecnológica Nacional.** Diseño e implementación de Sistemas Operativos. [En línea] Octubre de 2010. http://www.frsf.utn.edu.ar/.

22. **Jorritn. Herder, Herbertbos, Bengras, Philiphomburg, y Andrews. Tanenbaum.** *modular system programming in MINIX 3.* Amsterdam : s.n., 2005.

23. **Joaquin Aranda, Antonia Canto Diaz y Jesus Manuel de la Cruz Garcia.** *Sistemas Operativos: Teoria y Problemas.*

24. **J. Adrián Bravo Navarro, Héctor Cortiguera Herrera yJorge Quintás Rodríguez.** *Minix 3 sobre arquitectura ARM.* Madrid : s.n., 2009.

25. **Association for Computing Machinery and IEEE Computer Society.** *Computer Science Curriculum 2008.* 2009.