

Apunte de Sistemas Operativos

Una visión general

(borrador)



Esteban De La Fuente Rubio

esteban[at]delaf.cl

28 de abril de 2015

Licencia de uso

Este documento se encuentra bajo la licencia de documentación libre GNU o GFDL. A continuación se indican las condiciones generales respecto al uso de este documento, una versión oficial (en inglés) y una traducción no oficial.

Copyright (c) 2014 Esteban De La Fuente Rubio

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

Copyright (C) 2014 Esteban De la Fuente Rubio

Se autoriza la copia, distribución y/o modificación de este documento bajo los términos de la Licencia de documentación libre de GNU, Versión 1.3 o cualquiera posterior publicada por la Fundación para el Software Libre; sin secciones invariantes, ni textos de portada, ni textos de contraportada.

Una copia de esta licencia puede ser encontrada (en inglés) en la siguiente dirección:
<http://www.gnu.org/licenses/fdl-1.3.txt>.

Agradecimientos

Son varias las personas que me han ayudado y apoyado durante los años, varios durante el tiempo que llevo escribiendo este apunte, y justamente gracias a algunos de ellos es que decidí escribirlo en “formato libro”. Algunos son:

Al profesor Gabriel Astudillo, ya que con su apoyo y motivación fomentó en mi la parte práctica en sistemas operativos *like Unix*.

Al profesor Carlos Abarzúa, quién fue la persona que me introdujo en el área de teoría de los sistemas operativos.

Al profesor Guillermo Badillo, quién confió en mi por primera vez para dictar el curso de Sistemas Operativos en la Universidad Andrés Bello.

Al profesor Luis Mateu, quién me aceptó como oyente¹ en sus clases de Sistemas Operativos en la Universidad de Chile y en cuyas clases se basó gran parte de este documento.

Además quisiera agradecer a todos aquellos que han permitido que este proyecto salga adelante, ya sea con sus aportes, su ánimo o su paciencia (sobre todo alumnos).

Adicionalmente agradeceré a cualquier lector que me indique sugerencias sobre el contenido, reportes de errores o solicitud de algún tema que haya sido omitido o no cubierto en la profundidad esperada. La idea es evaluar estos comentarios y hacer los respectivos cambios en futuras versiones del documento.

¹Fui oyente en otoño 2012, su alumno en otoño 2013, su auxiliar en primavera 2013 y otoño 2014.

Resumen

Este documento tiene por objetivo presentar de manera sencilla los principales conceptos relacionados con los Sistemas Operativos. Para lograr esto se introducirá al lector en distintos temas, cubriendo tópicos desde los inicios de los sistemas operativos hasta sistemas operativos modernos.

Es importante destacar que la motivación al escribir este documento es que pueda servir como guía para quién se está introduciendo en conceptos relacionados al área de Sistemas Operativos o bien esté tomando un curso de introducción a los sistemas operativos donde la mayor parte de los temas mencionados en este documento son utilizados.

Al ir avanzando en el documento el lector se irá adentrando, básicamente, en las áreas de gestión de procesos, memoria principal y memoria secundaria. Para lograr esto se cubren aspectos tanto teóricos como prácticos, esto último en sistemas operativos *like Unix*, como GNU/Linux.

El presente material está realizado en base a conocimientos adquiridos a lo largo de los años, material de otros profesores que he tenido la suerte de conocer y bibliografía de autores reconocidos en la materia. No se pretende que este documento reemplace otras alternativas bibliográficas que el lector pueda consultar, solo se presenta como un apunte para aquellos interesados en el tema.

En <https://github.com/cursos/sistemas-operativos> el lector podrá encontrar una copia digital, actualizada y gratuita de este documento.

Índice general

Índice de figuras	xiv
Índice de cuadros	xv
1. Introducción	1
1.1. Visión general	2
1.2. Objetivos del sistema operativo	5
1.2.1. Objetivos del usuario	5
1.2.2. Objetivos del sistema	6
1.3. Servicios ofrecidos	6
1.3.1. Gestión de procesos	6
1.3.2. Gestión de memoria principal	7
1.3.3. Gestión de memoria secundaria	7
1.4. Ejercicios y preguntas	8
1.5. Referencias	8
2. Historia	9
2.1. Tipos de sistemas	9
2.1.1. Érase una vez (50's)	9
2.1.2. Sistemas por lotes (fines 50's)	10
2.1.3. Sistemas de operación <i>offline</i> (60's)	12

2.1.4. Sistemas con <i>buffering</i> (60's)	13
2.1.5. Sistemas de operación <i>online</i> (60's)	14
2.1.6. Sistemas multiprogramados (fines 60's)	14
2.1.7. Máquinas o procesadores virtuales	15
2.1.8. Sistemas de tiempo compartido (70's)	15
2.1.9. Computadores personales	16
2.1.10. Redes de computadores personales (80's)	17
2.1.11. Sistemas en tiempo real	17
2.1.12. Sistemas distribuídos	17
2.1.13. Sistemas multiprocesadores	18
2.2. Tendencias últimas dos décadas	18
2.3. Logros	19
2.3.1. Procesos y memoria	19
2.3.2. Seguridad y protección	20
2.3.3. Gestión de recursos	21
2.3.4. Estructura del sistema	21
2.4. Ejercicios y preguntas	22
2.5. Referencias	23
3. Estructura y diseño	25
3.1. Interfaz de usuario	26
3.1.1. CLI	27
3.1.2. GUI	28
3.2. API y llamadas al sistema	30
3.2.1. Tipos de llamadas al sistema	30
3.3. Diseño	33
3.3.1. Objetivos	35
3.3.2. Políticas y mecanismos	35

3.3.3. Requerimientos para protección de procesos	36
3.4. Estructura del sistema operativo	38
3.4.1. Estructura simple	38
3.4.2. Estructura en niveles	39
3.4.3. <i>Microkernels</i>	40
3.4.4. Módulos	41
3.5. Implementación	41
3.6. Ejercicios y preguntas	42
3.7. Referencias	43
4. Procesos	45
4.1. Distribución de la memoria	45
4.2. Contexto	47
4.2.1. Atributos del proceso	48
4.2.2. Cambios de contexto	50
4.3. Estados	51
4.4. Clasificación de procesos	55
4.4.1. Procesos <i>preemptive</i> y <i>non-preemptive</i>	56
4.5. Paralelismo	57
4.5.1. <i>Data races</i>	58
4.5.2. <i>Deadlock</i>	60
4.5.3. <i>Starvation</i>	64
4.6. Ejercicios y preguntas	65
4.7. Referencias	66
5. Sincronización	67
5.1. <i>Busy-waiting</i>	67
5.2. Problemas clásicos	69

5.2.1. Productor consumidor	69
5.2.2. Cena de filósofos	72
5.2.3. Lectores escritores	73
5.3. Semáforos	77
5.3.1. API	77
5.3.2. Modo de operación	78
5.3.3. Problema productor consumidor	78
5.3.4. Problema cena de filósofos	79
5.4. Monitores de Brinch Hansen	82
5.4.1. API	83
5.4.2. Problema productor consumidor	84
5.4.3. Patrón de solución usando monitores	87
5.4.4. Problema cena de filósofos	88
5.4.5. Problema lectores escritores	89
5.5. Mensajes	94
5.5.1. API	95
5.5.2. Ejemplos de uso	95
5.5.3. Exclusión mutua con mensajes	97
5.5.4. Implementación de semáforos a partir de mensajes	99
5.6. Monitores de Hoare	101
5.6.1. Implementación de semáforos a partir de monitores	101
5.6.2. Problema productor consumidor	103
5.6.3. Solución de verdad: monitores de Hoare	105
5.7. Ejercicios y preguntas	106
5.8. Referencias	108
6. Planificación de monoprocesadores	109
6.1. Algoritmos de planificación	113

6.1.1.	FCFS: First Come First Served	114
6.1.2.	SJF: Shortest Job First	115
6.1.3.	Primero el de menor tiempo restante	116
6.1.4.	Primero el de mayor tasa de respuesta	116
6.1.5.	Prioridades	117
6.1.6.	Round Robin	118
6.2.	Planificación en Linux	120
6.2.1.	Planificación de procesos convencionales	121
6.2.2.	nice	123
6.3.	Ejercicios y preguntas	124
6.4.	Referencias	126
7.	Memoria principal	127
7.1.	Espacio de direcciones	128
7.1.1.	Enlace de direcciones	129
7.1.2.	Direcciones virtuales y direcciones físicas	133
7.1.3.	Unidad de administración de memoria MMU	134
7.2.	Asignación no contigua	135
7.2.1.	Segmentación	135
7.2.2.	Paginación	142
7.3.	Memoria virtual	150
7.3.1.	Algoritmos de reemplazo de páginas	152
7.4.	Ejercicios y preguntas	160
7.5.	Referencias	162
8.	Memoria secundaria	163
8.1.	Archivo	163
8.1.1.	Estructura	164

8.1.2. Atributos	164
8.1.3. Operaciones	166
8.1.4. Métodos de acceso	167
8.2. Estructura del disco	169
8.3. Estructura de directorios	170
8.3.1. Operaciones	170
8.3.2. Organización de los directorios	170
8.4. Montaje	172
8.5. Compartición	173
8.5.1. Protección	174
8.6. Ejercicios y preguntas	175
8.7. Referencias	175
9. Protección y seguridad	177
9.1. Principios de protección	177
9.2. Dominios	178
9.2.1. Dominios en sistemas <i>like Unix</i>	179
9.3. Matriz de acceso	179
9.4. Ejercicios y preguntas	181
9.5. Referencias	182
A. Máquinas virtuales	183

Índice de figuras

1.1.	Visión global del Sistema Operativo	3
2.1.	Tarjetas de un sistema por lotes	11
2.2.	Sistemas por lotes vs sistema multiprogramado	15
3.1.	Diversos entornos gráficos	29
3.2.	Componentes presentes en el modo usuario y el modo sistema	37
4.1.	Distribución de la memoria	46
4.2.	Estados de un proceso	52
4.3.	Interbloqueo, espera circular entre proceso A y B	61
5.1.	Interacción entre productores y consumidores	71
5.2.	Problema cena de filósofos	72
5.3.	Ejemplo 1 de uso de mensajes	96
5.4.	Ejemplo 2 de uso de mensajes	96
5.5.	Ejecución de la función f en el servidor	97
6.1.	Ejemplo ráfagas de CPU	112
6.2.	Planificación considerando colas de prioridades (RQ_0 a RQ_n)	117
6.3.	Planificación considerando colas de prioridades con realimentación	119
7.1.	Tipos de memorias (tiempos, tamaños y costos)	128

7.2. Registro base y límite de un bloque de memoria	129
7.3. Protección de memoria, se limita el acceso al bloque del proceso	130
7.4. Asignación contigua	130
7.5. Tipos de enlaces de direcciones de memoria principal	131
7.6. Registro base y límite de un bloque de memoria	136
7.7. Traducción de segmentos	138
7.8. Ejemplo de paginación con 3 procesos	143
7.9. Biblioteca compartida entre dos procesos	145
7.10. Traducción de páginas	147
7.11. Proceso que ocurre al existir un fallo de página	151
7.12. Proceso de intercambio de páginas	152
7.13. Ejemplo de intercambio con algoritmo FIFO	154
7.14. Ejemplo de intercambio con algoritmo Second Chance	156
8.1. Diferentes tipos de archivos	165
8.2. Acceso secuencial a un archivo	168
8.3. Acceso secuencial simulado con acceso directo	169
8.4. Jerarquía de directorios simple	171
8.5. Jerarquía de directorios con dos niveles	171
8.6. Jerarquía de directorios en árbol	172
8.7. Enlaces	173
9.1. Ejemplo de dominios y sus conjuntos de derechos de acceso	179
9.2. Ejemplo de matriz de acceso	180
9.3. Ejemplo de matriz de acceso usando dominios como objetos	181

Índice de cuadros

4.1. Comparativa entre procesos pesados y livianos	56
--------------------------------------------------------------	----

Capítulo 1

Introducción

El Sistema Operativo (*Operating System, OS*) corresponde a un programa en ejecución que se encarga de actuar como intermediario entre el usuario y la máquina. Sus dos principales objetivos corresponden a la **administración del hardware** y **ser una interfaz para el usuario**, de tal forma que este pueda interactuar con la máquina.

El sistema operativo deberá proveer de un ambiente para ejecutar los programas del usuario, siendo este **el único con privilegios de acceso directo al hardware** y los procesos deberán mediante el sistema operativo acceder a los recursos disponibles.

Los principales requisitos que se piden al sistema operativo corresponden a ser **cómodo** en cuanto a su interfaz y **eficiente**, de tal forma que no utilice todos los recursos de la máquina, dejándolos *libres* para los procesos de los usuarios.

Al estudiar la historia de los sistemas operativos se podrá apreciar como estos han ido evolucionando en el tiempo. Se verá que inicialmente no existía un sistema operativo como tal, sino que el hardware era programado directamente recibiendo los trabajos (*jobs*) que los usuarios deseaban ejecutar. Más adelante, al aparecer el concepto de un monitor residente, se verá que este corresponde a un programa que **siempre se encuentra cargado en memoria principal**.

1.1. Visión general

En la figura 1.1 se puede apreciar una visión global del sistema operativo, en esta se muestran las principales partes que se verán en el sistema. A continuación se describirá cada uno de estos componentes y como es la interacción que estos tienen entre sí.

- **Hardware:** recursos disponibles en la máquina, también podrán ser dispositivos virtuales, por estos los procesos competirán y desearán su uso.
- **Drivers:** código que permite el uso del hardware (o dispositivo virtual) al que están asociados.
- **Núcleo:** componente principal del sistema operativo, es el intermediario entre aplicaciones y el hardware, se encarga de las tareas de administración de la máquina.
- **Llamadas al sistema:** es la forma en que una aplicación hace alguna solicitud a un servicio del sistema operativo, generalmente con acceso privilegiado por lo cual son accedidas mediante una API.
- **API:** corresponde a una interfaz que pueden utilizar las aplicaciones para interactuar con el sistema operativo, las llamadas al sistema y eventualmente el hardware disponible, algunas ventajas de esto son la abstracción y el escribir menos código.
- **Utilitarios:** aquellas aplicaciones que vienen incluídas con el sistema operativo al momento de realizar la instalación, sin embargo esto dependerá mucho del sistema operativo propiamente tal y del contexto en que cada uno se instala.
- **Aplicaciones:** corresponden al software que se instala posteriormente a la instalación del sistema operativo. Generalmente serán las aplicaciones que principalmente le interesa ejecutar al usuario.
- **Usuario:** usuario del equipo que está interesado en ejecutar sus aplicaciones.

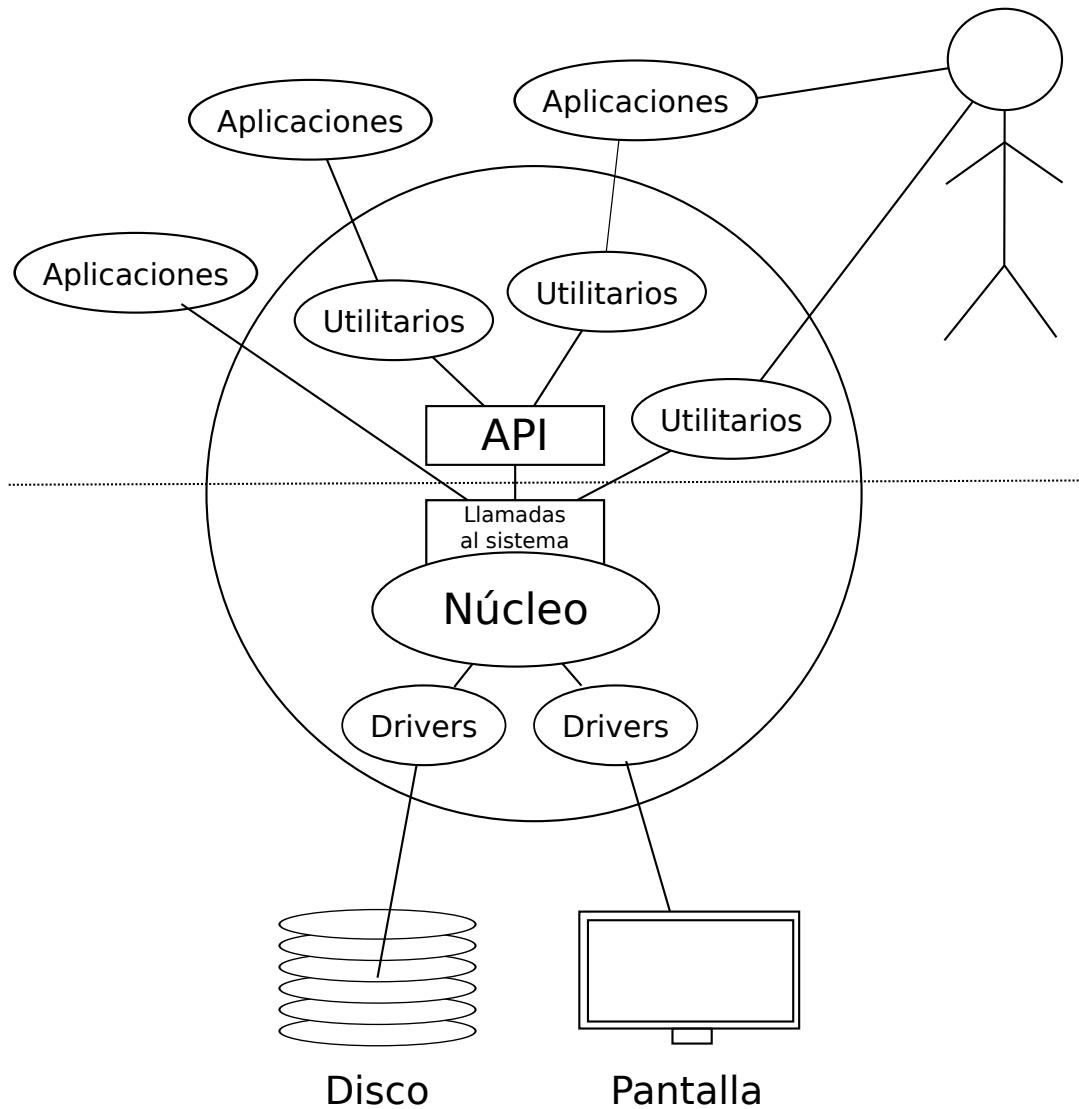


Figura 1.1: Visión global del Sistema Operativo

En la figura 1.1 se observa una línea divisoria entre los componentes superiores (Aplicaciones, Utilitarios y API) y la parte inferior (Llamadas al sistema, Núcleo, Drivers y Hardware). Esta división corresponde a la separación de privilegios necesarios para ser utilizadas. Si bien una aplicación podría acceder directamente a una llamada a sistema o al hardware (**en Unix todo es un archivo**, incluso el hardware) deberá tener los permisos adecuados para hacerlo. Se hablará más al respecto más adelante en el capítulo 3.

Una característica importante del sistema operativo, como ya se mencionó antes, es que es el único proceso que debe estar siempre cargado en memoria principal. Lo anterior implicará que todo el código del sistema operativo, incluyendo sus drivers deberán estar cargados siempre en RAM. Imagine que desea tener soporte para todos los tipos de impresoras disponibles, esto implicaría tener cargado todos los drivers en la memoria principal por si en algún momento usted la conecta. Son parte de la estructura del sistema operativo y su diseño la consideración de diferentes métodos de construcción del núcleo del sistema operativo, lo que permita cargar por *partes* el mismo, sin tener que tener cargado todo el código que eventualmente se pueda requerir en la memoria. Una de estas alternativas es lo que se utiliza en Linux, correspondiente al uso de módulos donde el núcleo los levanta a medida que los requiere. De esto también se hablará más en el capítulo ??.

Si bien los recursos con los que se cuenta hoy en día son muy superiores a los que se contaba cuando comenzaron los sistemas operativos, como se comentará en el capítulo 2, tenga en cuenta que la teoría de sistemas operativos asociada a los conceptos que se verán en este apunte es muy similar. Existen avances en tecnologías, pero lo que se abordará en este documento corresponde a lo tradicional relacionado con sistemas operativos. Por lo cual si usted no considera problema tener cargado 5, 15, 30 o 200 MB de núcleo RAM, recuerde que hace unos años los equipos no tenían las mismas capacidades y hace décadas eran impensables.

1.2. Objetivos del sistema operativo

El sistema operativo debe preocuparse de cumplir diferentes objetivos, estos se pueden dividir, según el interesado, en **objetivos del usuario** y **objetivos del sistema**.

1.2.1. Objetivos del usuario

Los usuarios finales en general no desean preocuparse por que tipo de hardware están utilizando. Un usuario que quiere guardar una fotografía en su computador no le interesa saber en que disco se esta guardando, de que tipo es el disco (IDE o SATA) o que tipo de sistema de archivos contiene (ext3, reiserfs o xfs), al usuario le interesa guardar la fotografía.

El objetivo de los desarrolladores estará asociado al fácil desarrollo de aplicaciones sobre el sistema operativo. Sin tener que, el programador de la aplicación, llegar a trabajar con lenguaje de bajo nivel o instrucciones privilegiadas para acceder al hardware de la máquina.

Servicios que serán de interés para los usuarios:

- **Creación de programas:** herramientas para realizar las tareas de desarrollo de aplicaciones para el sistema operativo.
- **Ejecución de programas:** administración de los procesos que se ejecutan en el sistema.
- **Acceso a los dispositivos de E/S:** simplificación de las tareas de uso del hardware, tales como escribir en una pantalla o leer datos desde el teclado.
- **Almacenamiento:** administración de los discos, la búsqueda de información en estos, su formato y gestión en general.
- **Memoria:** administración del uso de memoria principal disponible, así como la asignación de memoria virtual de ser necesario.

- **Detección y respuesta contra errores:** que sea capaz de detectar y proteger al sistema frente a eventuales anomalías.
- **Estadísticas:** llevar una recopilación con información sobre el uso de los recursos y parámetros generales sobre el sistema.

1.2.2. Objetivos del sistema

Desde el punto de vista del sistema la principal preocupación es realizar una administración eficiente y justa de los recursos de la máquina. Esto significa que todos sean atendidos en algún momento de tal forma que se les permita realizar sus operaciones de forma satisfactoria.

El sistema operativo será el encargado de determinar cuando y quién utilizará cierto recurso del sistema, tal como el procesador, la memoria principal, disco duro, etc. Y será el encargado de interrumpir al proceso que haga uso del recurso de tal manera de entregárselo a otro que también lo quiera utilizar.

1.3. Servicios ofrecidos

Históricamente se estudian principalmente tres áreas de la gestión realizada por el sistema operativo, aquellas relacionadas con los procesos, memoria principal y secundaria. Adicionalmente se verán en este documento otros aspectos como protección y seguridad.

1.3.1. Gestión de procesos

Un proceso corresponde a un programa en ejecución, el cual posee diferentes estados a lo largo de su vida como proceso. Principalmente interesan los estados listos y ejecución, que corresponden a la espera antes de ser planificado para entrar a la CPU y el de ejecución de código dentro de la CPU. Existen otros estados que serán vistos en detalle en el capítulo 4.

Todo proceso requerirá hacer uso de, al menos, memoria principal y CPU para su ejecución, por lo cual el sistema operativo deberá ser capaz de asignar estos recursos de una forma eficiente y justa, de tal forma que todos los procesos sean servidos según los vayan requiriendo.

Se estudiarán problemas que ocurren por la ejecución de múltiples procesos al mismo tiempo, concepto conocido como concurrencia, la forma de solucionarlo mediante sincronización, y algoritmos de planificación que permitirán elegir que proceso deberá entrar a la CPU.

1.3.2. Gestión de memoria principal

Todo proceso requerirá del uso de memoria principal para su ejecución, en este espacio de memoria se encontrará no solo el código del programa, sino también sus datos y su contexto. El sistema operativo deberá asignar, de algún modo, espacios de memoria para que el proceso los utilice, y si eventualmente el proceso requiere más espacio poder cumplir con su requerimiento.

¿Qué sucede si no disponemos de más memoria principal? La primera idea, sería decir que no podemos iniciar más procesos, lo cual sería cierto, sin embargo se discutirá el método de memoria virtual el cual permite utilizar un dispositivo de memoria secundaria para “obtener memoria” para los procesos.

1.3.3. Gestión de memoria secundaria

El sistema operativo debe ser capaz de almacenar datos en medios de memoria secundaria, la cual es permanente, a diferencia de la memoria principal que es volátil. Se deberá preocupar de la mantenición de una estructura de archivos y de poder realizar operaciones sobre esta estructura de tal forma que las aplicaciones no se preocupen de escribir *físicamente* el archivo que desean guardar en el disco.

1.4. Ejercicios y preguntas

1. Explique los objetivos del sistema operativo.
2. Explique los componentes de la visión general del sistema operativo y como se relacionan entre si. Realice diagrama.
3. ¿Por qué los objetivos del usuario y del sistema operativo no siempre son compatibles?.
4. ¿Cuáles son los servicios básicos que el sistema operativo debe proveer?.
5. ¿Cuándo se encuentra cargado el sistema operativo en RAM?.

1.5. Referencias

- Sistemas Operativos, Segunda Edición, Andrew Tanenbaum, Capítulo 1.1.

Capítulo 2

Historia

Los sistemas operativos han evolucionado enormemente desde sus inicios, durante este capítulo se mencionarán aspectos relacionados con los diferentes tipos de sistemas operativos existentes o que han existido junto a lo que se ha logrado a lo largo de los años.

2.1. Tipos de sistemas

2.1.1. Érase una vez (50's)

Cuando se comenzaron a utilizar grandes máquinas para realizar cómputos no existía un sistema operativo propiamente tal, solo había hardware y las aplicaciones (trabajos o *jobs*) de los usuarios. Se leía el programa a ejecutar y se ejecutaban sus instrucciones de manera secuencial.

Las aplicaciones en esta época debían incluir todo el código, incluyendo aquel para manejar cada uno de los dispositivos de hardware que la máquina tenía disponible. Se debe acceder directamente al espacio de direcciones, tanto de memoria principal como memoria secundaria. No existe estructura de directorios ni archivos en la memoria.

Si se desea ejecutar una nueva aplicación, se debe cargar el nuevo trabajo y la máquina

debe ser reiniciada. La depuración de la aplicación se realiza de forma presencial, observando las salidas indicadas mediante las luces del computador. El equipo es utilizado solo por un usuario al mismo tiempo y por un período largo de tiempo para realizar todas las tareas involucradas.

El principal problema de esta etapa es el bajo uso del hardware disponible, donde existe mucho tiempo que no es utilizado en cómputo. Siendo que el componente caro es el hardware y no el programador, se debe buscar una solución a este problema.

A pesar de lo anterior, y lo rudimentario que podría parecer frente a los computadores que actualmente existen, estas máquinas eran capaces de procesar cálculos mucho más rápido que un gran número de calculistas trabajando en conjunto.

2.1.2. Sistemas por lotes (fines 50's)

En un sistema operativo por lotes se requiere que todos los componentes del programa, ya sea el mismo código del programa, los datos y las llamadas al sistema que permiten usar el hardware sean introducidos, comúnmente, mediante tarjetas perforadas, ver figura 2.1, de 80 caracteres cada una. Este conjunto de instrucciones es conocido como un trabajo o un *job*, el cual poseía poca o ninguna interacción con el usuario.

Este tipo de sistemas operativos era útil con programas que fuesen largos y sin interacción con el usuario. Donde un operador recibía los trabajos de los programadores y los introducía en la máquina, estos eran procesados en orden FIFO (First In First Out), o sea el primer trabajo que llegaba era el primero que se procesaba.

La planificación del procesador y administración de la memoria es simple, en el primer caso bastaba pasar el control del mismo al *job* y que este lo devolviera al terminar su ejecución. En el caso de la memoria también la administración era simple, ya que el espacio se dividía en dos partes, una para el monitor residente (el sistema operativo) y otra para el trabajo en ejecución.

Al aparecer el monitor residente, también aparece el concepto de API, donde el progra-

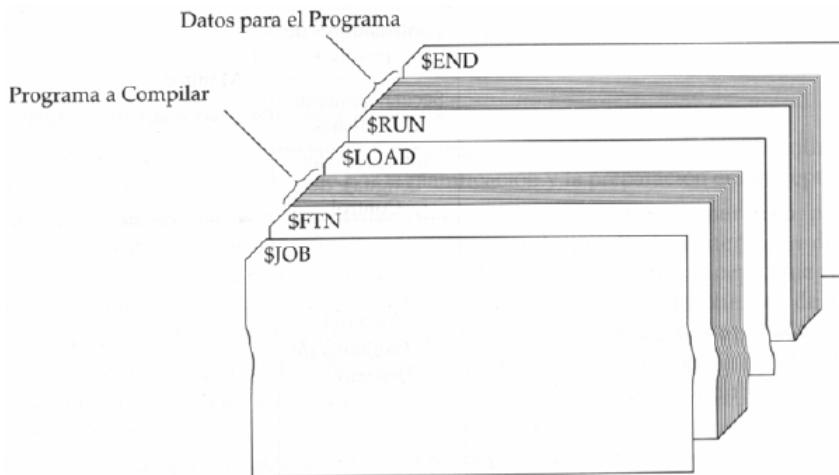


Figura 2.1: Tarjetas de un sistema por lotes

mador ya no debía escribir todas las instrucciones para acceder al hardware, sino que se le entregaba ya una herramienta para facilitar su trabajo.

Si ocurría algún error durante la ejecución del trabajo se entregaba al programador un *dump* de la memoria, de tal forma que este pudiera corregir el error en el programa y volver a entregar un conjunto de tarjetas corregidas para su ejecución. Esto evitaba que el programador tuviese que estar frente al computador para depurar su código.

A pesar de que el computador se encontraba más tiempo ocupado, porque siempre que hubiesen tarjetas se estarían procesando, su componente más costosa, la CPU, que es más rápida comparada con los dispositivos de entrada o salida de datos, no se encontraba necesariamente ocupada todo el tiempo. Las tareas de lectura y escritura son bloqueantes y mantienen a la CPU en un ciclo de *busy-waiting*. Lo anterior significa que mientras se está leyendo o escribiendo, la CPU no puede realizar otras tareas (ya que espera que la lectura o escritura termine).

2.1.3. Sistemas de operación *offline* (60's)

La idea principal tras las operaciones fuera de línea, o *offline*, corresponde a la lectura y escritura de los elementos utilizados en los sistemas operativos por lotes, o sea las tarjetas, de forma externa al computador principal. Lo anterior buscaba no utilizar el procesador más caro disponible en tareas lentas (como leer tarjetas), sino utilizar otras unidades que traspasaban las tarjetas perforadas a cintas y luego estas cintas eran cargadas en la máquina principal.

Análogamente a lo explicado, la salida del computador principal era generada en cintas magnéticas, las cuales en una etapa posterior a la ejecución del programa eran llevadas a un equipo de impresión donde se obtenía una salida que se entregaba al programador.

Con este sistema fuera de línea, se lograba un mejor rendimiento del procesador principal, utilizándolo para tareas de cómputo y dejando las tareas de lectura y escritura a otros equipos más económicos.

La cantidad de máquinas periféricas utilizadas dependerá de la capacidad del procesador, o sea, mientras existiera tiempo de CPU sin uso, se podían seguir agregando cintas al computador principal. Una vez el tiempo de CPU ya había alcanzado su uso constante, no tenía sentido agregar más impresoras (o lectoras) ya que no se generarían más cintas para imprimir por unidad de tiempo.

El trabajo solo entrega el control del sistema al monitor residente en caso de término, que se requiera algún servicio de entrada o salida o en caso de error.

A pesar de esta mejora en la velocidad de lectura y escritura, el procesador, mientras se realizan dichas operaciones, continúa estando ocioso, sigue existiendo problema de *busy-waiting* en cintas. Consideremos que para leer una línea de la cinta se tomaban 80 caracteres (misma cantidad que una tarjeta perforada) y se escribían de a 80 caracteres (en caso de cintas) y de a 132 caracteres en impresoras. El leer una línea implicaba hacer girar la cinta, lo cual generaba inercia en la cinta y hacía que hubiese que ajustarla (retrocediendo) para leer la próxima línea en el futuro.

Adicionalmente existía un mayor tiempo para los programadores, que debían esperar que sus programas fueran traspasados a cinta, ejecutados, luego los resultados traspasados a cinta e impresos.

2.1.4. Sistemas con *buffering* (60's)

Para ayudar a solucionar el problema de leer línea a línea las instrucciones de una cinta se empezaron a leer de a 10 líneas, o sea de a 800 caracteres, lo cual permitía disminuir los tiempos de lectura. En vez de leer de a una línea, se leían 10 considerando que en algún momento futuro esas líneas podrían ser utilizadas, las líneas leídas eran guardadas en un *buffer* a la espera de ser solicitadas. Por lo tanto cuando eventualmente el trabajo requería una línea, esta era leída desde el *buffer* y no desde la cinta, reduciendo los tiempos.

De la misma forma para escribir en la cinta, se guardaba en un *buffer* lo que se quisiera escribir, una vez lleno este *buffer* se escribía todo en la cinta de una vez.

Mediante el uso de canales, uno para la lectura y otro para la escritura, se podían conseguir mejoras en los tiempos y rendimiento de la CPU. Ya que la tarea de leer y llevar al *buffer* o sacar del *buffer* y escribir la realizaban los canales propiamente tal, y la CPU no era necesaria durante toda la operación de E/S, con lo que la misma podía ser utilizada para tareas de cómputo.

El rendimiento de esta técnica dependerá básicamente de si el proceso es intensivo en CPU, en E/S o es igual en ambos casos. A pesar de lo anterior el porcentaje de utilización de CPU, gracias al *buffering* y uso de canales aumentará.

El principal problema de esta técnica sigue siendo el tiempo de espera extra agregado al tener que leer las tarjetas y cargarlas en cintas y luego las cintas imprimirlas, esto básicamente por el transporte de un sistema periférico al computador principal.

2.1.5. Sistemas de operación *online* (60's)

En este tipo de sistemas la lectura de tarjetas e impresión de resultados ya no es realizada en equipos periféricos. Las lectoras e impresoras se conectan directamente al computador central y hacen uso de los canales de E/S que se agregaron en el sistema con *buffering*. Esto fue posible gracias a la aparición de discos duros los cuales contenían la entrada de los trabajos y almacenaban las salidas de los mismos.

En este tipo de sistemas el monitor residente es quien se encarga de leer tarjetas y dejarlas en el disco y de imprimir los resultados. Para ejecutar un trabajo debe haber sido leído completamente al disco, así mismo para imprimir los resultados de un trabajo este debe haber terminado su ejecución habiendo dejado la salida en el disco.

Esto mejora el tiempo de proceso de un trabajo, ya que no se deben utilizar lectores o impresoras externas al computador principal, ahorrando tiempo en el traspaso físico de las mismas de un equipo a otro.

El problema sigue siendo la ociosidad del procesador cuando se deben realizar operaciones de entrada o salida.

2.1.6. Sistemas multiprogramados (fines 60's)

El principal problema con un sistema de operación *online* es que la CPU no está siendo utilizada todo el tiempo, esto a pesar que pueden existir trabajos que pudiesen ser atendidos. Esto es básicamente porque no se permite la ejecución de más de un trabajo en “paralelo” y se debe esperar a que uno termine para iniciar otro.

En este tipo de sistemas se aprovecha el tiempo de E/S para ejecutar otros trabajos. Durante esta época aparece el concepto de planificación de procesos/trabajos (o *scheduling*) y con esto el concepto de Sistema Operativo. Ya que varios procesos deben ejecutarse, todos ellos deben estar residentes en memoria principal. La multiprogramación implica multiprocesos, o sea programas se ejecutan de forma “paralela”, ver figura 2.2.

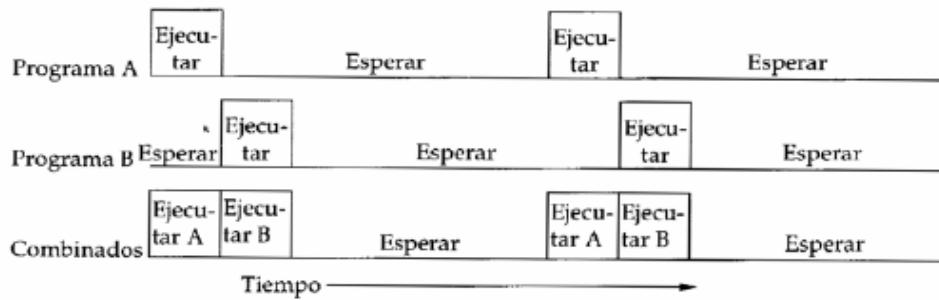


Figura 2.2: Sistemas por lotes vs sistema multiprogramado

El principal problema es la no existencia de un entorno protegido para la ejecución de los procesos, si un proceso cometía algún error podía hacer que todo el sistema fallase.

2.1.7. Máquinas o procesadores virtuales

En este tipo de máquinas, se da soporte para la emulación de varias máquinas o procesadores a partir de un solo procesador real. Cada máquina virtual entrega al proceso un entorno protegido, frente a otras máquinas virtuales, de tal forma que lo ejecutado en dicho entorno solo afecta a esa máquina virtual.

El tiempo de CPU utilizado por cada máquina es una tajada de tiempo del procesador real. Con esto se logra el mayor rendimiento del computador, utilizando multiprogramación, ya que la CPU siempre estará atendiendo a alguien que lo requiera.

El problema aquí comienza a ser la baja productividad de los programadores. Lo tiempos de ejecución total de un proceso sigue siendo alto, y la depuración de los trabajos comienza a tomar importancia como problema.

2.1.8. Sistemas de tiempo compartido (70's)

Los sistemas de tiempo compartido corresponden a entornos multiprogramados y multiusuario, los cuales entregaban un mejor tiempo de respuesta. Los usuarios (programadores) pueden trabajar de forma interactiva con los computadores mediante terminales conectadas

a ellos. O sea, vuelven a trabajar directamente con el computador, como lo hacían en los inicios, el operador ya no existe.

Cada programador dispone de una terminal con una consola, donde puede realizar una depuración de forma más rápida, no debiendo esperar la entrega de los resultados de la ejecución de su programa impresos.

En un sistema de tiempo compartido los usuarios comparten recursos, por lo cual se debe hacer un reparto equitativo de los mismos y además contar con sistemas de protección para ellos. Ahora se habla de sesiones de usuarios y no de trabajos.

El problema aquí surge con la cantidad de usuarios y procesos que estos ejecutan, donde el procesador no es capaz de atender a infinitos usuarios y el sistema puede ir degradándose con cada nuevo que entra. Esto ya que el recurso CPU que realiza los cómputos y el recurso memoria que se requiere para mantener los procesos en ejecución son limitados.

2.1.9. Computadores personales

Gracias al uso de componentes cada vez más pequeños se logró empezar a comercializar microprocesadores, los cuales permitían, por su bajo costo, que fuesen adquiridos por usuarios personales. De esta forma ya no se requería compartir el recurso CPU o memoria con múltiples usuarios, sino que cada usuario (o programador) disponía de un equipo dedicado a sus labores.

Originalmente los computadores personales, destinados al uso por parte de un único usuario, no requerían características de sistemas de tiempo compartido. Con esto su diseño era más simple y requerían menos soporte del hardware para su funcionamiento. Un ejemplo de este tipo de sistema operativo es MS-DOS.

Cada usuario posee su propia máquina, sin embargo el compartir datos entre máquinas resultaba un proceso complicado. Cada máquina debía tener su propia impresora y/o lectora de discos.

2.1.10. Redes de computadores personales (80's)

A mediados de los 80's surgen las redes de computadores personales, bajo esta modalidad los usuarios podían compartir discos e impresoras con otros equipos y de esta forma economizar en la compra de recursos.

Este esquema utiliza uno de los equipos como servidor de disco y/o impresora, y los demás computadores se conectan vía red a este. Los usuarios conectados a estas terminales, veían el disco o la impresora como si estuviese conectada en su equipo.

Ya que el sistema operativo utilizado, en realidad un monitor residente, no poseía características de protección es que era poco recomendable ejecutar aplicaciones de usuario en el servidor, ya que la caída de la aplicación podría hacer que todo el sistema fallase.

Las primeras redes solo permitían compartir directorios en modo solo lectura.

2.1.11. Sistemas en tiempo real

Este tipo de sistemas corresponde a los utilizados en aplicaciones de tiempo real, donde los tiempos de respuesta a eventos del mundo físico son críticos. Por ejemplo el uso en control de tráfico o procesos industriales.

Deben poseer tiempo de respuestas muy rápidos, para esto es requisito que los procesos residan permanentemente en memoria principal. Adicionalmente cualquier interrupción debe ser atendida inmediatamente. No se garantiza que se reciba de forma inmediata, pero si en un tiempo muy acotado.

Existen variantes de GNU/Linux que están orientadas a sistemas operativos en tiempo real.

2.1.12. Sistemas distribuídos

Corresponden a un conjunto de estaciones de trabajo o *terminales inteligentes* conectadas entre sí para trabajar de manera conjunta y como una sola. Este modo de operación ha sido

influenciado por el decaimiento del costo de los procesadores, donde es más barato tener dos CPU funcionando conjuntamente que una sola CPU del doble de velocidad.

Se hace uso de las redes de computadores, donde cada nodo de la red es una pieza del computador conformado. Con esto se consigue una serie de ventajas tales como alto rendimiento, alta disponibilidad, balanceo de carga y escalabilidad.

2.1.13. Sistemas multiprocesadores

Un sistema con múltiples procesadores permite la ejecución real en paralelo de al menos dos procesos, considerando que como mínimo existirán dos procesadores. En estricto rigor, y por definición de Intel, son *cores* o (núcleos), donde un procesador puede tener uno o más *cores*. Quedando el concepto de procesador o CPU como el chip y *core* como el componente de la CPU que ejecuta los procesos.

Cuando el número de procesos en ejecución supera el número de *cores* se debe recurrir al uso de algún mecanismo de planificación de procesos, donde se deberá, al igual que en sistema monoprocesador, compartir el tiempo de CPU entre los interesados.

Se hablará de tiempo de CPU durante el texto, pero recordar que nos estaremos refiriendo a los *cores* que están en la CPU.

2.2. Tendencias últimas dos décadas

Durante las últimas dos décadas, o sea desde los 90's, han existido diversas tendencias en lo referente al desarrollo de sistemas operativos.

El gran crecimiento que han experimentado las redes computacionales junto a las velocidades de acceso a Internet han permitido un mayor uso de computación distribuida, mediante el uso de plataformas multiprocesadoras y procesadores conectados en red.

El área de sistemas multimedia, datos más sonido más imágenes ha experimentado un alto desarrollo. Se están desarrollando cada vez dispositivos de entrada más rápidos y eficientes

como los sistemas de reconocimiento automático de voz o imágenes. Dichos sistemas tienen directa relación con los mencionados como sistemas de tiempo real.

Adicionalmente la tendencia va hacia el diseño e implementación de sistemas abiertos, tales como:

- Normas de comunicación abiertas, como el modelo de referencia OSI.
- Normas de Sistemas Operativos abiertos como GNU/Linux.
- Normas de interfaces de usuario abiertas, como el sistema de ventanas X desarrollado por MIT.
- Normas de aplicaciones de usuario abiertas, como las entregadas por la FSF¹.

2.3. Logros

Durante los años de desarrollo se han obtenido diferentes logros, que perduran en los sistemas hasta hoy en día. A continuación se mencionan brevemente estos, en capítulos posteriores se discutirá en detalle cada uno de ellos.

2.3.1. Procesos y memoria

Un proceso corresponde, en principio, a cualquier programa en ejecución. Este posee diversos estados, donde lo más común es encontrar: ejecución (proceso en cpu), bloqueado (en espera de un recurso) y listo (esperando entrar a cpu).

Cualquier proceso requerirá si o si al menos el uso de memoria principal y CPU, adicionalmente puede requerir utilizar otros dispositivos, en general cualquiera destinado a operaciones de entrada y salida. Esto implicará que diversos procesos podrán tratar de acceder a un mismo recurso al mismo tiempo, por lo cual existirá competencia por dicho recurso. Para esto,

¹Free Software Foundation / <http://www.gnu.org/licenses/gpl.html>

a lo largo de los años, se han diseñado diversos algoritmos que permiten al sistema operativo decidir que proceso utilizará que recurso.

Además un proceso para funcionar requerirá algo más que su código, un proceso estará formado por el programa o código, sus datos y un contexto (o descriptor del proceso).

Finalmente el sistema operativo debe ser capaz de prevenir o mitigar los problemas más comunes correspondientes a *data races*, *deadlock* y *starvation*. Para esto, existen mecanismos de sincronización que se pueden utilizar.

2.3.2. Seguridad y protección

Se debe garantizar la protección de los procesos en ejecución, se mencionó ya que sistemas operativos de tiempo compartido debían proteger a los procesos corriendo, ya que múltiples usuarios podrían estar trabajando en la máquina. Específicamente se deben implementar políticas que permitan controlar el acceso a un recurso solicitado por más de un proceso, a este recurso se le conocerá como **sección crítica** y algunas medidas que se pueden tomar son:

- No compartición: procesos se encuentran aislados.
- Compartición solo como lectura, para escribir un recurso se requieren mecanismos (o condiciones) especiales.
- Subsistemas confinados: similar a una protección por ocultación donde un proceso evita que otros sepan como opera.
- Diseminación controlada: en este caso existen credenciales de seguridad para acceder a los recursos, por lo cual se especifica quien podrá y quien no podrá acceder al recurso.

2.3.3. Gestión de recursos

La gestión de recursos corresponde a como se deberán asignar los recursos a un proceso que los solicite, considerando para esto que deberá existir algún tipo de planificación que determine el orden en que serán atendidas las solicitudes. Se deben considerar los factores:

- Equidad: igualdad de preferencias frente a una solicitud.
- Sensibilidad: poder priorizar ciertos procesos.
- Eficiencia: maximizar la productividad y minimizar los tiempos de respuestas.

Más adelante se hablará de la planificación de CPU y como el sistema operativo asigna este recurso a un proceso, se deberá considerar que conceptos mencionados para la CPU son análogos a los utilizados en la planificación de otro tipo de recursos.

2.3.4. Estructura del sistema

La estructura o arquitectura del sistema, determinará como se comportará y que capacidades podrá el sistema operativo entregar a los procesos y usuarios que están ejecutándose sobre el.

Es importante mencionar que la estructura del software utilizada dentro del sistema operativo puede afectar considerablemente el funcionamiento de este. No será lo mismo una rutina programada de cierta forma que de otra, una puede ser más o menos eficiente dependiendo de la implementación realizada. Así mismo un sistema con más o menos instrucciones no significa que sea un sistema más o menos eficiente, ni mucho menos más o menos simple. Ya se habrá visto en lenguajes de programación que existen instrucciones que utilizan muy pocas líneas, sin embargo son difícilmente entendibles.

Se deberá dividir el sistema operativo, de tal forma que cada una de las partes de este cumpla una función específica. Si bien se puede tener un único sistema que implemente todas las funcionalidades (este es el caso de un sistema operativo monolítico), aun así internamente

deberá estar organizado de tal forma que sea sencillo de mantener y programar. De no realizarse lo anterior de forma correcta podrían existir problemas con los tiempos de entrega del software, fallos y rendimiento en el momento de poner en funcionamiento un nuevo sistema.

El capítulo 3 discute los conceptos de la estructura del sistema operativo en un nivel más profundo.

2.4. Ejercicios y preguntas

1. ¿Cuándo es recomendable el sistema operativo por lotes?.
2. Describa las ventajas de un sistema de operación *offline* versus un sistema operativo por lotes.
3. ¿Cuál es el problema de los sistemas de operación *offline* que se soluciona en uno con *buffering*?.
4. Indique la característica que hace a un sistema multiprogramado ser más eficiente que sus predecesores.
5. En un sistema de tiempo compartido ¿cuántos usuarios pueden correr sus programas al mismo tiempo?.
6. ¿Cuál es la principal característica de un sistema operativo en tiempo real?.
7. ¿Quién inicio el proyecto GNU?.
8. ¿Quién inicio el proyecto Linux?.
9. ¿Cuáles son las 4 libertades que entrega el software libre?.
10. ¿Qué es un proceso?.
11. Un proceso ¿es solo código?.

12. ¿Qué se conoce como sección crítica?.
13. Explique los conceptos de equidad, sensibilidad y eficiencia.

2.5. Referencias

- Sistemas Operativos, Segunda Edición, Andrew Tanenbaum, Capítulo 1.2.
- Sistemas Operativos, Quinta Edición, Abraham Silberschatz y Peter Baer Galvin, Capítulo 1.
- Sistemas Operativos, Segunda Edición, William Stallings, Capítulo 2.

Capítulo 3

Estructura y diseño

El sistema operativo es el encargado de ofrecer diferentes servicios, tanto al usuario como a otros procesos. Es importante mencionar que aquí se contrapondrán los objetivos que tiene el sistema con los requerimientos de los usuarios.

A continuación se listan una serie de servicios que deben ser considerados al momento de diseñar un sistema operativo, algunos de los cuales se discutirán en mayor detalle más adelante.

- a. **Interfaz de usuario:** servicio que entrega un método para que el usuario pueda interactuar con el sistema operativo, ya sea una CLI o una GUI.
- b. **Ejecución de programas:** el sistema operativo se debe encargar de mantener a los procesos en ejecución durante todo su ciclo de vida, esto implica la administración de los mismos durante sus posibles estados de ejecución.
- c. **Operaciones de entrada/salida (E/S):** un proceso no podrá acceder directamente a los recursos disponibles en la máquina, debe ser el sistema operativo quien, mediante una interfaz de acceso, permita a los diferentes procesos acceder a los dispositivos de entrada y salida de forma concurrente y controlada.

- d. **Sistema de archivos:** se debe proveer de una forma de acceder al disco con alguna estructura, donde no se deban escribir directamente posiciones de memoria, sino que los procesos puedan escribir y leer archivos dentro de los dispositivos.
- e. **Comunicación entre procesos (IPC o *Inter Process Communication*):** corresponde al mecanismo que permite que diferentes procesos se comuniquen entre sí, por ejemplo mediante el uso de memoria compartida, *sockets* o tuberías.
- f. **Detección de errores:** sistema deberá capturar los errores, tanto físicos como lógicos que un proceso pueda generar y evitar que dicho error afecte a otros procesos en ejecución.
- g. **Asignación de recursos:** los diferentes dispositivos en la máquina podrán ser utilizados concurrentemente por muchos procesos, por lo que deberá existir algún algoritmo que permita planificar quien utilizará un recurso en un momento dado.
- h. **Estadísticas:** estas son llevadas con propósitos contables, para detectar errores o para, por ejemplo, predecir el comportamiento futuro de un proceso y poder tomar decisiones de planificación al respecto.
- i. **Protección y seguridad:** el acceso a los recursos disponibles debe ser controlado, se debe evitar que cualquier proceso pueda utilizar cualquier dispositivo, en cualquier momento.

3.1. Interfaz de usuario

Las **interfaces de usuario** permiten al usuario realizar una interacción con el sistema operativo, se dividen básicamente en dos tipos ***Command Line Interface*** (CLI) y ***Graphical User Interface*** (GUI).

3.1.1. CLI

La interfaz de línea de comando, o simplemente la *shell*, corresponde a un intérprete en modo texto que permite introducir órdenes para que sean ejecutadas por el sistema operativo. Su tarea principal es recibir las solicitudes del usuarios, y en la mayoría de los casos ejecutar un programa asociado a dicha solicitud.

Algunos ejemplos de *shells* conocidas en diferentes sistemas operativos *like Unix* son:

- sh: Steve Bourne, Unix v7, 1978.
- ash: usada como base para las shell de BSD.
- bash: parte del proyecto GNU.
- dash: ash mejorada para Debian GNU/Linux.

La *shell* ejecutará los comandos que el usuario introduzca, algunos de ellos serán comandos básicos (como listar directorios, crear una carpeta, ver la fecha) o podrían ser programas más complejos (como un editor de texto o una aplicación en modo texto). Adicionalmente se puede utilizar un lenguaje de programación para realizar *scripts*, donde existe un estándar denominado *shell scripting*, sin embargo cada intérprete puede implementar extensiones para el mismo.

Un comando al ser ejecutado deberá ser buscado dentro del PATH del sistema, el cual corresponde a la ruta de directorios donde posiblemente se podría encontrar dicho comando, si luego de revisar todos los directorios del PATH el comando no es encontrado se informa al usuario. En caso de ser encontrado el comando puede estar implementado como un programa externo de la *shell* o como un programa dentro de la *shell*, como el caso de algunas extensiones. La ventaja de utilizar el primer método, fuera del intérprete, es que no se debe modificar este para agregar nuevos comandos, bastará agregarlos a alguna de las rutas en el PATH.

En general la shell de un usuario no privilegiado (usuario normal) tendrá el signo \$ y un usuario privilegiado (o sea, el usuario root) tendrá el signo # para indicar que está en espera

del ingreso de comandos. De esta forma si vemos un comando precedido por un signo \$, dicho comando se ejecuta como usuario sin privilegios, en cambio si vemos un comando precedido por un signo # necesariamente dicho comando se debe ejecutar con el usuario root.

3.1.2. GUI

La interfaz de usuario gráfica corresponde al entorno de ventanas, el cual permite tener diversas aplicaciones encapsuladas dentro de un cuadro (ventana) y de esta forma compartir de manera fácil un único recurso, la pantalla, con múltiples procesos que quieren dibujar en ella. En sistemas *like Unix* es conocido como X en honor a Xerox que lo ideo en los años 70s¹.

Algunos entornos de escritorio y gestores de ventanas son KDE, Gnome, XFCE, Lxde, Fluxbox y OpenBox. Algunos de estos pueden apreciarse en la figura 3.1.

El entorno gráfico no es propiamente una función del sistema operativo, de hecho es una aplicación más que funciona sobre este, la cual entrega una forma “más amigable” de interactuar con el sistema.

¹¡Si! mucho antes que Microsoft Windows empezara a usarlas



Figura 3.1: Diversos entornos gráficos

3.2. API y llamadas al sistema

Las **llamadas al sistema** corresponden a una interfaz para utilizar los servicios del sistema operativo, algunos ejemplos de estos son:

- Errores de procesos (hardware o software).
- Lectura, creación o borrado de archivos.
- Imprimir texto por pantalla.
- Acceso a dispositivos de E/S.

Una **API** o *Application Program Interface* corresponde al conjunto de instrucciones y procedimientos que se ofrecen como biblioteca. En el caso del sistema operativo, es la biblioteca que entrega las funciones que permiten hacer uso de las llamadas al sistema.

La API es dependiente del sistema operativo, y algunos ejemplos de estas son la API POSIX², la API Win32 y la API de Java.

La principal ventaja de utilizar una API para el desarrollo de aplicaciones tienen que ver con la abstracción que el programador realiza del sistema, donde no necesita conocer a fondo el mismo y puede generar una menor cantidad de código e instrucciones más simples. Lo anterior implica una mayor facilidad al momento de portar el código desde un sistema operativo (o máquina) a otro(a).

3.2.1. Tipos de llamadas al sistema

Las llamadas a sistemas se pueden dividir en cinco grupos principales, los cuales corresponden a control de procesos, manipulación de archivos, manipulación de dispositivos, mantenimiento de información y comunicaciones. Estas serán discutidas a continuación.

²Portable Operating System Interface, utilizada en sistemas *like Unix*

3.2.1.1. Control de procesos

Estas llamadas a sistema se encargan de diferentes tareas que tienen relación con los estados y vida de un proceso.

Por ejemplo se debe manejar el **término** donde en caso de errores se podrá producir un volcado de memoria y el programador deberá proceder a depurar el programa, por ejemplo utilizando una herramienta como gdb. En el caso de término el sistema operativo deberá pasar a la siguiente tarea a realizar, generalmente planificando un nuevo proceso. En caso del retorno de salida, existen distintos niveles de error, donde el estándar es que 0 corresponde a un retorno normal y cualquier valor positivo a un error, donde entre más alto el número más grave debiese ser el error.

También se deben manejar temas relacionados con la **carga y ejecución** del proceso, donde puede ser necesario cargar y/o ejecutar otro programa, por ejemplo cuando un proceso *A* llama a un proceso *B*. Una vez termina la ejecución del proceso *B* el control debería volver al proceso *A*. Esto último se observa claramente al ejecutar un comando en un intérprete, ya que al ejecutar, por ejemplo, el comando *ls* cuando este termine el control volverá al intérprete.

Otra llamada al sistema tiene relación con los **atributos de procesos**, donde el sistema operativo deberá obtener y fijar los mismos, como prioridad o tiempo máximo de ejecución del proceso. **Tiempos de espera** los cuales son determinados por un tiempo *X* de espera o bien por la espera de algún suceso que se requiera. O llamadas al sistema para la asignación de **memoria principal**.

La llamada al sistema ***kill*** permite enviar señales a los procesos. Estas señales envían una instrucción al proceso con diferentes objetivos, por ejemplo para matar el proceso (KILL), terminar el proceso (TERM), suspender el proceso (STOP) o ejecutar una alarma (ALRM). Para enviar una señal en sistemas *like Unix* se utiliza el comando *kill*.

Algunos ejemplos de llamadas al sistema relacionadas son **fork** (crea un proceso hijo), **exec** (carga programa en memoria y ejecuta), **wait** (espera hasta la finalización del proceso

hijo) y `exit` (termina la ejecución del proceso).

3.2.1.2. Manipulación de archivos

Las llamadas a sistema relacionadas con la manipulación de archivos tienen principalmente las funciones de realizar **operaciones básicas sobre archivos y determinar atributos o cambiarlos**, como el nombre el tipo del archivo, los permisos que tiene un usuario, etc.

Algunos ejemplos de llamadas al sistema relacionadas son `create`, `delete`, `open`, `read`, `write`, `reposition` y `close`.

3.2.1.3. Manipulación de dispositivos

Permiten controlar el acceso a los dispositivos, el cual debe ser controlado, si un proceso requiere un recurso y este está ocupado el proceso deberá esperar por el recurso.

Los dispositivos que se deberán manipular pueden ser tanto físicos, como el disco duro, y virtuales, como archivos. En sistemas **like Unix** los dispositivos pueden ser encontrados en el directorio `/dev`.

Algunos ejemplos de llamadas al sistema relacionadas son `request`, `release`, `open`, `close`, `read`, `write` y `reposition`.

3.2.1.4. Mantenimiento de información

El propósito de estas llamadas al sistema es transferir datos entre el programa de usuario y el sistema operativo. Ejemplos de estos tipos de datos son tiempo, usuarios, versión S.O., memoria libre (o disco duro), etc. Información general del funcionamiento del sistema operativo puede ser encontrada, en sistemas **like Unix**, en el directorio `/proc`.

Algunos ejemplos de llamadas al sistema relacionadas son `time`, `date` y `sysinfo` (usada por comando `uname`).

3.2.1.5. Comunicaciones

En este tipo de llamadas al sistema se incluyen aquellas que permiten realizar la comunicación entre procesos, por ejemplo mediante el **modelo por paso de mensajes**, usando sockets o el **modelo de memoria compartida**, donde se debe eliminar la restricción del sistema operativo que protege datos en memoria en el caso de proceso pesados.

Algunos ejemplos de llamadas al sistema relacionadas son `get_hostid`, `get_processid`, `open`, `close`, `accept_connection`, `wait_for_connection`, `read_message`, `write_message`, `shared_memory_create` y `shared_memory_attach`.

3.3. Diseño

Durante el diseño del sistema operativo se deberá considerar que los dispositivos sean mapeados en la memoria del computador como si fuesen posiciones en ella, si se lee en dicha dirección de memoria, en el fondo, se accede al dispositivo en sí, análogamente si se escribe en esa dirección de memoria se hará escritura en el disco. Esto es básicamente lo que sucede con los ficheros que se encuentran en `/dev` que representan dispositivos físicos de la máquina.

Sigamos con el ejemplo del disco, una vez terminada la ejecución de la rutina llamada para realizar la lectura no significa que se haya realmente terminado de leer desde el disco. En realidad la rutina que se ejecuta es el comando que se introduce para que se inicie la verdadera lectura y el disco tiene su propio microcontrolador que se encarga de realizar la operación. La CPU consultará entonces reiteradamente para verificar si se completo o no la lectura en un ciclo conocido como *busy-waiting*, esto es lo que sucedía originalmente en los primeros sistemas operativos como ya fue discutido anteriormente en el capítulo 2. El problema de este enfoque es que se pierde tiempo mientras se realiza la operación de lectura, alrededor de 10 [ms], donde no se hace otro trabajo útil.

Mucho mejor podría ser ejecutar otros procesos, mientras se espera que se lea el disco, y se obtengan los datos que requiere el proceso para continuar. En este caso el proceso quedará

bloqueado y deberá esperar a que el sistema operativo le notifique que los datos solicitados ya se encuentran listos para su uso.

El uso de interrupciones permite al disco avisar a la CPU que la operación en disco terminó, se suspende al proceso que está actualmente en la CPU y se ejecuta una rutina para atender la interrupción. Esta rutina de atención informa al proceso que lo solicitado del disco ya está disponible y se pasa el proceso a un estado listo para esperar a ser planificado nuevamente.

Existe dentro del núcleo del sistema operativo un vector de interrupciones con todas las posibles fuentes de las mismas, como de disco o las del **cronómetro regresivo**.

Un proceso que se está ejecutando podría acaparar la CPU, entonces el sistema operativo utiliza la interrupción del cronómetro regresivo para interrumpir al proceso que se está ejecutando, por ejemplo después de 10 o 100 [ms], y asignar la CPU a otro proceso, con este mecanismo se implementan las **tajadas de tiempo**. Si estas son suficientemente pequeñas el usuario tendrá la sensación que todo avanza al mismo tiempo, o sea, “ejecución en paralelo”.

Otros ejemplos de interrupciones son aquellas que ocurren al hacer divisiones por 0 o la ejecución de instrucciones ilegales (código de operación indefinida, operación que no existe en el procesador o operación privilegiada).

No confundir interrupciones con señales, estas son “interrupciones virtuales” y su ámbito es en los procesos. Cada proceso maneja su propio cronómetro regresivo virtual. El núcleo tiene una agenda con todas las señales que debe generar y revisa cual es la próxima que debe ocurrir y entonces el cronómetro regresivo coloca la alarma a dicha señal que se requiere para dicho proceso. Mandar una señal a un proceso implica activar el proceso para que este pueda atender la señal.

Otro aspecto a considerar en el diseño del sistema operativo son los canales que se utilizan para acelerar la entrada y salida de datos, que pueden ayudar a transferir muy rápido los datos. Lo anterior se logra utilizando un mecanismo del mismo hardware que permite hacer una transferencia directa entre dispositivos y memoria. Una vez terminada la transferencia

se genera una interrupción que indica que los datos ya están en la memoria. Con lo anterior el núcleo evita tener que mover los datos desde el dispositivo a la memoria, esta tarea la realiza el canal. Estos canales son conocidos como DMA o *Direct Memory Access*, donde el dispositivo, a través de este mecanismo, accede directamente a la memoria.

3.3.1. Objetivos

Se deben definir objetivos y especificaciones, por ejemplo el hardware que se requerirá y el tipo de sistema operativo que se desea implementar. Estos se dividirán en objetivos del usuario y objetivos del sistema.

El **usuario** está preocupado por que el sistema operativo sea cómodo de utilizar, fácil de aprender y usar, fiable, seguro y rápido.

El **sistema** está preocupado por que el sistema operativo sea flexible, fiable, libre de errores, eficiente, fácil de diseñar, implementar y mantener.

Los objetivos tanto de usuario como de sistema a veces pueden no ser compatibles, por ejemplo, para ser muy eficiente, quizás se deba sacrificar usabilidad. Por lo anterior es que se deberá encontrar un equilibrio entre los objetivos de ambos lados.

3.3.2. Políticas y mecanismos

Se deben definir **políticas** que indicarán **¿qué hacer?** y **mecanismos** que indicarán **¿cómo hacerlo?**. Es recomendable que políticas y mecanismos se encuentren separados, esto permitirá tener una mayor flexibilidad ya que si se desea modificar una se puede minimizar el impacto en la otra.

Las políticas determinarán todas las decisiones que el sistema operativo debe tomar. Por ejemplo si se debe o no asignar un recurso, deberá existir una política que indique cuando se aceptará la asignación y cuando se rechazará. Asociado a esta política debe ir un mecanismo que indique como hacer la asignación o como indicar el rechazo.

3.3.3. Requerimientos para protección de procesos

La **protección de procesos** significa que un proceso no debería interferir con otros procesos, un proceso que esta corriendo no debería poder acceder a los datos que otro esta manipulando. Ejemplos de estos sistemas operativos son Unix y Windows NT, y los derivados de ambos.

3.3.3.1. Modo dual

En esta forma de operación se deben implementar dos modos básicos en que el sistema operativo debe funcionar. El primero corresponde al ***user mode***, o modo usuario o no privilegiado, en donde se ejecutan todos los procesos, incluyendo aquellos que son ejecutados por el usuario root. Este modo tiene ciertas restricciones que impiden que un usuario pueda ejecutar cualquier instrucción o código en la máquina, por ejemplo la instrucción que permite deshabilitar las interrupciones. Si este modo no existiese un proceso cualquiera podría desactivar, por ejemplo, el cronómetro regresivo, y evitar que otros ocupen la CPU. En este modo, dicha instrucción es privilegiada, por lo cual al ejecutarse ocurriría una interrupción de software (interrupción de comando ilegal).

Es importante notar que a pesar de estar en modo usuario uno si podría desactivar las señales en procesos (no ignorar, desactivar), excepto la señal KILL. Si el proceso recibe una señal, la interrupción asociada ocurrirá pero el sistema operativo no la entregará al proceso hasta que estén habilitadas nuevamente. Importante mencionar que solo se desactivan señales de ese proceso, ya que al ser modo usuario un proceso no puede interferir con otro.

El otro modo corresponde al ***kernel mode***, modo sistema, modo supervisor o modo privilegiado. En este modo todo es permitido, por ejemplo aquí si se podrían desactivar las interrupciones. El núcleo es el único que corre sobre modo sistema, incluyendo sus módulos. Importante mencionar que dentro del núcleo no hay *segmentation fault*, en caso de existir algún error podría derivar en un *kernel panic*, es por esta razón que solo el usuario root puede cargar módulos al núcleo.

En la figura 3.2 se ilustra cada una de las partes que están involucradas en el modo usuario y el modo sistema en GNU/Linux. Las aplicaciones de los usuarios y la API glibc corren sobre el modo usuario, mientras que las llamadas a sistema, el núcleo y las instrucciones directas al hardware lo hacen en modo sistema. Si un usuario requiere hacer uso de una llamada a sistema deberá hacerlo a través de la API correspondiente, entonces el sistema operativo concederá solo por la ejecución de esa parte del código acceso a modo sistema, revocándose una vez la instrucción termine y siguiendo su ejecución en modo usuario. Es importante destacar que un usuario normal no puede acceder llamadas de sistema privilegiadas, se requiere ser usuario root para esto.

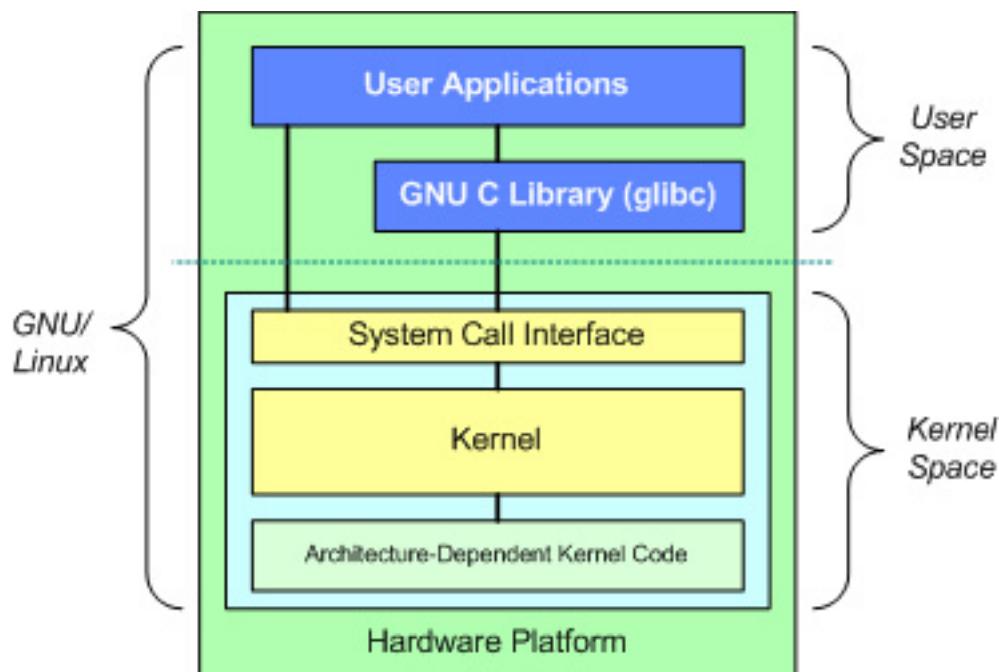


Figura 3.2: Componentes presentes en el modo usuario y el modo sistema

3.3.3.2. Unidad de gestión de memoria

La **MMU** o *Memory Management Unit* básicamente lo que hace es traducir de direcciones virtuales a direcciones reales. Las direcciones utilizadas por los procesos son virtuales, varios pueden usar la dirección 0x3A, pero cuando se mapean a la memoria real se mapean a direcciones físicas distintas, esto lo hace la MMU. Esto será cubierto en detalle en el capítulo 7.

Los primeros procesadores para computadores personales que aparecieron con MMU fueron los x86, a mediados de los años 80s. Antes ya habían equipos con MMU, pero eran los *mainframes*³, ya que estos eran para sistemas multiusuario. Como ejemplo de sistemas operativos que la requieren esta Unix que al ser multiusuario necesita MMU, Linux es derivado de Unix por lo que también la requiere y Android al ser derivado de Linux igualmente la necesita.

3.4. Estructura del sistema operativo

Se deberá elegir un método para estructurar las funcionalidades que se proveerán. Actualmente los sistemas operativos se encuentran divididos por jerarquías con funciones bien definidas. Se mencionarán algunas formas de estructurar el sistema a continuación.

3.4.1. Estructura simple

La estructura simple está orientada a sistemas operativos pequeños, simples y a su vez limitados.

Por ejemplo, MS-DOS entregaba máximas funcionalidades en un tamaño reducido, no poseía una división cuidadosa de sus módulos. Adicionalmente dicho sistema operativo entregaba acceso directo a rutinas que podían utilizar el hardware, por lo cual no se considera un sistema operativo con protección de sus procesos.

³Computadoras grandes, potentes y costosas utilizados en grandes instituciones

En el caso del Unix original, el kernel a través de las llamadas al sistema provee las funcionalidades necesarias para acceder a los recursos.

3.4.2. Estructura en niveles

En este tipo de estructuras se utiliza un método de diseño arriba-abajo, el sistema resultante corresponderá a un sistema por niveles donde la estructura jerárquica se determinará de acuerdo a la complejidad de las funciones de cada nivel.

Las ventajas de utilizar esta estructura radica en la independencia que se conseguirá entre los niveles, ya que cada uno se encargará de una tarea específica que le entregará servicios a otro nivel. Se debe preocupar mantener las mismas funcionalidades que se entregan a otras capas, no importando como se cambie esto internamente. Esto proporciona facilidad en la construcción, mantención y depuración del sistema operativo.

Se debe tener especial cuidado en la definición apropiada de los diferentes niveles, donde esto debe hacerse de forma correcta para lograr la independencia anteriormente mencionada. Además se debe considerar que ciertas capas podrán depender de otras para operar. Una desventaja es que al introducir niveles la operación total podría resultar un poco más lenta, ya que se deben utilizar interfaces entre las diferentes capas del sistema.

A continuación se muestra un ejemplo de los posibles niveles de jerarquía para un sistema operativo. Notar los niveles del 1 al 4 no corresponden directamente a funciones del sistema operativo, estas son realizadas por hardware. También observar que las capas superiores requerirán servicios de capas inferiores como es el caso del nivel de directorios que requiere servicios de la capa sistema de archivos y esta a su vez de la capa de almacenamiento secundario.

1. Circuitos electrónicos: registros, puertas, buses.
2. Instrucciones: evaluación de la pila, microprogramas, vectores de datos.
3. Procedimientos: pila de llamadas, visualización.

4. Interrupciones: manejo de interrupciones del hardware.
5. Procesos primitivos: semáforos, colas de procesos.
6. Almacenamiento secundario: bloques de datos.
7. Memoria virtual: paginación.
8. Comunicaciones: tuberías.
9. Sistema de archivos: almacenamiento en disco duro u otro medio.
10. Dispositivos: impresoras, pantallas, teclados.
11. Directorios: árbol de directorios.
12. Procesos de usuario: programas en ejecución.
13. Shell: intérprete de comandos.

3.4.3. *Microkernels*

Un sistema operativo que está organizado como micro núcleo entrega solo las tareas básicas, como: planificación de procesos, gestor de memoria y comunicaciones. Otras tareas son realizadas por programas del sistema operativo y el núcleo es utilizado como un intermediario para la comunicación entre el usuario y los programas del sistema operativo que ofrecen los servicios.

Los programas nuevos para el sistema operativo son añadidos al espacio del usuario, se ejecutan en modo usuario y no como modo sistema. El núcleo entonces se encarga de realizar las llamadas al sistema a través de mensajes hacia los servicios correspondientes que entregan las funcionalidades solicitadas.

Su ventaja es que al incorporar las mínimas funcionalidades, son más estable. Sin embargo la principal desventaja en este tipo de núcleos es que son ineficientes al tener que realizar muchos cambios de contexto para ir a los servicios prestados.

Minix es un ejemplo de este tipo de sistema operativo.

3.4.4. Módulos

En este caso el sistema operativo está compuesto por módulos, donde lo fundamental se encuentra en el núcleo en si, pero otras funcionalidades son entregadas como módulos del núcleo. Ambos, tanto el núcleo como los módulos corren en modo sistema.

Esto permite que componentes sean cargados dinámicamente al núcleo, evitando tener que disponer del soporte para todos los dispositivos o funcionalidades permanentemente cargados en memoria principal. En Linux esto se puede realizar mediante el uso de las instrucciones `lsmod`, `modprobe` y `rmmmod`.

Algunos ejemplos de módulos que pueden existir son controladores de disco, controladores de tarjetas de red o el soporte para IPv6. Es importante mencionar que el soporte necesario para que la máquina pueda ser arrancada, en estricto rigor para que el disco duro que contiene el sistema raíz del sistema operativo sea abierto, no puede ir como módulo del núcleo. Lo anterior ya que los módulos se cargan cuando el sistema esta iniciando, una vez que ya se montó el sistema de archivos.

Ejemplos de estos sistemas operativos son Unix modernos, Solaris, Linux y Mac OSX.

Se hablará más adelante de módulos en Linux en el capítulo ??.

3.5. Implementación

Una vez se decide la estructura del sistema operativo y están definidas las políticas del mismo se debe realizar la implementación. Originalmente esto se realizaba programando el hardware de la máquina, posteriormente se utilizaba un lenguaje de bajo nivel o lenguaje de

máquina (*assembler*) y actualmente se utilizan lenguajes de alto nivel (como C o C++).

La principal ventaja de utilizar lenguajes de alto nivel radica en que es fácil de programar, el código que se escribe es compacto, fácil de entender y depurar. Adicionalmente las mejoras introducidas en los compiladores significarán mejoras en el código generado, por lo tanto mejoras en el sistema operativo que se está compilando. Finalmente colabora con la portabilidad de un sistema operativo de un hardware a otro, recordar que será la API de cada lenguaje la que se encargará de traducir las instrucciones a la arquitectura seleccionada.

Desde el punto de vista de la optimización del sistema operativo es recomendable atacar a las estructuras de datos y algoritmos utilizados en tareas críticas, tales como el planificador de la CPU y el gestor de memoria. Una vez identificados los problemas se deben optimizar, por ejemplo reemplazando el código de alto nivel por código de máquina.

3.6. Ejercicios y preguntas

1. Mencione y explique 5 servicios que se deben considerar al momento de diseñar el sistema operativo.
2. ¿Cuál es la diferencia entre CLI y GUI?.
3. Mencione tres ventajas de utilizar una API.
4. ¿Qué es una llamada al sistema?, de dos ejemplos (que no sea kill).
5. La llamada sistema kill permite enviar señales a procesos, indique la diferencia entre la señal KILL y TERM.
6. ¿Por qué *busy-waiting* es ineficiente?.
7. ¿Quién atiende las interrupciones?.
8. ¿Para qué es utilizado el cronómetro regresivo?.

9. Explique la diferencia entre interrupciones y señales.
10. Explique el concepto de modo dual, o sea, explique los modos usuario y sistema. Adicionalmente de ejemplos de cuando se utiliza cada uno de ellos.
11. ¿Cuándo ocurre un cambio de modo usuario a modo sistema?.
12. ¿Una aplicación puede ejecutar directamente una llamada al sistema sin utilizar una API?
13. Explique la estructura de núcleo monolítico.
14. Explique la estructura de *microkernels*.
15. Linux ¿a que tipo de estructura de sistema operativo corresponde?.
16. ¿Cuál es la ventaja de utilizar un sistema con estructura modular?.

3.7. Referencias

- Sistemas Operativos, Segunda Edición, Andrew Tanenbaum, Capítulo 1.3, 1.4 y 1.5.
- Sistemas Operativos, Quinta Edición, Abraham Silberschatz y Peter Baer Galvin, Capítulo 3.

Capítulo 4

Procesos

La definición más simple para describir un proceso corresponde a un **programa en ejecución**. Es importante notar que el proceso no es solo el código, sino que es el código más los datos que conforman al proceso, su pila, registros del procesador, descriptores de E/S, etc, en general cualquier dato que permita administrar el proceso. Veremos que esto último es conocido como contexto del proceso.

Adicionalmente se debe considerar que un proceso para ser ejecutado, deberá ser planificado por el sistema operativo, de esta forma podrá hacer uso de la CPU por un período determinado de tiempo. Lo anterior ocurre ya que el proceso está siendo ejecutado con otros procesos y debe compartir los recursos, incluyendo la CPU.

4.1. Distribución de la memoria

Todo proceso que se ejecuta dentro del sistema operativo, utilizando una MMU, verá direcciones virtuales. En estas direcciones virtuales se ordena la memoria del proceso a partir de las direcciones más bajas como se muestra en la figura 4.1.

- **Código del programa:** instrucciones en código de máquina a ejecutar por el proceso.

- **Datos:** área para variables globales, inicializadas o no inicializadas.
- **Stack:** área para datos de funciones.
- **Heap:** espacio utilizado para la asignación dinámica de memoria, por ejemplo mediante `malloc`.

Existen implementaciones donde el *stack* y el *heap* están invertidos y el *stack* crece hacia las direcciones bajas de la memoria virtual.

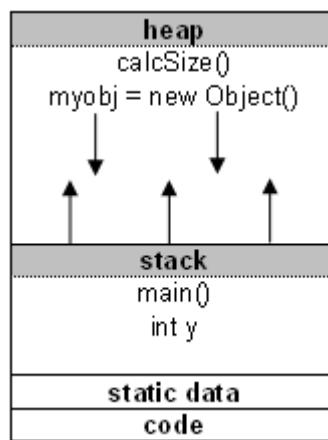


Figura 4.1: Distribución de la memoria

Las direcciones virtuales que no se encuentran asignadas al proceso, o sea donde no hay un mapeo de dichas direcciones virtuales a direcciones físicas no se pueden utilizar. Por lo cual si un proceso trata de acceder arbitrariamente a una zona ilegal de memoria, donde no hay asignación o en algunos sistemas al área del código se producirá una **segmentation fault**. A continuación se muestra un ejemplo de código que podría generar este error:

```

1 #include <stdio.h>
2 int main ()
3 {
4     char *p = (char *) 500000;

```

```

5     printf("%s\n", p);
6     return 0;
7 }
```

En caso de un *segmentation fault* el sistema generará una interrupción llamada dirección ilegal, el sistema operativo determina la causa y envía una señal al proceso indicando el error ocurrido. Si esta señal no es capturada por el proceso hijo, entonces el proceso padre, muchas veces la *shell*, recibirá la señal y mostrará el mensaje “Segmentation fault”, el proceso hijo terminará y el padre (*shel*) continuará su ejecución.

En direcciones virtuales superiores, se encuentra la memoria del sistema, la cual es solo accesible en modo sistema. Cuando ocurre una interrupción, el núcleo atrapará la interrupción y el sistema operativo ejecutará, según el vector de interrupciones, las instrucciones que atienden dicha interrupción. Esta parte de la memoria, corresponde al núcleo del sistema operativo la cual está siempre residente en memoria y es “compartida” entre los procesos. Un proceso normal no puede ver el área de código del sistema operativo, solo se accede cuando hay un cambio a modo sistema. Ningún código que no pertenezca al sistema deberá ejecutarse en dicha área de memoria, ya que de hacerlo implicaría ejecución en modo sistema.

4.2. Contexto

El sistema operativo para gestionar el sistema requiere de diferentes datos, los cuales se organizan en *tablas*, ejemplo de estas tablas son:

- I. **Tabla de memoria:** asignación de memoria principal (RAM), asignación de memoria secundaria (almacenamiento), atributos de protección o de compartición y datos para la gestión de memoria virtual.
- II. **Tabla de E/S:** disponibilidad de recursos, estado de las operaciones de E/S y porción de memoria principal usada como origen/destino (*buffers de E/S*).

III. **Tabla de archivos:** existencia de archivos, posición en memoria secundaria, estado actual y otros atributos.

IV. **Tabla de procesos:** contexto del proceso.

De esta última tabla, la de procesos, nos preocuparemos a continuación.

El **contexto**, imagen o descriptor del proceso corresponde a todos los datos que el sistema operativo requiere para realizar la administración del proceso. Este contendrá diversos datos referentes al estado de ejecución del proceso.

El contexto será una estructura de datos que representará al proceso, conteniendo datos del mismo, como por ejemplo cuantos milisegundos de CPU ha usado el proceso en modo sistema o en modo usuario. El comando `time` entrega el tiempo en modo usuario (*user*) y modo sistema (*sys*). Las interrupciones que ocurren son contabilizadas en el tiempo del proceso que ocurren, las cuales no necesariamente son del proceso que se está ejecutando. Un ejemplo de ejecución del comando `time` para un proceso de compilación es el siguiente:

```
$ time gcc -Wall codigo.c -o programa
real    0m0.082s
user    0m0.052s
sys     0m0.020s
```

4.2.1. Atributos del proceso

A continuación se listan algunos de los elementos que pueden ser encontrados dentro del contexto de un proceso.

4.2.1.1. Identificación del proceso

- Identificador del proceso.
- Identificador del proceso padre.

- Identificador del usuario.

El identificador del proceso o **PID** corresponde a la identificación pública de un proceso. El Sistema operativo administra una tabla que permite asociar el PID hacia la dirección donde se encuentra el contexto del proceso. Los procesos entre sí se conocen únicamente por su PID.

4.2.1.2. Información del estado del procesador

- Registros visibles para el usuario: aquellos que se pueden referenciar mediante el lenguaje de máquina.
- Registros de control y estado: aquellos utilizados por el procesador para ejecutar el código, ej: *program counter*.
- Punteros de pila: apunta a la cima de la pila.

4.2.1.3. Información de control del proceso

- Información de planificación y estado: estado, prioridad, sucesos u otros.
- Estructuración de datos: enlaces entre procesos, ejemplo: colas por estar bloqueados.
- Comunicación entre procesos: señales, mensajes, tuberías.
- Privilegios: memoria, tipo de instrucciones, servicios o utilidades del sistema.
- Gestión de memoria: punteros hacia las direcciones de memoria asignadas.
- Propiedad sobre recursos: recursos controlados por el proceso, ejemplo: archivos abiertos.

4.2.2. Cambios de contexto

El **cambio de contexto** de un proceso ocurre cuando el proceso que se está ejecutando sale de la CPU y entra uno nuevo. Lo anterior ya que cada proceso necesita su propio contexto para la ejecución del mismo, por lo cual el que está almacenado debe ser limpiado y cargado el nuevo.

Si un proceso está ejecutando operaciones de entrada y salida, y los datos asociados a estas operaciones no están en *buffers* el proceso no puede continuar, por lo cual debe bloquearse, entregar el control al sistema operativo y el *scheduler* tomará el control escogiendo otro proceso que si pueda continuar, en ese momento ocurre un cambio de contexto entre los procesos (el que sale por estar bloqueado y el que entra por estar listo). Lo mismo ocurre cuando se acaba el tiempo mediante la interrupción del cronómetro regresivo, ya que al no poder seguir usando la CPU el proceso debe salir y entrar uno diferente (asumiendo que hay más procesos listos).

Suponga el siguiente escenario: tiene un proceso en ejecución en la CPU al cual todavía le queda tiempo del asignado, sin embargo el sistema operativo debe atender una interrupción que llegó por alguna razón. Al ser una interrupción el proceso en ejecución será interrumpido y se pasará a ejecutar la parte de código en el área de sistema, todo esto en el tiempo de ejecución del proceso que está en la CPU. Luego de atender la interrupción se continuará con el proceso por el tiempo que le queda disponible.

Es importante mencionar que una interrupción podría originar un cambio de contexto, pero no necesariamente. Por ejemplo en el caso del cronómetro regresivo se generará una interrupción que implicará cambio de contexto, pero un aviso del disco duro enviando una interrupción informando que los datos están ubicados en el *buffer* no generará cambio de contexto. De todas formas siempre esto tiene que ver con las políticas y mecanismos ya que un sistema operativo podría generar un cambio de contexto ante una interrupción que otro no lo genera.

Los cambios de contexto son caros ya que se debe limpiar la memoria donde se almacena el

mismo, y esto al acceder al hardware, es lento. Luego de limpiar se debe restaurar el contexto de interés (escribiendo en la memoria).

Al realizar un cambio de contexto se debe:

- Resguardar los registros del proceso que sale.
- Contabilizar el uso de CPU.
- Cambiar de espacio de direcciones virtuales. Usualmente implica invalidar caché de nivel 1, lo cual es lo más costoso, esto es así en los procesos pesados y deben su nombre justamente a esto.
- Resguardar los registros del proceso que entra.

4.3. Estados

Durante la ejecución de un proceso este puede encontrarse en diferentes estados, se debe comprender que el proceso al iniciar su ejecución no siempre se estará *ejecutando*, ya que deberá compartir el tiempo de CPU con otros procesos en el sistema, e inclusive con el mismo sistema operativo, el cual también es un proceso en ejecución. Adicionalmente pueden ocurrir otras situaciones que lleven al proceso de un estado a otro.

Los diferentes estados pueden ser vistos en la figura 4.2. A continuación se describirán estos y los motivos que pueden llevar a pasar de uno a otro durante la ejecución del proceso.

Cuando se lanza un programa a ejecución, el proceso no necesariamente comienza a ejecutarse inmediatamente, sino que pasará por un estado de **inicio**, donde se deberán realizar distintas operaciones que tienen que ver con la preparación del entorno para la ejecución del proceso.

Una vez que se ha creado el entorno del proceso, y existe memoria para que este pueda comenzar, pasa a estado **listo** donde espera a ser planificado para entrar a la CPU y ejecutar su código.

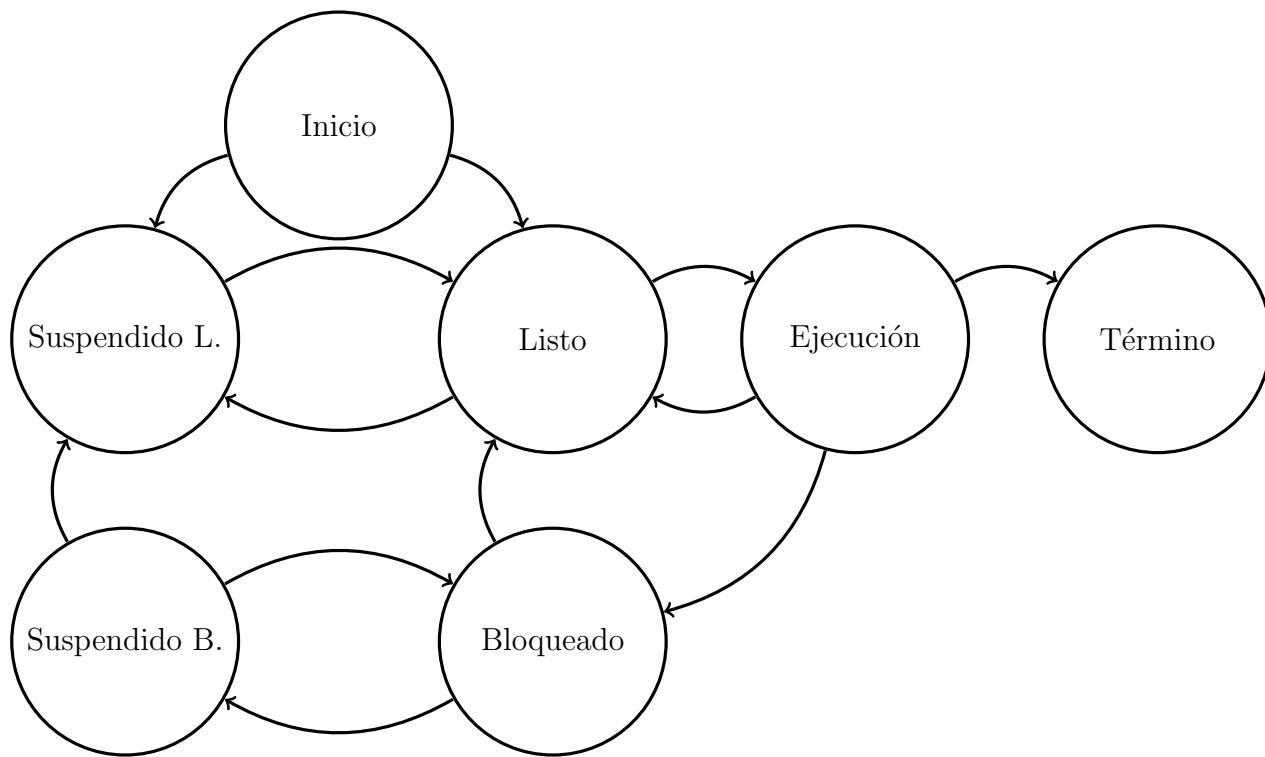


Figura 4.2: Estados de un proceso

Al momento de ser elegido el proceso para su ingreso a la CPU pasa a estado de **ejecución**. Donde se encontrará, en una primera instancia, hasta que el tiempo asignado por el sistema operativo expire. Una vez el tiempo expire el proceso volverá a estado listo, donde volverá a esperar para ser planificado.

Si durante la ejecución del proceso este requiere algún recurso que no está disponible, el proceso pasará a estado **bloqueado** hasta que el sistema operativo le indique que el recurso que está solicitando le fue asignado. Como se vió anteriormente, esto podría ser por ejemplo una lectura de datos desde el disco. Una vez se asigne el recurso el proceso pasará a estado listo nuevamente con el recurso ya disponible para ser utilizado la próxima vez que entre a la CPU.

Una vez el proceso haya cumplido con la ejecución de su programa, o haya ocurrido algún evento que lleve al proceso a su estado final, se encontrará en estado de **término** o estado *zombie*, donde el proceso ya terminó su ejecución pero aun no se han liberado sus recursos. Esto es utilizado, por ejemplo, por un proceso padre que requiere datos una vez el proceso haya terminado, por lo cual será la llamada a **wait** del proceso padre la que liberará finalmente los recursos del proceso *zombie*.

Las razones de término de un proceso no solo se deben porque terminó con la ejecución de su código, a continuación se mencionan otras causas:

- Límite de ejecución excedido.
- Límite de espera excedido.
- No hay memoria disponible.
- Violación de segmento (o límites).
- Error de protección.
- Error aritmético.

- Error E/S.
- Instrucción inválida.
- Instrucción privilegiada.
- Mal uso de datos.
- Intervención del SO.
- Terminación del padre.
- Solicitud del padre.

Con los estados descritos hasta ahora un sistema podría funcionar, sin embargo ¿qué sucedería si en un determinado momento el sistema tiene muchos procesos bloqueados y otros nuevos esperando entrar a estado listo? Con el esquema descrito hasta ahora, si la RAM estuviese completamente ocupada nuevos procesos no podrían ser recibidos. Considerando esto es que aparecen dos estados adicionales, **suspendido listo** y **suspendido bloqueado**, los cuales se encargarán de mover a un almacenamiento secundario los procesos que por alguna razón no puedan ser llevados a estado listo. Si un proceso se encuentra bloqueado será llevado a bloqueado suspendido para que espere sin consumir RAM por el recurso que está solicitando, en cambio si un proceso es nuevo y no hay memoria RAM podrá ser iniciado en un estado suspendido listo, donde ya tendrá su contexto y solo faltará memoria principal para poder ser candidato a planificación.

El sacar un proceso de la CPU y colocar otro en esta implicará diversos pasos, los cuales se mencionan a continuación:

1. Guardar el contexto del proceso que sale.
2. Actualizar el bloque de control del proceso que sale.

3. Mover el bloque de control a la cola adecuada (según estado en que quedó el proceso que sale).
4. Seleccionar otro proceso para ejecución (planificación).
5. Actualizar el bloque de control del proceso seleccionado (cambiar a ejecución).
6. Actualizar las estructuras de datos de gestión de memoria.
7. Restaurar el contexto del proceso, incluyendo los registros del procesador a aquel estado que existía cuando el proceso seleccionado dejó el procesador la vez anterior.

Nos preocuparemos especialmente de los algoritmos de planificación más adelante.

4.4. Clasificación de procesos

Los procesos pueden clasificarse en dos grupos básicos, como **procesos pesados** y como **procesos livianos**. Cada uno tendrá sus características, ventajas y desventajas. En el cuadro 4.1 se pueden apreciar sus similitudes y diferencias.

Notar que la comparación se hace pensando en la ejecución de varios procesos pesados en paralelo en un sistema operativo, o bien la ejecución de un único proceso liviano con muchas hebras ejecutándose de forma paralela.

En sistemas operativos *like Unix* tradicionalmente se han utilizado procesos pesados. Si bien POSIX entrega una implementación de hebras, esto es más “moderno”, en los tiempos iniciales solo habían procesos pesados.

Sistemas operativos *de juguete* por lo general utilizan procesos livianos, ya que el sistema operativo en sí corre sobre un proceso pesado de Unix.

Sistemas operativos como Unix modernos, Linux o WIndows 2000 y NT hacia adelante pueden proveer tanto procesos pesados como livianos.

	Pesado	Liviano
Jerga	procesos Unix	threads
Implementación	fork	hebras
Espacio de direcciones	propio	compartido
Archivos	compartido	compartido
Procesador	propio (1)	propio (varios)
Requisitos de hardware	MMU, interrupciones y timers	interrupciones y timers
Protección	si	no
Comunicaciones	mensajes, sockets, pipes	memoria compartida (punteros)
Costo cambio contexto	alto	bajo
Ejemplos de S.O.	Unix e IBM VM370	AmigaOS, MacOS y Win 3.11

Cuadro 4.1: Comparativa entre procesos pesados y livianos

4.4.1. Procesos *preemptive* y *non-preemptive*

Adicionalmente a la clasificación anterior el sistema operativo puede ofrecer, una de estas opciones, procesos de tipo *preemptive* y *non-preemptive*.

4.4.1.1. Procesos *preemptive*

Los procesos *preemptive* son aquellos donde el núcleo puede quitar la CPU a un proceso en cualquier momento, esto mediante interrupciones.

Ejemplos de este tipo de sistema son sistemas *like Unix* y Windows NT y posteriores.

4.4.1.2. Procesos *non-preemptive*

En los procesos *non-preemptive* es el proceso quien decide invocar al núcleo y devolver el control al sistema operativo. En estos casos debe haber una cooperación entre las aplicaciones y el sistema operativo para ofrecer paralelismo.

Ejemplo de este tipo de sistema son Windows 3.11 y MacOS antes de la versión 6.X. Los

sistemas operativos mencionados no estaban diseñados para la ejecución simultánea de varias aplicaciones, siendo las aplicaciones quienes debían implementar mecanismos de sincronización.

La principal ventaja de esta forma de ejecución de procesos es que son fáciles de programar. Como desventaja se tiene que sin un proceso se queda en un *loop* infinito la única solución es reiniciar la máquina.

4.5. Paralelismo

Un sistema con multiprocesador será capaz de ejecutar procesos en paralelo, en este caso se están considerando varios *chips*. Otra alternativa corresponde a un sistema multinúcleo, donde existe un solo chip de procesador el cual posee varias CPU (núcleos).

En general, lo que hará el sistema operativo será emular el multiprocesamiento, ya que si bien se puede contar con un procesador con 2 o 4 núcleos, o más, siempre se querrá tener más procesos en ejecución que la cantidad de núcleos que la máquina pueda proveer. En estos sistemas se entregarán tajadas de tiempo, donde cada proceso dispondrá de un tiempo finito y determinado para ejecutar su código, de no alcanzar deberá intentarlo más tarde nuevamente.

Si bien este paralelismo solo se podría lograr al disponer de un sistema con múltiples procesadores se debe recordar que los tiempos son tan pequeños que al ejecutarse todos los procesos da la sensación que ocurren en paralelo.

El concepto de concurrencia está relacionado con la ejecución en *paralelo* de los procesos. La **concurrencia** aparecerá al ejecutarse procesos en paralelo, donde dos o más procesos querrán acceder al mismo tiempo a un determinado recurso. Originalmente la única programación con múltiples procesos era la del sistema operativo, ya que los lenguajes no entregaban soporte para concurrencia. Pero hoy en día, como los lenguajes si ofrecen concurrencia como parte del lenguaje o biblioteca es importante conocer lo que esta implica.

Independientemente de si estamos trabajando en un sistema multiprocesador, multinúcleo o con emulación del paralelismo existirán problemas relacionados a este *parallelismo*. Los cuales tendrán directa relación con la forma en que se ejecutan los procesos. Se discutirán a continuación los problemas que pueden ocurrir en un ambiente con múltiples procesos en ejecución, los cuales corresponden a *data races*, *deadlocks* y *starvation*. En el capítulo 5 se verán métodos de sincronización que permitirán controlar estas situaciones.

4.5.1. *Data races*

Los ***data races*** o condición de carrera (*race condition*) ocurre en un proceso cuando se obtiene un estado inconsistente del sistema, o bien cuando los datos que se obtienen se encuentran en un estado inconsistente. La idea de carrera se puede considerar como dos o más procesos que compiten para producir cierto estado final del sistema.

Considere el siguiente código:

```
1 /* parte global (comun a todos los hilos) */
2 int contador = 0;
3 /* codigo principal de cada hilo en ejecucion */
4 void aumentar ()
5 {
6     int aux = contador; /* instruccion 1 */
7     contador = aux + 1; /* instruccion 2 */
8 }
```

Suponga que la función `aumentar()` se está ejecutando en forma paralela en dos hilos (hebras o *threads*), ¿qué problema se podría presentar?. Se debe considerar que las operaciones de la función no son atómicas, o sea pueden ser divididas, por lo tanto el sistema operativo puede interrumpir al proceso y detener su ejecución en cualquier línea de ejecución del código¹. Al existir la posibilidad que el sistema operativo interrumpa la ejecución del

¹Se ha forzado el código a tener 2 líneas, sin embargo debe considerar que aunque fuese una línea la

código en cualquier parte del código puede ocurrir la siguiente situación:

- I. Hilo 1 ejecuta la función `aumentar()`, guarda el valor del contador y es interrumpido.
Entonces `aux = 0`.
- II. Hilo 2 ejecuta la función `aumentar()`, guarda el valor del contador y es interrumpido.
Entonces `aux = 0`.
- III. Hilo 1 hace la suma y guarda el valor. Entonces `contador = 1`.
- IV. Hilo 2 hace la suma y guarda el valor. Entonces `contador = 1`.

Al final de la operación la variable contador tendrá el valor 1, ¿era el valor esperado? ¿qué valor debiera tener la variable contador?.

El resultado esperado será inconsistente ya que se esperaba que después de la ejecución de los dos hilos el contador tuviese valor 2. Sin embargo a causa de la ejecución en paralelo y la no existencia de sincronización el valor resultante es incorrecto. Este problema también es conocido como el problema de **exclusión mutua**, ya que lo lógico que se esperaría es que mientras un proceso modifica una sección crítica los otros no puedan hacerlo.

Este es, de los problemas de concurrencia que se verán, el peor de todos ya que son **difíciles de detectar** y son **no determinísticos**. Adicionalmente entregan un resultado al usuario, uno incorrecto, con lo cual él podría no darse cuenta si dicho resultado es o no el esperado.

4.5.1.1. Agregar elementos a una pila o *stack*

```
1 Pila p;  
2 int indice = 0;  
3 void agregar(Pila p, Objeto o)  
4 {
```

operación será dividida en varias operaciones al ser pasada a código de más bajo nivel

```

5     put(p, o, indice++);
6 }
```

4.5.1.2. Sentarme en una silla

```

1 int sillitas[10]; /* arreglo inicializado en 0 */
2 void sentarme()
3 {
4     int i;
5     for (i=0; i<10; ++i) {
6         if (sillitas[i]==0) {
7             sillitas[i] = 1;
8             me_siento(i);
9             break;
10        }
11    }
12 }
```

Se debe evitar pensar que el orden de las instrucciones puede ayudar con los problemas de *data races*, ya que el compilador puede reordenar el código secuencial dejándolo de una forma no deseada. La solución correcta es el uso de alguna herramienta de sincronización, como semáforos, para garantizar la exclusión mutua.

4.5.2. *Deadlock*

Un ***deadlock*** o interbloqueo corresponde a una situación donde un proceso requiere cierto recurso que algún otro tiene asignado, pero el otro proceso para continuar, y eventualmente liberar el recurso, requiere el que tengo yo asignado. Esto se puede observar en la figura 4.3.

Supongamos por un momento que tenemos una función que permite solicitar un recurso y otra que permite liberar el recurso, más adelante veremos que esto es posible hacerlo mediante

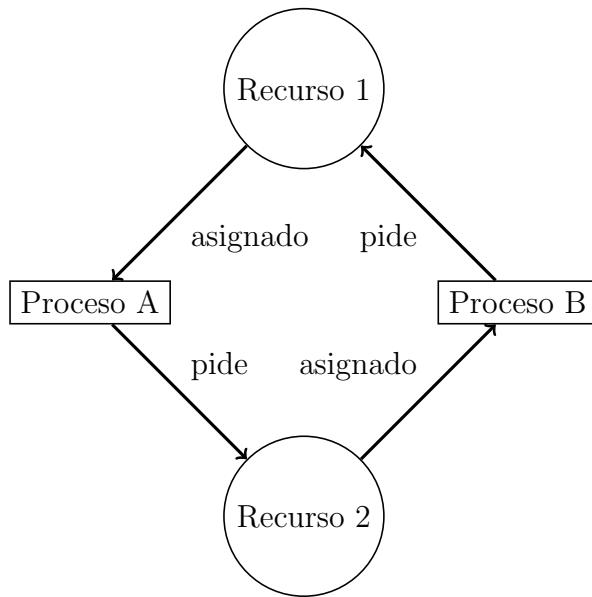


Figura 4.3: Interbloqueo, espera circular entre proceso A y B

herramientas como los semáforos. En este escenario se propone la siguiente situación, donde Pa y Pb son dos procesos diferentes y que se están ejecutando de forma paralela.

1 Pa	Pb
2 solicitar (S);	solicitar (Q);
3 solicitar (Q);	solicitar (S);
4 /* uso de Q y S en la sección crítica */	
5 devolver (S);	devolver (Q);
6 devolver (Q);	devolver (S);

Si el proceso A se está ejecutando, solicita *S*, lo sacan de la CPU, entra el proceso B y solicita *Q*. ¿Qué sucederá cuando entre nuevamente A y solicite *Q*? Ambos procesos estarán esperando que el otro libere el recurso que necesitan.

Pensando en un ejemplo más concreto podría corresponder al problema:

- 1 Servicio tenedor;
- 2 Servicio cuchillo;

```
3 function comer_asado()
4 {
5     solicitar( tenedor );
6     solicitar( cuchillo );
7     comer();
8     liberar( tenedor );
9     liberar( cuchillo );
10 }
```

Para comer se requiere tanto el tenedor como el cuchillo y solo hay disponibles uno de cada uno. ¿Qué podría ocurrir al haber dos personas tratando de comer?

Otro ejemplo puede ser el del puente colgante, donde:

- Tráfico en una sola dirección.
- Cada sección del puente será un recurso.
- Si ocurre un *deadlock*, uno de los usuarios deberá retroceder.
- Puede ser que varios usuarios deban retroceder.
- Puede haber inanición.

Para que ocurra interbloqueo se requieren las siguientes condiciones:

1. Debe existir exclusión mutua.
2. Los procesos deben mantener tomado el recurso y esperar por el siguiente.
3. No debe existir apropiación por parte del sistema operativo (o sea que pueda quitarles el recurso).
4. La espera debe ser circular.

A continuación se mencionan posibles casos con los que el sistema operativo podrá enfrentar un interbloqueo.

4.5.2.1. Ignorar el problema

- Hacer como si el problema no existiera.
- Fundamento: bloqueos pueden ocurrir muy pocas veces, donde las políticas para solucionarlo pueden llevar a mecanismos complejos y que degraden el rendimiento del sistema.
- Unix utiliza este mecanismo.

4.5.2.2. Detección y recuperación

- Permite que ocurran bloqueos.
- Cuando ocurren los detecta y lleva a cabo una acción para solucionarlo.
- Detección: ejecutar algoritmo cada X tiempo que verifique si existen interbloqueos.
- Recuperación:
 - Apropiación: quitar el recurso y asignarlo al otro proceso.
 - Rollback: volver el sistema hacia un punto donde no hay bloqueo.
 - Eliminación del proceso: se eliminan procesos hasta romper el bloqueo.

4.5.2.3. Evitarlo dinámicamente

- Se hace una simulación de como quedaría el sistema si se asigna un recurso solicitado por un proceso.
- Se considera un estado seguro (todos satisfacen sus requerimientos) y uno inseguro (si uno o más procesos no podrán verse satisfechos).

- Si el estado en que queda la simulación es insegura, los recursos no serán asignados al proceso y deberá esperar.
- Algoritmo difícil de implementar, ya que procesos no conocen sus necesidades de recursos para un estado futuro.

4.5.2.4. Evitar las cuatro condiciones

Se busca que al menos una de las 4 condiciones necesarias para el bloqueo no se cumpla.

- Exclusión mutua: si los recursos no se asignaran de forma exclusiva a un proceso no habría problema de interbloqueos.
- Retención y espera: se debe evitar que procesos que ya tienen asignados recursos puedan solicitar nuevos sin liberar los que ya tienen (al menos temporalmente).
- No apropiación: quitar el recurso y asignarlo a otro (no siempre aplicable, ejemplo: impresora).
- Espera circular: los procesos deberán ordenarse para solicitar los recursos, no pudiendo hacerlo todos al mismo tiempo.

4.5.3. *Starvation*

Una situación de ***starvation***, hambruna o inanición corresponde a la situación donde por alguna razón un proceso no obtiene nunca el recurso solicitado. Finalmente el proceso termina por tiempo de espera excedido, o sea, muere de hambre. La existencia de hambruna permitirá tener un mayor paralelismo.

Un ejemplo de esta situación:

- Procesos A, B y C.
- Recurso R.

- Planificador asigna recurso R a A y B, pero nunca a C.
- C nunca adquiere el recurso para completar su objetivo y muere.

Para manejar el problema de inanición el sistema operativo puede asignar los recursos mediante una cola FIFO o bien utilizar una prioridad para los procesos, penalizando a quienes han adquirido el recurso y favoreciendo a quienes no. Esto lo que busca es una asignación equitativa de los recursos, donde ninguno de los procesos debe quedar sin ser atendido.

4.6. Ejercicios y preguntas

1. ¿Qué compone a un proceso?.
2. ¿Cuándo se ejecuta en CPU un proceso?.
3. ¿Para qué se utiliza el espacio de memoria *heap*?.
4. ¿Para qué se utiliza el espacio de memoria *stack*?.
5. ¿Qué es un *segmentation fault*?.
6. Describa a qué corresponde el descriptor de un proceso.
7. ¿Qué es el PID de un proceso?.
8. ¿Qué información guarda el registro de CPU PC (*program counter*)?.
9. Explique el proceso de cambio de contexto.
10. ¿Cuáles son los estados de un proceso?, no es necesario que considere estados suspendidos.
11. ¿Cuándo se pasa de estado listo a ejecución?.
12. ¿Cuándo se pasa de estado ejecución a bloqueado?.

13. ¿Cuándo se pasa de estado bloqueado a listo?.
14. Indique 5 razones de término de un proceso.
15. ¿Por qué es necesario el estado *zombie* o terminado?.
16. ¿Qué se debe restaurar cuando un proceso pasa de estado listo a ejecución?.
17. Explique los proceso pesados y livianos, con sus ventajas y desventajas.
18. Explique la diferencia entre procesos *preemptive* y *non-preemptive*.
19. ¿Bajo que condición existe paralelismo en un sistema operativo?.
20. Explique el problema de *data races*.
21. Explique el problema de *deadlock*.
22. Explique el problema de *starvation*.
23. Explique dos medidas que se puedan tomar frente a un interbloqueo.
24. ¿Por qué el problema de *data races* es considerado el mas complicado (malo) de los tres?.

4.7. Referencias

- Sistemas Operativos, Segunda Edición, Andrew Tanenbaum, Capítulo 2.1.
- Sistemas Operativos, Quinta Edición, Abraham Silberschatz y Peter Baer Galvin, Capítulo 4.
- Sistemas Operativos, Segunda Edición, William Stallings, Capítulo 3.

Capítulo 5

Sincronización

Este capítulo se centra en mecanismos de sincronización entre procesos, esto con el objetivo de solucionar los problemas descritos anteriormente: *data races*, *deadlock* y *starvation*.

Se mostrará primero la forma incorrecta de solucionar los problemas de exclusión mutua, utilizando *busy-waiting*. Luego se presentarán los problemas clásicos estudiados en sistemas operativos, estos son “productor consumidor”, “cena de filósofos” y “lectores escritores”. Finalmente se introducirán tres herramientas de sincronización, las cuales corresponden a semáforos, monitores y mensajes.

5.1. *Busy-waiting*

Una solución a los problemas de exclusión mutua es consultar reiteradamente si el recurso que se está solicitando está o no disponible. Son interesantes, ya que permiten entender porque son incorrectas y que las hace inapropiadas para el problema de exclusión mutua.

Una de las posibles soluciones es utilizar una bandera o *flag* para indicar si la sección crítica esta (1) o no (0) siendo ocupada.

```
1 int bandera = 0;  
2 int contador = 0;
```

```
3 void aumentar()
4 {
5     while (bandera);
6     bandera = 1;
7     int aux = contador;
8     contador = aux + 1;
9     bandera = 0;
10 }
```

- a) **¿Qué sucede con la variable global bandera?** al ser compartida y accedida (para lectura y modificación) por más de un proceso se convierte también en una sección crítica vulnerable *data races*. O sea, lo que se pretendía usar para controlar a sección crítica se convierte en una sección rítica.
- b) **¿Qué ocurre si dos procesos consultan en el while y la bandera es 0?** Puede ocurrir que el primer proceso sea interrumpido justo después del `while` y, al no alcanzar a poner la bandera en 1, el segundo proceso también entre a la sección crítica que se quiere proteger.

Existen diversas soluciones para el problema de exclusión mutua utilizando *busy-waiting*. Se pueden revisar los algoritmos de Dekker y Peterson para más soluciones de este tipo.

El gran problema con las soluciones de *busy-waiting* es que realizan una **espera activa** del recurso. Esto significa que utilizan CPU para consultar cada vez por el recurso. Lo que se verá en el resto del capítulo serán herramientas de sincronización con *espera pasiva*, donde el proceso consulta, y si el recurso está ocupado se “duerme”.

Es importante destacar que la espera activa es el problema de todas las soluciones que utilizan *busy-waiting*, sin embargo pueden existir soluciones de este tipo que a pesar de este problema sean correctas para realizar la sincronización.

5.2. Problemas clásicos

En la literatura relacionada con sistemas operativos generalmente se mencionan tres problemas clásicos, los cuales se enunciarán a continuación y serán utilizados durante los ejemplos de los diferentes métodos de sincronización.

5.2.1. Productor consumidor

En el problema del productor consumidor, o *buffer acotado*, existe uno o varios productores que **producen cierto elemento** mientras que uno o varios consumidores **consumen dicho elemento**. Las operaciones *put* y *get* se realizan sobre una pila finita y los elementos deben ser entregados en el mismo orden en que fueron depositados.

A continuación se muestra una visión global del problema, posteriormente una solución incorrecta. Soluciones correctas serán vistas cuando se estudien los diferentes mecanismos de sincronización.

Versión secuencial (no interesante):

```

1 void productor_consumidor ()
2 {
3     for (;;) {
4         item it = produce();
5         consume(it);
6     }
7 }
```

Versión paralela (interesante):

```

1 void productor ()
2 {
3     for (;;) {
4         item it = produce();
```

```

5     put( it );
6 }
7 }
8 void consumidor ()
9 {
10    for (;;) {
11        item it = get();
12        consume( it );
13    }
14 }
```

Restricciones:

- *get* no puede entregar *items* que no fueron suministrados con *put* (se deberá bloquear el consumidor si no hay items).
- Con *put* se pueden recibir hasta N *items* sin que hayan sido extraídos con *get* (se deberá bloquear del productor si la pila está llena).
- *Items* tienen que ser extraídos en el mismo orden en que fueron depositados.
- Tiene que funcionar en paralelo (no sirve versión secuencial).

Las funciones *produce* y *consume* son lentas y pueden ser ejecutadas en paralelo sin ningún problema (ya que no interactúan al mismo tiempo con otros procesos sobre algún recurso compartido). La solución deberá considerar la implementación sincronizada de *put* y *get*, asumiendo el resto de funciones como dadas (o sea, existen y funcionan).

5.2.1.1. Solución incorrecta: *data races*

```
1 #define N 100
```

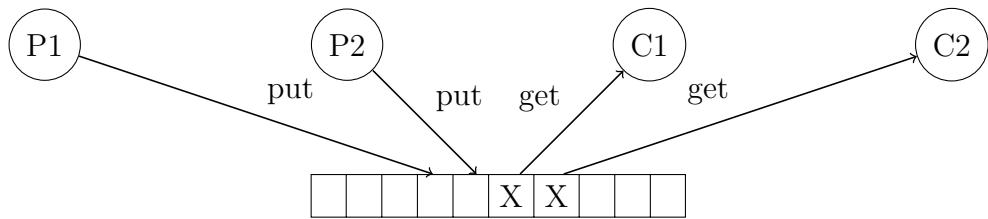


Figura 5.1: Interacción entre productores y consumidores

```

2 Item buffer [N];
3 int e = 0;
4 int f = 0;
5 int c = 0;
6 void put (Item item)
7 {
8     while (c==N);
9     buffer [e] = item;
10    e = (e+1) %N;
11    c++;
12 }
13 Item get ()
14 {
15     Item item ;
16     while (c==0);
17     item = buffer [f];
18     f = (f+1) %N;
19     c--;
20     return item;
21 }
```

¿Por qué la solución anterior es incorrecta? Al utilizar una variable global c para

indicar la cantidad de elementos que hay en la pila, se incurre en el problema de exclusión mutua, donde si no hay sincronización la variable c podría sufrir problemas de *data races* y quedar en un estado inconsistente durante algún punto de la ejecución. Adicionalmente la solución considera el uso de *busy-waiting* lo cual ya se discutió que no es aceptable.

5.2.2. Cena de filósofos

Se ha invitado a una cena a cinco filósofos chinos a comer y pensar, donde realizan solo una de estas actividades al mismo tiempo. En la mesa donde se han de sentar existen 5 puestos y 5 palitos (o tenedores, dependiendo de la bibliografía).

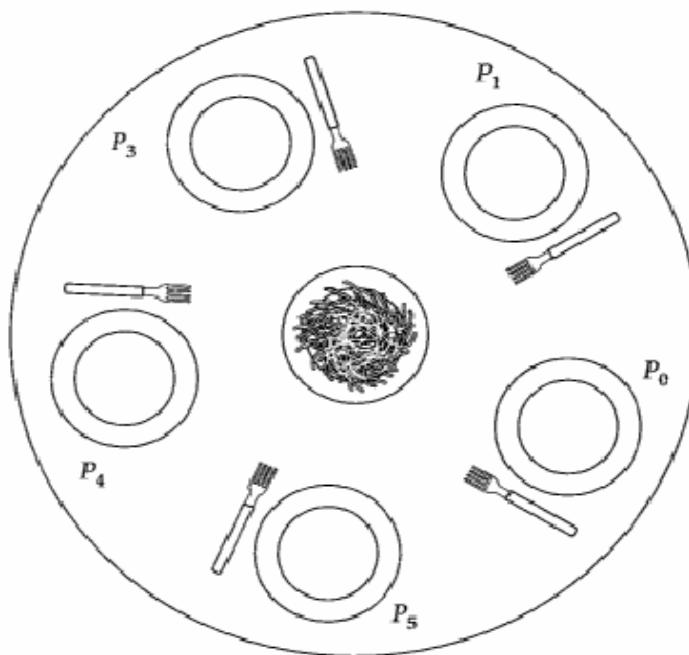


Figura 5.2: Problema cena de filósofos

Restricciones:

- Cada filósofo solo puede tomar palitos que están a su lado.
- Un filósofo requiere dos palitos para comer.

- Un palito no puede ser utilizado por dos filósofos al mismo tiempo.

5.2.2.1. Solución incorrecta: *data races*

```

1 void filosofo (int i)
2 {
3     for (;;) {
4         comer(i, (i+1)%5);
5         pensar();
6     }
7 }
```

El problema en esta solución ocurre por el uso de la función comer directamente con i e $i + 1$, dos filósofos podrían tomar el mismo palito.

5.2.3. Lectores escritores

Este problema utiliza la idea de un diccionario, se dispone de dos arreglos de tamaño fijo donde uno contiene las palabras del diccionario y el otro sus definiciones. Se mostrarán los prototipos (o firmas) de funciones para poder agregar una nueva definición al diccionario, para consultar por una definición y para eliminar una definición.

Restricciones:

- n lectores o escritores requieren acceso a una estructura de datos compartida.
- Los lectores solo consultan.
- Los escritores modifican la estructura de datos.
- Lectores y escritores deberán utilizar herramientas de sincronización ya que si bien múltiples lectores pueden trabajar sobre la estructura, solo un escritor puede hacerlo en un determinado tiempo.

Suponga la siguiente API:

```

1 void newDef ( char *k, char *d );
2 char *query ( char *k );
3 void delDef ( char *k );

```

Un ejemplo del uso de la API se ilustra a continuación:

```

1 newDef("a", "1");           /* se crea palabra "a" con definicion "1" */
2 printf("%s\n", query("a")); /* imprime: 1 */
3 delDef("a");               /* se elimina la palabra a */
4 printf("%s\n", query("a")); /* NULL */

```

Se debe implementar la API de tal forma de cumplir con los requisitos y restricciones del problema.

5.2.3.1. Solución incorrecta: *data races*

```

1 /* Definicion de arreglos globales para palabras y definiciones */
2 #define MAX 100
3 char *keys [MAX], *defs [MAX];
4
5 /* Funcion que obtiene una casilla vacia */
6 int getSlot ()
7 {
8     int i;
9     for ( i=0; i<MAX; i++)
10        if ( keys [ i]==NULL)
11            return i;
12    return -1;
13 }

```

14

15 /* Funcion que crea una nueva definicion en el diccionario */

16 void newDef (char *k, char *d)

17 {

18 int i = getSlot();

19 if (i != -1) {

20 keys[i] = k;

21 defs[i] = d;

22 }

23 }

24

25 /* Funcion que obtiene la posicion en la casilla a partir de una clave */

26 int getIdX (char *k)

27 {

28 int i;

29 for (i=0; i < MAX; i++)

30 if (keys[i] != NULL && !strcmp(k, keys[i]))

31 return i;

32 return -1;

33 }

34

35 /* Funcion que recupera una definicion */

36 char *query (char *k)

37 {

38 int i = getIdX(k);

39 return i == -1 ? NULL : defs[i];

40 }

41

```
42 /* Funcion que elimina una definicion del diccionario */  
43 void delDef (char *k)  
44 {  
45     int i = getIdX(k);  
46     if (i != -1) {  
47         keys[i] = NULL;  
48         defs[i] = NULL;  
49     }  
50 }
```

Al analizar el código se observa que pueden ocurrir *data races* si se realizan al menos dos modificaciones en paralelo o si se realiza una modificación con una consulta. No ocurrirán problemas si se realizan dos consultas en paralelo.

Algunas de las situaciones anómalas se describen a continuación (el lector deberá considerar que otros problemas pueden ocurrir).

- I. **newDef con newDef**: al realizar una nueva definición se consulta por una casilla libre, si justo en el momento después de consultar si la casilla está libre (instrucción `if` en `getSlot`), se saca al proceso de la CPU y entra otro proceso que hace la misma consulta se podría entregar la misma casilla a ambos procesos.
- II. **query con delDef**: al seleccionar una definición se consulta mediante `getIdX` el índice dentro del arreglo, si justo después de consultar por el índice se saca al proceso de la CPU y se llama a `delDef` eliminando la misma definición consultada, cuando `query` retome la CPU y use el índice obtenido no estará lo esperado en la casilla.
- III. **query con newDef**: el resultado de `query` puede ser `NULL` si se ejecuta antes que `newDef` o distinto de `NULL` si se ejecuta después.

5.3. Semáforos

Los **semáforos** corresponden básicamente a contadores (de *tickets*), donde para utilizar una sección crítica se consulta por el valor del semáforo, si este es mayor a cero (o sea hay *tickets*) se usa la sección, si es igual a cero, se deberá esperar.

El valor del semáforo (o la cantidad de *tickets*) será inicializado, generalmente, en la cantidad de procesos que pueden hacer uso al mismo tiempo de la sección crítica. Adicionalmente las rutinas utilizadas deben ser atómicas, esto para garantizar que varios procesos puedan consultar el valor del semáforo sin que ocurran *data races*.

```

1 s=1;
2 void proceso ()
3 {
4     solicitar_ticket(s);
5     // ejecucion de seccion critica
6     liberar_ticket(s);
7 }
```

Los procesos que no tengan acceso al semáforo, por falta de *tickets*, deberán esperar en una cola FIFO a que se depositen *tickets*, en cuyo momento se despertará al proceso que estaba primero en la cola para que obtenga el *ticket* del semáforo y entre en la sección crítica.

5.3.1. API

```

1 /* Inicializa el semaforo */
2 struct semaphore *semaphore_make (int tickets);
3
4 /* Solicita el semaforo */
5 void semaphore_wait (struct semaphore *s);
6
```

```

7 /* Libera el sem foro */
8 void semaphore_signal ( struct semaphore *s );

```

5.3.2. Modo de operación

Un bosquejo de la implementación de las funciones de la API que ayudará a comprender como operan los semáforos es la siguiente:

```

1 void wait ( s )
2 {
3     s--;
4     if ( s < 0 )
5         block ( s );      /* suspender proceso y agregarlo al final de la cola */
6 }
7 void signal ( s )
8 {
9     s++;
10    if ( s <= 0 )
11        wakeup ( s );   /* despertar primer proceso en la cola */
12 }

```

¿Qué significa que el valor del semáforo sea -1?, significaría que ya hay un proceso antes en la cola FIFO. Por lo cual al recibir un **signal** se despertará primero al otro proceso.

Una implementación real, utilizando **nSystem**¹, puede ser vista en el anexo ??.

5.3.3. Problema productor consumidor

5.3.3.1. Solución correcta

```

1 #define N 100

```

¹Sistema Operativo de juguete desarrollado por el profesor Luis Mateu de la Universidad de Chile

```
2 Item buffer [N];
3 int e = 0;
4 int f = 0;
5 struct semaphore *empty; /* = semaphore_make (N) */
6 struct semaphore *full; /* = semaphore_make (0) */
7 void put (Item item)
8 {
9     semaphore_wait (empty);
10    buffer [e] = item;
11    e = (e+1) %N;
12    semaphore_signal (full);
13 }
14 Item get ()
15 {
16     Item item;
17     semaphore_wait (full);
18     item = buffer [f];
19     f = (f+1) %N;
20     semaphore_signal (empty);
21     return item;
22 }
```

Esta solución es válida para un productor y un consumidor. Para n productores y m consumidores se debe definir un semáforo adicional para cubrir la sección crítica tanto de la función `put` como de la función `get`. Esto queda de tarea al lector.

5.3.4. Problema cena de filósofos

5.3.4.1. Solución incorrecta: *deadlock*

```

1 struct semaphore s[5]; /* for ( i=0; i<5; i++) s[ i ] = semaphore_make ( 1 ); */
2 void filosofo ( int i )
3 {
4     for ( ; ) {
5         semaphore_wait ( s[ i ] );
6         semaphore_wait ( s[ ( i+1)%5 ] );
7         comer ( i , ( i+1)%5 );
8         semaphore_signal ( s[ i ] );
9         semaphore_signal ( s[ ( i+1)%5 ] );
10        pensar ( );
11    }
12 }
```

El problema en esta solución es lo que sucedería si todos los filósofos solicitaran el palito i “al mismo tiempo”, cuando quieran solicitar el palito $i + 1$ ya alguien lo tendrá tomado.

5.3.4.2. Solución correcta específica

Se debe evitar que entren los 5 filósofos al mismo tiempo a la mesa, con esto se asegurará que al menos uno de ellos coma. En este caso la misma mesa se convierte en una sección crítica.

```

1 struct semaphore *m;      /* m = semaphore_make ( 4 ); */
2 struct semaphore s[5];   /* for ( i=0; i<5; i++) s[ i ] = semaphore_make ( 1 ); */
3 void filosofo ( int i )
4 {
5     for ( ; ) {
6         semaphore_wait ( m );
7         semaphore_wait ( s[ i ] );
8         semaphore_wait ( s[ ( i+1)%5 ] );
```

```

9      comer ( i , ( i +1)%5);
10     semaphore_signal ( s [ i ] );
11     semaphore_signal ( s [ ( i +1)%5] );
12     semaphore_signal ( m );
13     pensar ();
14 }
15 }
```

5.3.4.3. Solución correcta general

Se deben solicitar los recursos siempre en el mismo orden, ya sea ascendente o descendente. Esto evitará el interbloqueo.

```

1 struct semaphore s [ 5 ]; /* for ( i=0; i <5; i++) s [ i ] = semaphore_make ( 1 );
2 void filosofo ( int i )
3 {
4     int min = min ( i , ( i +1)%5);
5     int max = max ( i , ( i +1)%5);
6     for (;;) {
7         semaphore_wait ( s [ min ] );
8         semaphore_wait ( s [ max ] );
9         comer ( min , max );
10        semaphore_signal ( s [ min ] );
11        semaphore_signal ( s [ max ] );
12        pensar ();
13    }
14 }
```

Asumamos que entra el filósofo 2, solicitará el palito 2 y el palito 3 y comerá. Luego entra el filósofo 1, solicita el palito 1 (está disponible), pero al pedir el palito 2 (que esta ocupado por el filósofo 2) se bloquea a la espera que se desocupe. El problema aquí es que el filósofo 1 espera reteniendo el palito 1, entonces si llega un filósofo 0, que puede usar el palito 0, el palito 1 no lo conseguirá a pesar de que no está siendo utilizado para comer (solo está siendo retenido por el filósofo 1). Eventualmente, cuando el filósofo 2 libere el palito 2, el filósofo 1 comerá y eventualmente el 0 también lo hará. Sin embargo ¿podría haber comido el 2 y el 0 al mismo tiempo?, la respuesta es sí.

El problema de los semáforos, en general, es que no se puede saber si el semáforo está o no ocupado antes de pedirlo. Esto trae como consecuencia una limitación del paralelismo, ya que al consultar por el palito este es retenido independientemente de si el otro está o no disponible. Sin embargo, si se considera que la operación comer es mucho más rápida que la de pensar, por ejemplo asumiendo que comer es leer datos y pensar es procesarlos, esta limitación de paralelismo no es tan grave para ciertos problemas.

Lo anterior se podría solucionar implementando “algo” que permita pedir dos semáforos al mismo tiempo, pero que no los deje tomados si uno de ellos está ocupado. Esto entregará un mayor paralelismo, pero introducirá el problema de hambruna, donde dos filósofos (por ejemplo filósofos 0 y 2) podrían estar pidiendo siempre los palitos que un tercero (filósofo 1) quisiera utilizar.

5.4. Monitores de Brinch Hansen

Los **monitores** de Brinch Hansen² son una herramienta de sincronización que ofrece más paralelismo que los semáforos, pero pueden provocar hambruna. Permitirán consultar al mismo tiempo por el valor de más de una condición, si alguno de los elementos de dicha condición no se cumple de la forma requerida el proceso se suspenderá.

²http://es.wikipedia.org/wiki/Per_Brinch_Hansen

Los monitores pueden ser vistos como semáforos binarios, donde el “*ticket*” del monitor es la propiedad del mismo.

```
1 void proceso ()
2 {
3     solicitar_propiedad(m);
4     // ejecución de sección crítica
5     liberar_propiedad(m);
6 }
```

Es importante mencionar que la propiedad del monitor debe ser devuelta siempre por el mismo proceso que la solicitó.

5.4.1. API

```
1 /* Crear un monitor */
2 struct monitor *monitor_make ();
3
4 /* Destruir un monitor */
5 void monitor_destroy (struct monitor *m);
6
7 /* Solicitar la propiedad sobre un monitor */
8 void monitor_enter (struct monitor *m);
9
10 /* Devuelve la propiedad del monitor */
11 void monitor_exit (struct monitor *m);
12
13 /* Devuelve la propiedad y suspende el proceso hasta un monitor_notify_a
   */
```

```
14 void monitor_wait ( struct monitor *m);  
15  
16 /* Despierta las tareas suspendidas con monitor_wait que esperan la propiedad  
17 del monitor */  
18 void monitor_notify_all ( struct monitor *m);
```

Un proceso que espera, suspendido, por la propiedad del monitor al haber usado `monitor_enter` esperará hasta que esta sea liberada por el proceso que la ocupa, ya sea mediante un `monitor_wait` o un `monitor_exit`.

Al despertar procesos bloqueados por un `monitor_enter` o un `monitor_wait` obtendrán la propiedad sin garantía del orden, o sea, **monitores no son FIFO**. Igualmente, cuando se hace una llamada a `monitor_notify_all` se despertarán todos los procesos suspendidos por un `monitor_wait` del mismo monitor, el orden en que despiertan no está garantizado, o sea no necesariamente es FIFO respecto a la ejecución de `monitor_wait`. Adicionalmente una vez despertados los procesos deben esperar la propiedad del monitor (esto ya que `monitor_notify_all` solo despierta, no entrega la propiedad, la cual será entregada al usar `monitor_exit`) y evaluar nuevamente sus condiciones, si nuevamente no se cumplen se suspenderán liberando el monitor, así otro proceso despertado con `monitor_notify_all` podrá obtener la propiedad y también evaluar sus condiciones nuevamente. Podría ocurrir también que con `monitor_notify_all` los procesos al evaluar sus condiciones, ninguno pueda continuar, y todos se vuelvan a suspender.

5.4.2. Problema productor consumidor

5.4.2.1. Solución incorrecta: *deadlock*

```
1 void put ( Item it )  
2 {  
3     monitor_enter ( monitor );  
4     while ( c==N );
```

```

5   buffer [ e ] = it ;
6   e = ( e+1 ) %N;
7   c++;
8   monitor_exit ( monitor );
9 }

10 Item get ()
11 {
12   Item it ;
13   monitor_enter ( monitor );
14   while ( c==0 );
15   it = buffer [ f ];
16   f = ( f+1 ) %N;
17   c--;
18   monitor_exit ( monitor );
19   return it ;
20 }
```

Suponga que $c = N$, o sea el productor no puede depositar más *items* en la pila y debe esperar. En este caso al ejecutar **put**, se solicitará la propiedad del monitor y el proceso quedará en el ciclo del **while**, con espera activa y con el monitor tomado. Si llega un consumidor y solicita la propiedad sobre el monitor para extraer un *item*, lo cual es válido por la situación descrita, no podrá hacerlo, ya que el monitor está tomado por un productor, el cual espera (la condición de su **while**) que un consumidor extraiga al menos un *item*. En este caso se produce el problema de interbloqueo.

5.4.2.2. Solución correcta

Se debe buscar una solución que permita devolver el monitor si la condición de espera se cumple, esto se logra utilizando **monitor_wait** sobre el monitor.

```
1 void put (Item it)
2 {
3     monitor_enter (monitor);
4     while (c==N)
5         monitor_wait (monitor);
6     buffer [e] = it ;
7     e = (e+1) %N;
8     c++;
9     monitor_notify_all (monitor);
10    monitor_exit (monitor);
11 }
12 Item get ()
13 {
14     Item it ;
15     monitor_enter (monitor);
16     while (c==0)
17         monitor_wait (monitor);
18     it = buffer [f ];
19     f = (f+1) %N;
20     c--;
21     monitor_notify_all (monitor);
22     monitor_exit (monitor);
23     return it ;
24 }
```

Si llega un productor y ve que la pila está llena, hará espera pasiva y devolverá la propiedad del monitor. Análogamente si un consumidor ve que la pila está vacía esperará de forma pasiva y devolverá la propiedad del monitor. Esta acción de devolver la propiedad del monitor y

quedan en espera pasiva es lograda mediante `monitor_wait`. Adicionalmente se agrega la instrucción `monitor_notify_all`, la cual despertará a todos los procesos bloqueados mediante un `monitor_wait` de dicho monitor, obtendrán la propiedad y podrán evaluar nuevamente la condición de espera, si no se sale del `while` se volverá a dormir entregando la propiedad a otro proceso para que pueda continuar. Notar que no se puede determinar en qué orden serán despertados los procesos, por lo cual cualquier proceso podría tomar la propiedad una vez sean despertados por `monitor_notify_all`.

5.4.3. Patrón de solución usando monitores

```

1 ... operacion (...)

2 {
3 ...
4 monitor_enter (m);
5 while (!invariante)      /* condicion para quedar en espera */
6     monitor_wait (m);
7 ...                      /* operaciones */
8 monitor_notify_all (m); /* opcional , solo si las operaciones modifican
9                           datos que afecten la condicion de otros que
10 esperan (ademas podran ir antes del while)
11 */
12 monitor_exit (m);
13 return ...;
14 }
```

Se recomienda aislar los distintos aspectos de la solución, separando el código de sincronización del resto de la aplicación. Se verá esto a continuación en la solución de la cena de

filósofos.

5.4.4. Problema cena de filósofos

5.4.4.1. Solución “correcta”

```
1 /* llamadas a rutinas del filosofo */
2 void filosofo (int i)
3 {
4     for (;;) {
5         pedir (i , (i+1)%5);
6         comer (i , (i+1)%5);
7         devolver (i , (i+1)%5);
8         pensar ();
9     }
10 }
11
12 /* sincronizacion */
13
14 struct monitor *m;           /* m = monitor_make (); */
15 int ocup[5] = {0, 0, 0, 0, 0} /* =0 palito esta libre */
16
17 void pedir (int i , int j)
18 {
19     monitor_enter (m);
20     while(ocup[i] || ocup[j])
21         monitor_wait (m);
22     ocup[i] = ocup[j] = 1;
23     monitor_exit (m);
```

```

24  }
25
26 void devolver (int i , int j) {
27   monitor_enter (m);
28   ocup[ i ] = ocup[ j ] = 0;
29   monitor_notify_all (m);          /* DEBE ir , ya que se liberan los palitos
30   monitor_exit (m);
31 }
```

Esta solución evita que un filósofo retenga un palito sin estar comiendo (lo que sucedía con semáforos), por lo cual aumenta el paralelismo. Sin embargo introduce el problema de **hambruna**.

Suponga que en el tiempo 0 ingresa el filósofo 0, podrá comer con los palitos 0 y 1. Luego ingresa el filósofo 1 el cual no podrá comer ya que de los palitos requeridos (1 y 2) el 1 ya está siendo ocupado. Luego ingresa el filósofo 2 y podrá comer (usando palitos 2 y 3). Resumiendo, F_0 y F_2 están comiendo, mientras F_1 está esperando a que los palitos (1 y 2) sean liberados. Suponga ahora que después de un tiempo F_0 deja de comer, se notifica a F_1 que hay palitos libres, sin embargo solo dispone del palito 1, por lo cual deberá seguir esperando por el palito 2 y no podrá comer. Luego F_0 vuelve a pedir los palitos, están libres y come. Ahora F_2 liberará los palitos, pero como F_0 volvió a pedir el palito 1, a pesar de tener el 2 libre F_1 no podrá comer. Esta situación puede repetirse indefinidamente, donde por la ejecución de F_0 y F_2 , F_1 podría nunca tener disponibles los 2 palitos que necesita para poder comer.

Se deja al lector la tarea de mostrar mediante un diagrama de hebras la situación de hambruna descrita anteriormente.

5.4.5. Problema lectores escritores

Las funciones `newDef` y `delDef` definirán un conjunto de operaciones de escritura, donde su sección crítica será rodeada por un `enterWrite` y `exitWrite` que se encargarán de realizar las

tareas de sincronización. Análogamente, la función `query` denotará una operación de lectura, donde su sección crítica será rodeada por un `enterRead` y `exitRead`.

A continuación se muestra el modo de uso de estas funciones de sincronización, posteriormente se revisarán dos implementaciones de las mismas.

```
1 /* Lector */
2 char *query (char *k)
3 {
4     enterRead (); /* inicio seccion critica */
5     int i = getIdX(k);
6     char *aux = i== -1 ? NULL : defs [ i ];
7     exitRead (); /* fin seccion critica */
8     return aux;
9 }
10
11 /* Escritor */
12 void newDef (char *k, char *d)
13 {
14     enterWrite (); /* inicio seccion critica */
15     int i = getSlot ();
16     if (i!= -1) {
17         keys [ i ] = k;
18         defs [ i ] = d;
19     }
20     exitWrite (); /* fin seccion critica */
21 }
22 void delDef (char *k)
23 {
```

```

24     enterWrite(); /* inicio seccion critica */
25     int i = getIdX(k);
26     if (i != -1) {
27         keys[i] = NULL;
28         defs[i] = NULL;
29     }
30     exitWrite(); /* fin seccion critica */
31 }
```

El uso de funciones diferentes para acceder a la sección crítica en lecturas y escrituras está relacionado con que varios lectores pueden consultar al mismo tiempo la sección crítica, sin embargo no puede haber alguien más cuando se hace una modificación (o sea, escritores deben trabajar solos).

5.4.5.1. Solución “correcta”

```

1 struct monitor *c;          /* = monitor_make () */
2 int readers = 0; /* contador de lectores leyendo */
3 int writing = 0; /* =0 no hay alguien escribiendo */
4
5 void enterRead ()
6 {
7     monitor_enter (c);
8     while (writing) /* se pregunta si alguien esta escribiendo */
9         monitor_wait (c);
10    readers++; /* se indica que entro un lector */
11    monitor_exit (c);
12 }
```

13

```
14 void exitRead ()
15 {
16     monitor_enter (c);
17     readers--;
18     monitor_notify_all (c); /* avisa a escritores que podrian entrar */
19     monitor_exit (c);
20 }
21
22 void enterWrite ()
23 {
24     monitor_enter (c);
25     while (readers>0 || writing)
26         monitor_wait (c);
27     writing = 1; /* se indica que un escritor esta trabajando */
28     monitor_exit (c);
29 }
30
31 void exitWrite ()
32 {
33     monitor_enter (c);
34     writing = 0;
35     monitor_notify_all (c); /* avisa a lectores o escritores que podrian entrar */
36     monitor_exit (c);
37 }
```

Recordar que al utilizar `monitor_exit` se libera la propiedad del monitor y alguno de los que estaba esperándola la tomará, sin embargo no se especifica cual proceso será (recordar,

no hay orden FIFO).

Si “llegan y llegan” lectores, los cuales pueden leer en paralelo, y nunca el contador `readers` es 0, ningún escritor podrá acceder a la sección crítica. Esto corresponde a un problema de hambruna en los escritores.

5.4.5.2. Solución correcta

Se busca la ausencia de hambruna en los escritores, se debe utilizar una estrategia en que la llegada continua de lectores no deje a los escritores sin acceso a la sección crítica.

La idea aquí es autorizar las entradas en orden FIFO, donde se atenderán solicitudes de lectores en paralelo hasta que llegue un escritor, en cuyo caso se esperará que los lectores salgan y luego se atenderá al escritor. Lo anterior, independientemente de si después del escritor llegan nuevos lectores que podrían haber leído en paralelo junto a los iniciales.

Se utilizará un turno para la ejecución, donde los procesos deberán esperar su turno para poder ejecutar las acciones sobre la sección crítica. Esto claramente disminuirá el paralelismo, pero evitará la hambruna.

Se usa el mismo código de la solución anterior (con hambruna), agregando dos nuevas variables globales y las modificaciones descritas para las funciones `enterRead` y `enterWrite`. El resto es igual.

```

1 int visor = 0;           /* la idea es que del distribuidor se saca un "turno"
2 int distribuidor = 0;    solo se avanza cuando el visor así lo indica */
3
4 void enterRead ()
5 {
6     monitor_enter (c);
7     int miturno = distribuidor++;
8     while (writing || visor != miturno)
9         monitor_wait (c);
10    readers++;

```

```
11     visor++;
12     monitor_notify_all ( c )
13     monitor_exit ( c );
14 }
15
16 void enterWrite ()
17 {
18     monitor_enter ( c );
19     miturno = distribuidor++;
20     while( writing || readers>0 || miturno !=visor )
21         monitor_wait ( c );
22     writing = 1;
23     visor++;
24     monitor_notify_all ( c );
25     monitor_exit ( c );
26 }
```

En el problema de filósofos se puede utilizar una solución similar para el problema de hambruna con monitores, sin embargo el problema ahí sería que un filósofo no podría comer (a pesar de tener los dos palitos en su poder) por no ser su turno. Se reitera que solucionar el problema de hambruna podría traer como consecuencia una disminución del paralelismo.

5.5. Mensajes

Los **mensajes** corresponden a otra herramienta de sincronización, en este caso la atención de los mensajes es en orden FIFO y es un mecanismo síncrono, ya que las funciones **send** y **receive** son bloqueantes (o sea hay espera).

Aquí la idea es que los procesos “conversan” entre sí para lograr la sincronización de sus

operaciones. Lo anterior, por ejemplo, es lo que ocurre al utilizar *pipes*.

5.5.1. API

```

1 /* Envio de mensaje m a t. Espera hasta que se responda el mensaje con
2    message_reply */
3 int message_send (struct task *t, void *m);
4
5 /* Espera a que se le envie un mensaje y lo entrega (retorna). *pt es la
6   identificacion del emisor, si delay >=0 se espera como maximo delay [m]
7   si se indica como -1 se esperara infinitamente. */
8 void *message_receive (struct task *pt, int delay);
9
10 /* Responde el mensaje emitido por t, t se desbloqueara retornando rc
11   (en la llamada de message_send) */
12 int message_reply (struct task *t, int rc);
```

5.5.2. Ejemplos de uso

En la figura 5.3 se puede observar un ejemplo de uso de mensajes, donde la tarea 1 envía un mensaje *m* a la tarea 2, esta procesa de alguna forma el mensaje *m* recibido (retornado por `message_receive`) y finalmente responde el mensaje a la tarea 1 con el valor de retorno 100, el cual es entregado una vez se desbloquea la función `message_send` que originó el mensaje en la tarea 1.

En la figura 5.4 se observa otro ejemplo del uso de mensajes. En este caso la tarea 2 se bloquea esperando la recepción de un mensaje desde alguna otra tarea (cualquiera). Una vez se recibe el mensaje en la tarea 2, esta se desbloquea y procesa el mensaje, mientras tanto la tarea 1 quedó bloqueada a la espera de la respuesta. Una vez se termina de procesar el mensaje la tarea 2 responde a la tarea 1 y se desbloquea.

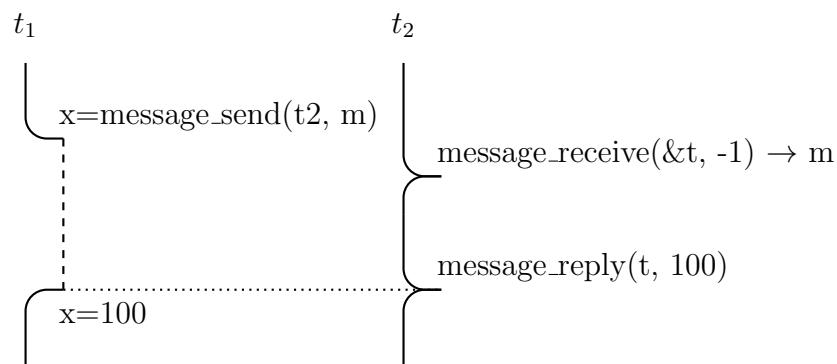


Figura 5.3: Ejemplo 1 de uso de mensajes

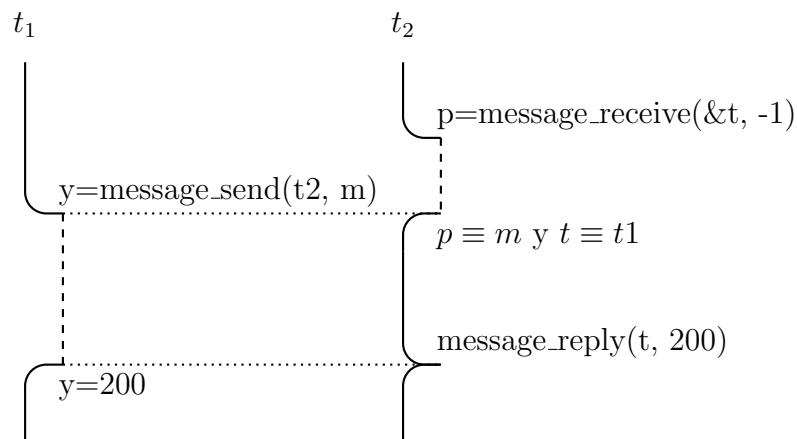


Figura 5.4: Ejemplo 2 de uso de mensajes

Es importante mencionar que `message_reply` no necesariamente debe ser enviado por quien recibió el mensaje, puede enviarlo otra tarea, la cual ni siquiera haya recibido un mensaje.

Ejemplos reales del uso de mensajes son el sistema *xWindow*³ y sistemas de bases de datos. Donde el sistema en sí funciona como un servicio y los clientes envían los “mensajes” al servidor para ser procesados. El procesamiento de estos se hace respetando la exclusión mutua y de tal forma que se simula un paralelismo para los clientes.

5.5.3. Exclusión mutua con mensajes

Supongamos un sistema donde múltiples procesos solicitan servicios a otro proceso. En este escenario, cada solicitud de ejecución de un cierto requerimiento, llamémoslo función f , corresponderá a una sección crítica. Esto podría ser, por ejemplo, dibujar ciertos pixeles en la pantalla.

```
1 /* Sea f la función que se debe ejecutar en exclusion mutua */
2 int (*f)(Param *p);
```

5.5.3.1. Implementación

Cada vez que una tarea t quiera ejecutar la función f solicitará al servidor s que realice esta tarea, esto es lo que se observa en la figura 5.5

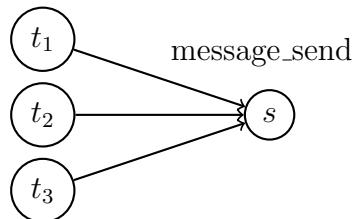


Figura 5.5: Ejecución de la función f en el servidor

³http://es.wikipedia.org/wiki/X_Window_System

Se utilizará una función *doReq* que enviará el requerimiento, la ejecución de la función *f* al servidor utilizando mensajes, o sea, mediante `message_send`.

```
1 struct task *server; /* = task_emmit (serverProc); */  
2  
3 /* Mensaje que se pasara al servidor (es el requerimiento) */  
4 typedef struct req  
5 {  
6     int (*f) (Param *p);  
7     Param *p;  
8 } Req;  
9  
10 /* Funcion que hace el requerimiento al servidor */  
11 int doReq(int (*f)(Param *p), Param *p)  
12 {  
13     Req r;  
14     r.f = f;  
15     r.p = p;  
16     return message_send(server, &r);  
17 }  
18  
19 /* Funcion que ejecuta funcion f de forma secuencial segun se reciben */  
20 int serverProc ()  
21 {  
22     for (;;) {  
23         struct task *t;  
24         Req *pr = (Req *) message_receive (&t, -1);  
25         int rc = (*pr->f)(pr->p);
```

```
26     message_reply(t, rc);  
27 }  
28 }
```

5.5.4. Implementación de semáforos a partir de mensajes

```
1 #define WAIT 1  
2 #define SIGNAL 2  
3  
4 typedef struct  
5 {  
6     struct task *semTask;  
7 } *Sem;  
8  
9 Sem semMake (int ini) {  
10    Sem s = (Sem) nMalloc (sizeof(*s));  
11    s->semTask = task_emmit (semProc, ini);  
12 }  
13  
14 void semWait (Sem s)  
15 {  
16     int cmd = WAIT;  
17     message_send(s->semTask, &cmd);  
18 }  
19  
20 void semSignal (Sem s)  
21 {
```

```
22     int cmd = SIGNAL;
23     message_send (s->semTask , &cmd);
24 }
25
26 int semProc (int tickets)
27 {
28     FifoQueue q = MakeFifoQueue ();
29     for (;;) {
30         struct task *t;
31         int *pcmd = (int *) message_receive (&t , -1);
32         /* Si es un WAIT */
33         if (*pcmd==WAIT) {
34             /* Si hay tickets otorgar */
35             if (tickets>0) {
36                 message_reply (t , 0);
37                 tickets--;
38             }
39             /* En caso que no hayan tickets encolar */
40             else {
41                 PutObj (q , t);
42             }
43         }
44         /* Si es un SIGNAL */
45         else if (*pcmd==SIGNAL) { /* if no es necesario en este caso */
46             /* Si la cola esta vacia se aumentan los tickets */
47             if (EmptyFifoQueue (q)) {
48                 tickets++;
```

```

49      }
50      /* Si hay elementos en la cola se despierta */
51      else { /* implica tickets = 0 */
52          struct task *w = (nTask) GetObj(q);
53          message_reply(w, 0);
54      }
55      message_reply(t, 0); /* podria ir antes de verificar la cola */
56  }
57 }
58 }
```

5.6. Monitores de Hoare

Los **monitores de Hoare**⁴ corresponden a otro tipo de monitores, donde su implementación difiere de la vista anteriormente. Antes de entrar en su definición y API, una pregunta que aparece es **¿por qué se necesitan?**, para responder a esto se verá el caso de la implementación de semáforos mediante monitores la cual resultará en una implementación con problemas.

5.6.1. Implementación de semáforos a partir de monitores

```

1 typedef struct sem
2 {
3     int c;                      /* este es el contador para el semaforo */
4     struct monitor *m;          /* monitor para controlar el acceso al semaforo */
5 } *Sem;
6
```

⁴http://en.wikipedia.org/wiki/Tony_Hoare

```
7 Sem semMake ( int ini )
8 {
9     Sem s = (Sem) nMalloc ( sizeof(*s));
10    s->c = ini;
11    s->m = monitor_make ();
12    return s;
13 }
14
15 void waitSem (Sem s)
16 {
17     monitor_enter (s->m);
18     while (s->c==0)
19         monitor_wait (s->m);
20     s->c--;
21     monitor_exit (s->m);
22 }
23
24 void signalSem (Sem s)
25 {
26     monitor_enter (s->m);
27     s->c++;
28     monitor_notify_all (s->m);
29     monitor_exit (s->m);
30 }
```

Pueden existir n hebras que están esperando que otra hebra ejecute un `signalSem`, que ejecutará un `monitor_notify_all`. Al ocurrir esto se despertarán todos las hebras esperando con `monitor_wait` y el primero que lo haga adquirirá el *ticket* que se depositó con

`signalSem`, mientras que el resto deberá llamar a `monitor_wait` nuevamente. Esto trae dos problemas:

1. No está garantizado el orden en que se despiertan los procesos al usar `monitor_notify_all`, por lo cual el semáforo no tendría orden FIFO.
2. Esta implementación es inefficiente debido a los numerosos cambios de contexto necesarios para que cada uno de los procesos verifique que no hay *tickets* (a partir del segundo despertado) y se vuelva a dormir.

Los problemas anteriores se evitarían si existiese una forma de despertar solo a un proceso, y que solo ese proceso tome el *ticket*. Este proceso debiese ser el primero que se puso en espera con `monitor_wait`. Como solución a esto se debe utilizar `monitor_notify` que despertará solo a una hebra, la primera que se puso a dormir con `monitor_wait`.

A pesar de utilizar una cola FIFO en la implementación de `monitor_wait` podría igualmente haber competencia entre procesos, ya que justo cuando se está despertando a una hebra (y solo a una, por el uso de FIFO) puede existir otra hebra que está ejecutando justamente un `monitor_enter`, en dicho caso ambas competirían.

Lo interesante es ver si esta solución con `monitor_notify` servirá para todos los casos, veremos a continuación que no sucede así, mediante el ejemplo del problema del productor consumidor.

5.6.2. Problema productor consumidor

```

1 void put(Item it)
2 {
3     monitor_enter (ctrl);
4     while (c==N)
5         monitor_wait (ctrl);
6     ...

```

```
7     monitor_notify_all (ctrl);  
8     monitor_exit (ctrl);  
9 }  
10 Item get()  
11 {  
12     monitor_enter (ctrl);  
13     while(c==0)  
14         monitor_exit (ctrl);  
15     ...  
16     monitor_notify_all (ctrl);  
17     monitor_exit (ctrl);  
18 }
```

Para el ejemplo anterior uno podría, intuitivamente, cambiar los `monitor_notify_all` por `monitor_notify`. Sin embargo, ¿podría darse la situación en donde hay tanto un productor y un consumidor en un `monitor_wait`? Si un consumidor ejecuta un `get` y despierta con `monitor_notify` a otra hebra, se esperaría que esa hebra fuese un productor, pero si hubiera tanto un productor como un consumidor se podría despertar el consumidor, que al revisar que no hay objetos se dormiría y el productor, que debiese haber despertado, seguirá durmiendo por que nadie le avisa que debe producir. ¿Utilizar un monitor para los productores y otro para los consumidores ayudaría a solucionar el problema recién planteado?, quizás si, pero en general el trabajo con dos monitores es complicado y propenso a interbloqueos.

Aquellos problemas donde no sirve simplemente que se despierte al primero que se fue a dormir, `monitor_notify` no será la solución. En muchos casos `monitor_notify_all` no puede ser reemplazado directamente por un `monitor_notify`. El caso de la implementación de semáforos es uno de los pocos casos donde funciona, ya que ahí justamente se requiere el orden FIFO que provee `monitor_notify`, despertando al primero que está esperando por el

monitor.

5.6.3. Solución de verdad: monitores de Hoare

Se utilizarán los tipos de datos `struct monitor_condition*` que básicamente representan una cola de tareas que se administra en orden FIFO. Estas colas están asociadas al monitor que se está utilizando. En el fondo se usan los mismos monitores, pero se añade una cola para que esperen los hilos que se están durmiendo, de esta forma al despertarlos se despierta a las hebras de una cola específica, no a “cualquiera”, esto solucionaría el problema del productor consumidor, donde con `monitor_notify` se podía despertar a cualquiera (productor o consumidor).

¿Se garantiza orden FIFO?, nuevamente no necesariamente, ya que se puede despertar a un consumidor que estaba durmiendo y justo en ese momento entrar un nuevo consumidor que ejecuta `monitor_enter` y ambos competir por la propiedad del monitor. Sin embargo los “wait” si son en orden FIFO, o sea se despertará al primero que llamó a `monitor_condition_make`. Es por esta razón que el `while` debe seguir estando presente, para seguir evaluando la condición una vez es despertada la tarea.

5.6.3.1. Problema productor consumidor

Ejemplo del productor consumidor con monitores de Hoare.

```

1 struct monitor *ctrl;           /* = monitor_make (); */
2 struct monitor_condition *noempty, /* = monitor_condition_make (ctrl); */
3     *nofull;      /* = monitor_condition_make (ctrl); */
4 void put(Item it)
5 {
6     monitor_enter (ctrl);
7     while(c==N)
8         monitor_condition_wait (nofull);

```

```
9     ...
10    monitor_condition_signal (noempty);
11    monitor_exit (ctrl);
12 }
13 Item get()
14 {
15    monitor_enter (ctrl);
16    while(c==0)
17        monitor_condition_wait (noempty);
18    ...
19    monitor_condition_signal (nofull);
20    monitor_exit (ctrl);
21 }
```

Existen variantes que pueden implementar `monitor_condition_signal_all`, lo cual sería el equivalente a `monitor_notify_all`.

Este tipo de monitores pueden presentar utilidad en algunos casos, como en este problema del productor consumidor, sin embargo, en general, se utilizarán los monitores de Brinch Hansen.

5.7. Ejercicios y preguntas

1. ¿Por qué utilizar `while(flag);` como solución en los problemas de sincronización es incorrecto?. Indique las dos razones.
2. Explique el concepto de espera activa.
3. Explique el concepto de espera pasiva.
4. Considere la solución siguiente para el problema de la cena de filósofos:

```
1 void filosofo (int i)
2 {
3     for (;;) {
4         comer(i, (i+1)%5);
5         pensar();
6     }
7 }
```

¿por qué es incorrecta?

5. En el problema de la cena de filósofos, ¿por qué usando semáforos puede ocurrir interbloqueo? ¿cómo se soluciona con semáforos?.
6. ¿Por qué en el problema de lectores escritores, varios lectores pueden ejecutarse simultáneamente pero no así los escritores?.
7. Mencione tres características de los semáforos.
8. En semáforos ¿se garantiza el orden en la entrega de los *tickets*?.
9. ¿Un monitor puede ser considerado como un semáforo?.
10. ¿Qué operación de monitores hace la diferencia entre un semáforo binario y un monitor?.
11. Un monitor permite consultar por un invariante sin dejar “tomado” el monitor en caso que este no sea satisfactorio para poder continuar, esto hace uso de un ciclo `while`. Este ciclo ¿es espera activa o pasiva?.
12. ¿Para qué se utiliza `monitor_notify_all`?.
13. En monitores ¿se garantiza la entrega de la propiedad del monitor?.
14. Indique el patrón de solución de problemas de sincronización usando monitores.

15. ¿Por qué los monitores pueden producir hambruna?.
16. En mensajes ¿se garantiza el orden en que se procesan los mensajes?.
17. ¿Quién debe responder a un mensaje?.

5.8. Referencias

- Sistemas Operativos, Segunda Edición, Andrew Tanenbaum, Capítulo 2.2 y 2.3.
- Sistemas Operativos, Quinta Edición, Abraham Silberschatz y Peter Baer Galvin, Capítulo 6 y 7.
- Sistemas Operativos, Segunda Edición, William Stallings, Capítulo 4 y 5.

Capítulo 6

Planificación de monoprocesadores

Durante este capítulo se asumirá que la máquina dispone de una única CPU, o sea, se discutirá la planificación en monoprocesadores. Considerando este escenario se analizarán diferentes métodos mediante los cuales el sistema operativo puede decidir que proceso ocupará el recurso CPU y será ejecutado.

El ***scheduling***, o planificación de procesos, corresponde a la asignación conveniente del recurso CPU a los procesos. Se debe elegir que proceso tomará la CPU y por cuanto tiempo. La recuperación de la CPU por parte del sistema operativo se hace mediante una interrupción del cronómetro regresivo. El ***scheduler*** de procesos es la componente en el núcleo, por lo tanto en el área de sistema, que realiza el *scheduling*.

Cada recurso deberá tener su propio planificador y la idea siempre será favorecer algún tipo de parámetro, como: tiempo de espera u orden de llegada. Al estudiar y comprender la planificación de procesos para su ingreso a CPU es posible realizar una analogía con lo que sucede en el resto de dispositivos de la máquina que requieren algún tipo de planificación para su acceso (donde lo que generalmente podría cambiar es el algoritmo de decisión).

Las **colas de *scheduling*** pueden ser, por ejemplo, colas FIFO o con prioridades, donde los procesos esperan por un recurso. El mejor ejemplo es la cola de procesos listos para ejecutarse en espera de CPU donde el *scheduler* deberá elegir un proceso de esta cola para la

ejecución. También hay colas para disco donde si un proceso requiere un dato de este deberá esperar en la cola hasta que el disco esté disponible para ser usado.

Las **jerarquías de *scheduling*** corresponden a los diferentes niveles de planificación que pueden existir dentro del sistema operativo, lo anterior según el proceso y recurso que se debe planificar.

I. **Corto plazo:** el distribuidor (*dispatcher* o *scheduler*) es el planificador de mayor frecuencia, el cual debe tomar decisiones con un mayor de detalle, ya que decide que proceso se ejecutará a continuación (entrará a la CPU). Se ejecuta cuando ocurre un suceso que lleva a la interrupción del proceso actual:

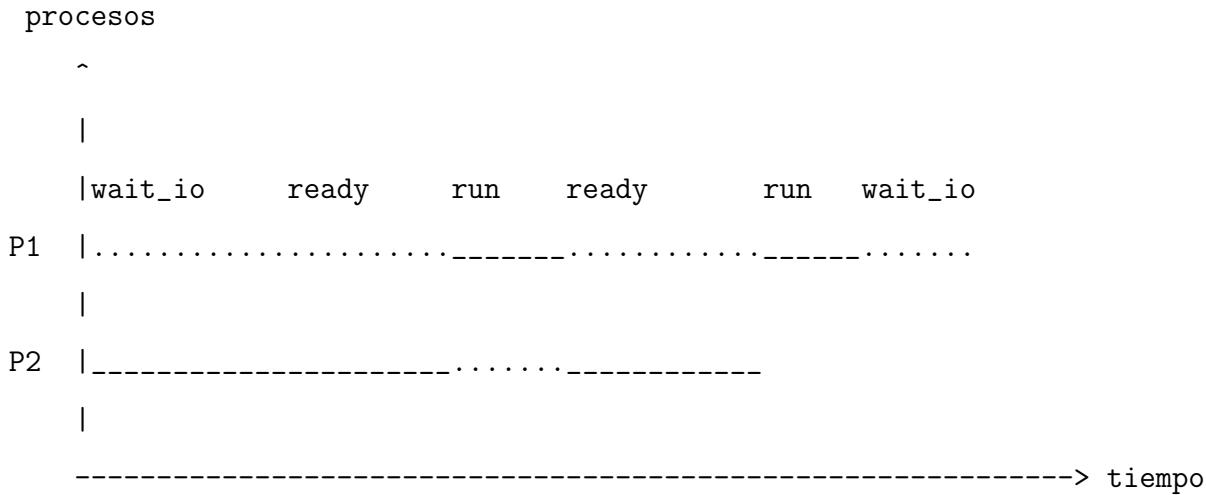
- Interrupciones del reloj.
- Interrupciones de E/S.
- Llamadas al sistema operativo.
- Señales.

II. **Mediano plazo:** elige quién va a la memoria o a disco (evitando procesos interactivos en disco). Encargado de manejar el intercambio entre los procesos suspendidos y no suspendidos. Al salir de la cola de mediano plazo vuelven a la cola a corto plazo para ahora esperar a entrar a la CPU.

III. **Largo plazo:** para procesamientos por lotes (o *jobs*). Una vez lanzado no podrá ser detenido por el *scheduling* de largo plazo y el *job* se ejecutará hasta el final. Es el que determina cuales son los programas que serán admitidos para ejecución en el sistema, independientemente de lo que suceda después en las otras colas (de mediano o corto plazo). Una vez aceptado el proceso pasará a la cola de corto plazo o mediano plazo (en caso de comenzar con menor prioridad o suspendido). En sistemas interactivos se aceptan todos los trabajos hasta que el sistema ya no puede atender a más procesos (dados por el nivel de multiprogramación y recursos del sistema).

Las **ráfagas** de CPU corresponden a una secuencia de instrucciones ejecutadas por un proceso sin pasar a un modo de espera.

Supongamos que en un instante determinado tenemos un proceso P_2 en ejecución y un proceso P_1 en un estado WAIT (debido a un requerimiento de E/S). En algún momento la operación de E/S concluirá pasando a estado READY, sin embargo el *scheduler* no le asigna la CPU de forma inmediata por lo cual sigue en estado READY.



Le interrupción (int) del término del proceso de E/S solicitado por el proceso P_1 le llegará al proceso P_2 , por lo cual la rutina de instrucción que atiende la interrupción se ejecuta en tiempo de sistema pero dentro del tiempo real de P_2 , y es en el tiempo de P_2 donde se pasa P_1 a estado READY. Todo esto sin existir cambio de contexto, ya que el tiempo que dura la interrupción es mucho menor al que toma el hacer un cambio de contexto para que P_1 atienda la interrupción.

Una vez P_2 entrega la CPU, por algún motivo, P_1 se ejecuta, si se asume un sistema con procesos donde el sistema operativo puede quitar la CPU, en algún momento a P_1 se le puede obligar a dejar la CPU, para volver a colocar a P_2 , luego P_1 podrá volver a ejecutarse nuevamente más adelante, esto seguirá sucediendo hasta que el proceso termine o bien exista alguna espera por E/S, a menos que alguna de estas dos últimas situaciones ocurra se dirá

que la ráfaga de CPU de P_1 serán los estados RUNNING consecutivos sin que exista WAIT entremedio.

Recordar que una ráfaga de CPU no es una secuencia ininterrumpida de instrucciones, sino que es una secuencia de instrucciones sin pasar a estados de espera.

La mayoría de las ráfagas de CPU son cortas en tiempo, si hacemos un histograma se podría observar algo similar a lo mostrado en la figura 6.1. Donde se observa una curva, donde los procesos con menores tiempos de ráfagas son procesos intensivos en E/S, y aquellos con altos tiempos de ráfagas son intensivos en CPU.

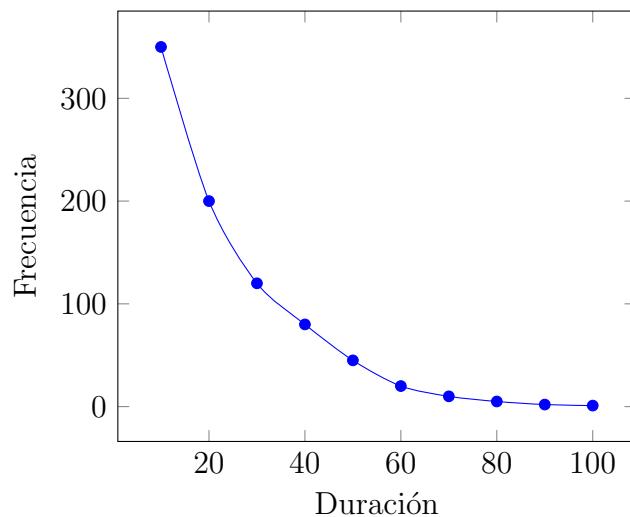


Figura 6.1: Ejemplo ráfagas de CPU

El objetivo es disminuir la cantidad de cambios de contextos que ocurren en el sistema y con esto minimizar el sobre costo de los mismos, para esto se debe fijar una tajada de tiempo adecuada. Según la figura 6.1 sabemos que la mayoría de las tajadas son de duración corta, entonces se les podría dar a los procesos una tajada de 40 [ms]. De esta forma un proceso tiene mayor probabilidad de terminar su ráfaga de CPU dentro de la tajada de tiempo, ya que se asume que la mayoría de los procesos tienen ráfagas menores a 40 [ms]. La estrategia es darle un tiempo lo suficientemente grande para que las ráfagas se ejecuten de forma continua, lo ideal es que esto cubra el 80 % de los casos. Esto corresponde a lo usado en un tipo de

scheduling el cual es **Round-Robin**.

El **tiempo de despacho** de un proceso corresponderá al tiempo desde que una ráfaga de CPU se encuentra disponible para su ejecución, o sea el proceso está en estado READY, hasta que la ráfaga fue atendida por completo. El ideal es minimizar en los algoritmos la media de los tiempos de cada uno de los procesos que se están atendiendo. Se definirá el tiempo de despacho total (la media de todos los procesos) como:

$$T_{despacho} = \frac{T_{P_1} + T_{P_2} + \dots + T_{P_n}}{n}$$

6.1. Algoritmos de planificación

Se deberán fijar criterios para realizar la planificación y de esta forma poder evaluar cada una de las estrategias posibles a utilizar. Estos criterios pueden ser:

- Tiempo de respuesta.
- Tiempo de retorno.
- Productividad.
- Utilización del procesador.
- Equidad.
- Prioridades.

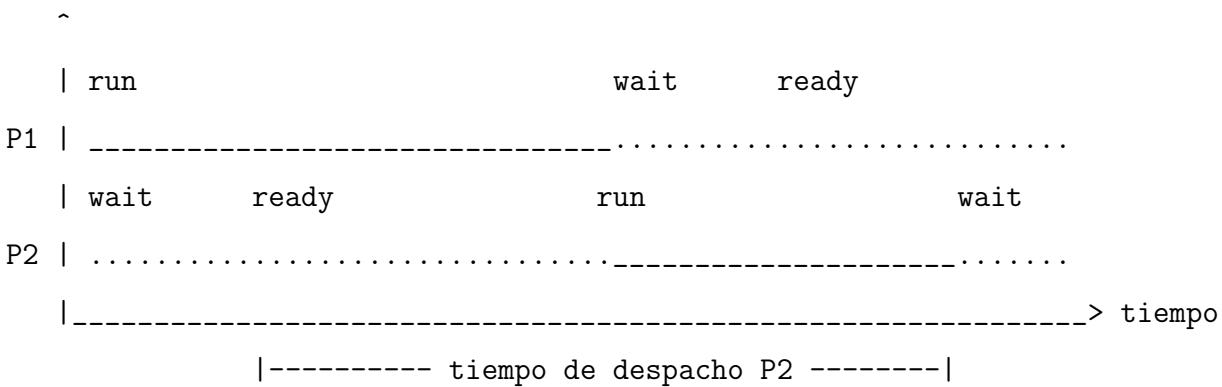
Los algoritmos de planificación están directamente relacionados con el tipo de procesos que se estará ejecutando, considerando los tipos *preemptive* y *non-preemptive*. En aquellas estrategias donde por algún motivo el sistema operativo puede quitar la CPU al proceso diremos que dicho algoritmo es del tipo apropiativo y si no la puede quitar será del tipo no apropiativo.

A continuación se describirán algunas de las estrategias de planificación que puede ser utilizadas en el caso del planificador a corto plazo.

6.1. ALGORITMOS DE PLANIFICACIÓN DE MONOPROCESADORES

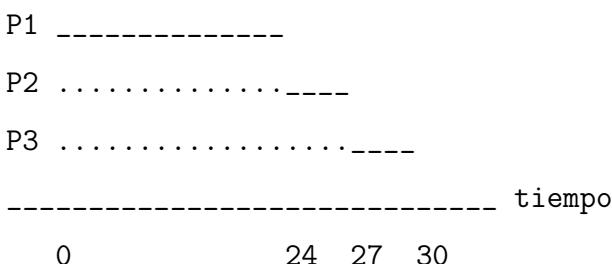
6.1.1. FCFS: First Come First Served

Corresponde al sistema de planificación donde la primera ráfaga de CPU que llega es la primera que se atiende, o sea, orden FIFO. La ráfaga es ejecutada hasta que termine su ejecución o bien se bloquee por E/S, por lo cual es un método de planificación *non-preemptive* (o no apropiativo), mientras el proceso no pase a un estado de espera, el sistema operativo no podrá utilizar la CPU.



Este método presenta el gran problema de que los tiempos de despacho son muy variables y generalmente largos, lo cual lo hace un sistema de planificación horrible para sistemas interactivos.

Ejemplo: se tienen 3 ráfagas de procesos de tiempos 24, 3 y 3, correspondientes a los procesos P1, P2 y P3 respectivamente que llegan en ese mismo orden a la cola de procesos listos. Se tendrá el siguiente esquema de planificación:



$$T_{despacho} = \frac{T_{24} + T_{27} + T_{30}}{3} = 27$$

Si se planifican de otra forma el tiempo de despacho será diferente, por ejemplo supongamos el orden P2, P3, P1:

$$T_{despacho} = \frac{T_{P1}+T_{P2}+T_{P3}}{Procesos} = \frac{T_{30}+T_3+T_3}{3} = 13$$

6.1.2. SJF: Shortest Job First

En este caso los programadores entregaban un tiempo que se esperaba que durara el trabajo (*job*), la idea de esto es minimizar la media del tiempo de despacho de los procesos. Si el tiempo de duración era mayor al esperado (ejemplo: el doble), se asumía que el proceso estaba caído y se mataba, por lo que el programador debía colocar un tiempo realista.

La idea es atender primero las ráfagas cortas, no atender el proceso completo de una vez, sino solo la ráfaga de CPU y con esto minimizar el tiempo de despacho de dicha ráfaga, minimizando lo que ocurría con FCFS. Para esto se debe estimar la duración de una ráfaga, la forma de hacer esto es con un promedio ponderado de la duración de ráfagas anteriores (esto mediante estadísticas del proceso que lleva el sistema operativo).

Sea:

- τ_{n+1}^p el predictor para la ráfaga $n + 1$ del proceso p , el cual indicará cuanto podría demorar la siguiente ráfaga de CPU del proceso p .
- τ_n^p la duración de la ráfaga n .
- α ponderados para las duraciones de las ráfagas anteriores.

Se puede calcular la duración estimada de la siguiente ráfaga de CPU de un proceso utilizando la siguiente fórmula, donde α típicamente es un valor como 0,5.

$$\tau_{n+1}^p = \alpha\tau_n^p + (1 - \alpha)\alpha\tau_{n-1}^p + (1 - \alpha)^2\alpha\tau_{n-2}^p + \dots + (1 - \alpha)^i\alpha\tau_{n-i}^p$$

El problema con la fórmula anterior, es que calcular todas las ráfagas anteriores de todos los procesos es algo muy lento, sin embargo esta fórmula se puede simplificar ya que es igual a:

6.1. ALGORITMOS DE PLANIFICACIÓN DE MONOPROCESADORES

$$\tau_{n+1}^p = \alpha \tau_n^p + (1 - \alpha) \tau_n^p$$

En el fondo se va utilizando el cálculo realizado para la ráfaga anterior. Caso $n = 0$ es una condición de borde que es un valor fijo no muy relevante, ya que al ir teniendo ráfagas se irá calculando el valor real para el proceso.

Este método puede ser *preemptive* o no. Sin embargo si es *preemptive* se le asigna una cota al proceso a través del *timer* para introducir un cambio de contexto, donde lo lógico sería utilizar como tiempo máximo la siguiente ráfaga más corta o bien el predictor calculado, donde si el proceso excede dicho valor se le quita la CPU. En este caso se “cubren” posibles errores que hayan ocurrido al realizar la predicción de la duración de la siguiente ráfaga del proceso.

Funciona mejor en sistemas interactivos, sin embargo recordar que la motivación de este método es reducir el tiempo de despacho.

El problema de esto es la hambruna para las ráfagas largas. Esto sucede si llegan constantemente ráfagas cortas a la cola de listos y nunca se pueden atender ráfagas más largas.

6.1.3. Primero el de menor tiempo restante

Este caso es muy similar a SJF, sin embargo aquí se evalúa el tiempo que le queda a cada proceso para terminar su ráfaga y siempre se ejecutará la ráfaga más corta. Esto significa que si tenemos un proceso en ejecución y entra un proceso a la cola de listos cuya ráfaga siguiente es más corta que la que está actualmente ejecutándose habrá un cambio. Por lo anterior este método es apropiativo.

6.1.4. Primero el de mayor tasa de respuesta

En este tipo de estrategia se define una tasa de respuesta (o *response ratio*) como $RR = (w + s)/s$, donde w es el tiempo que el proceso lleva esperando por el procesador y s es el tiempo total de servicio esperado para la ráfaga que se está atendiendo.

Este método entregará la “edad” del proceso, donde entrarán al procesador aquellos que lleven más tiempo esperando por el uso de CPU.

Si en algún momento el proceso que se está ejecutando deja de ser el de mayor “edad”, porque otro lleva más tiempo esperando, la CPU le será quitada y asignada al otro proceso. Por esta razón esta política es apropiativa.

6.1.5. Prioridades

En este método se ejecuta de la cola de listos el proceso con mejor prioridad. Se manejarán diferentes colas listas, cada una representará un nivel de prioridad diferente y los procesos de prioridades inferiores no serán atendidos hasta que se hayan atendido los de la prioridad superior, ver figura 6.2.

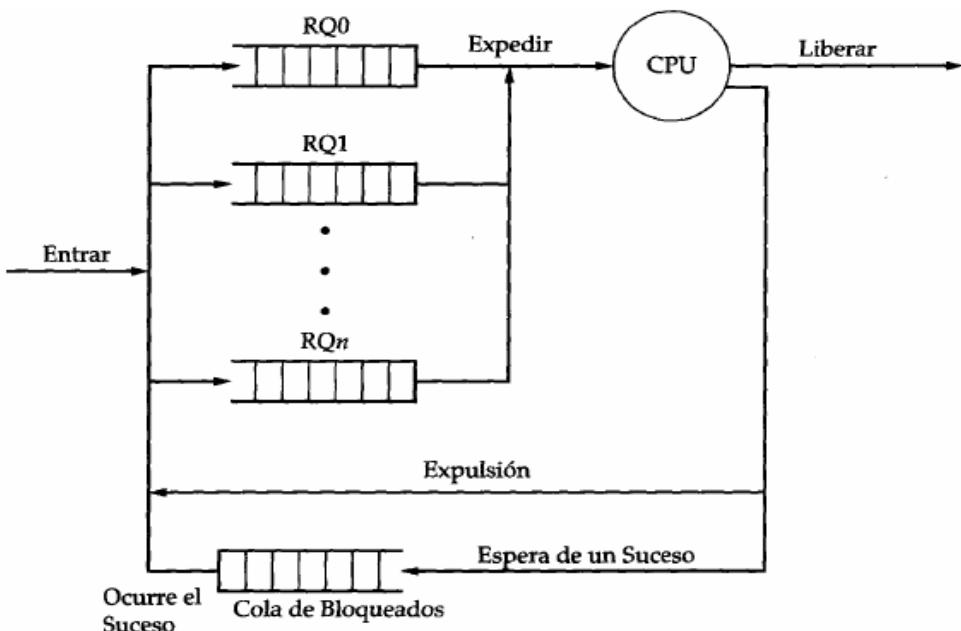


Figura 6.2: Planificación considerando colas de prioridades (RQ_0 a RQ_n)

Los casos anteriores (excepto FCFS) son casos particulares de prioridades, por ejemplo en SJF la prioridad del proceso es su predictor τ_{n+1}^p .

6.1. ALGORITMOS DE PLANIFICACIÓN DE MONOPROCESADORES

La idea de utilizar prioridades pretende que el sistema pueda ser sensible frente a las situaciones que en el están ocurriendo, donde un sistema podrá ir adecuándose a la forma en la que se van ejecutando los procesos.

En general el problema de esta solución es la hambruna, donde procesos con peor prioridad pueden no ser atendidos nunca.

La solución para la hambruna consiste en utilizar una variante llamada *aging* (añejamiento), donde cada cierto tiempo se mejora temporalmente la prioridad de todos los procesos que están en la cola listos. Ejemplo, cada 10 [ms] hay una interrupción y se mejora la prioridad a todos los procesos, con esto se espera que después de un tiempo X el proceso alcanzará la prioridad necesaria para entrar a la CPU. Una vez se concede el acceso a la CPU y el proceso sale, este retoma su prioridad original.

Otra técnica para evitar la situación de hambruna es el uso de prioridades con retroalimentación, donde una vez que el proceso ha sido atendido por la CPU al salir de esta pasará a una cola de menor prioridad. Esto con el objetivo de permitir que otros procesos puedan entrar a la CPU, ver figura 6.3.

6.1.6. Round Robin

Se busca minimizar el tiempo de respuesta y es la estrategia de planificación utilizada hoy en día. Se diseñó específicamente para sistemas de tiempo compartido, los cuales típicamente son interactivos.

El **tiempo de respuesta** en sistemas interactivos es el tiempo que transcurre desde que el usuario inicia una interacción hasta observar el primer resultado. Si un comando entrega una serie de líneas, el tiempo entre línea y línea será el tiempo de respuesta.

Psicológicamente se mostró que al usuario le interesa ir viendo resultados, o sea ir viendo que aparecen líneas avanzando en la pantalla. Un comando que se demora 10 segundos, pero estuvo entregando durante los 10 segundos respuestas es mejor, psicológicamente, que un comando que duro 5 segundos, pero durante esos 5 segundos no se entregó ninguna respuesta

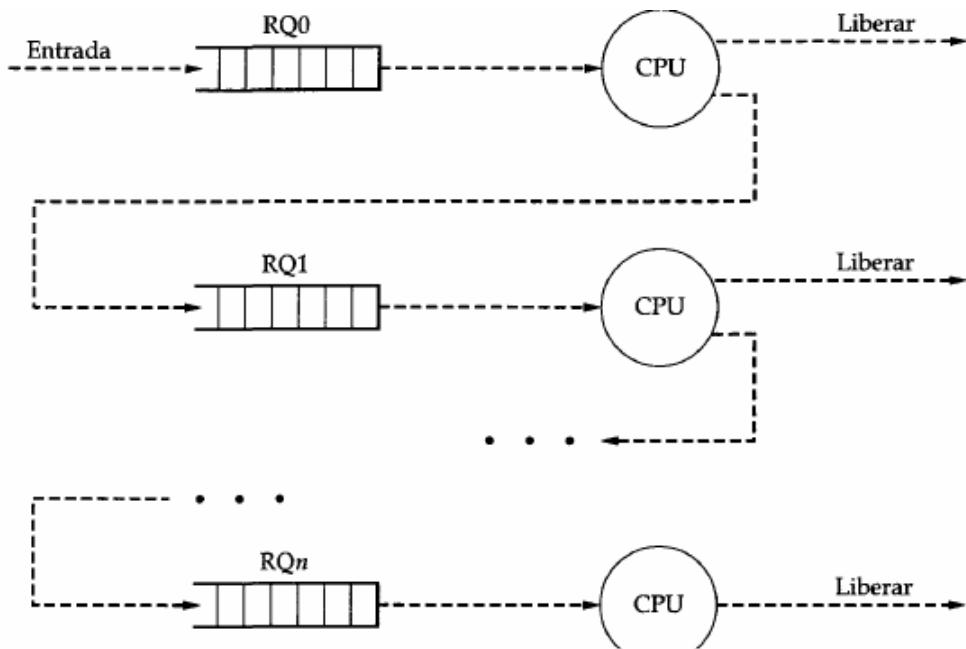


Figura 6.3: Planificación considerando colas de prioridades con realimentación

por la pantalla, sino solo hasta el final.

En este método de planificación los procesos forman una lista circular y el *scheduler* da tajadas de tiempo, típicamente de 10 a 100 [ms], a cada uno de los procesos.

¿Cuál es el tamaño de la tajada de tiempo? Si es muy grande se parece a FCFS, si es muy pequeña habría un costo excesivo en el cambio de contexto. Regla empírica, el 80 % de las ráfagas de CPU deben durar menos que el tiempo de la tajada.

En este tipo de *scheduling* la decisión crítica es **¿qué hacer con las ráfagas que llegan?**, ya que cuando la ráfaga llega uno podría proponer:

- i. Se otorga inmediatamente la CPU a dicho proceso con la tajada de tiempo completa (de bloqueado a listo), esto tiene sentido porque un proceso que estaba en estado de espera, el mismo cedió la CPU para ingresar al estado de espera. El problema, según como sea el *scheduling*, procesos que son intensivos en lectura escritura podrían causar hambruna a los que son intensivos en CPU.

- II. Para evitar la hambruna, se puede adoptar que cuando un proceso pase a estado de espera, al recibir el recurso, no recibe la tajada completa, sino lo que le quedaba de tajada. El problema con esto, hay muchos cambios de contexto si los procesos son intensivos en E/S.
- III. Otra variante, para minimizar el cambio de contexto, el proceso se deja para más adelante ubicándolo al final o al inicio. Cada una con sus problemas, la primera lo desfavorecería, la segunda podría provocar cierta hambruna en procesos que son intensivos en CPU.

El debate es ¿cómo minimizar el sobre costo de cambios de contexto dando uso equitativo de la CPU?

6.2. Planificación en Linux

El *scheduling* en sistemas Unix se realiza utilizando el sistema de prioridades más el uso de *aging* (para evitar hambruna). En caso de procesos de misma prioridad se utiliza Round Robin.

Para efectos de planificación los procesos son clasificados en 3 grupos:

- Procesos batch: sin interacción con el usuario, generalmente en segundo plano (ej: compiladores, base de datos).
- Procesos interactivos: interacción continua con los usuarios, deben ser procesos atendidos rápidamente (ej: editor de textos).
- Procesos de tiempo real: tiempo corto de respuesta (ej: video, audio, sensores externos).

Es difícil determinar si un proceso es batch o interactivo, por lo cual se utilizan estadísticas basadas en el comportamiento previo de un proceso para determinar a qué tipo corresponde.

Además los procesos pueden ser clasificados como:

- Procesos convencionales: interactivos y batch.
- Procesos no convencionales: tiempo real.

“Para poder determinar qué proceso se debe ejecutar a continuación, el planificador de Linux busca en la lista no vacía con la prioridad estática más alta y toma el proceso a la cabeza de dicha lista. La política de planificación determina para cada proceso, dónde se insertará en la lista de procesos, con qué prioridad estática y cómo se moverá dentro de esta lista.”¹

Los algoritmos de scheduling encontrados dentro del núcleo Linux son:

- SCHED_FIFO: cola estándar para procesos en tiempo real, mientras no haya un proceso de prioridad mayor el proceso tendrá la CPU.
- SCHED_RR: tiempo compartido en tiempo real, asignación justa a procesos con igual prioridad.
- SCHED_OTHER: planificador de tiempo compartido universal.
- SCHED_BATCH: para procesos convencionales cuando el procesador está “inactivo”.

6.2.1. Planificación de procesos convencionales

Para asignar la prioridad se monitorean los procesos y según lo que van haciendo son favorecidos o penalizados en su prioridad (dependiendo de si han obtenido o no la CPU). Se distinguen 2 tipos:

- Estática: asignada inicialmente a un proceso, no responde a los cambios del sistema (heredada del proceso padre).
- Tiempo real o dinámica: responde a los cambios del sistema, y al comportamiento del proceso.

¹Citado textualmente desde man

Ambas prioridades se mueven entre los valores 100 (mejor prioridad) y 139 (peor prioridad).

6.2.1.1. Prioridad estática

Para procesos convencionales se utiliza una prioridad estática, que va desde 100 a 139. Esta se verá afectada por el comando *nice* o la llamada a sistema *setpriority*. Esta prioridad ayudará a determinar el “quantum” base del proceso, de tal forma que:

- $PE < 120 \Rightarrow quantum = (140 - PE) * 20[ms]$
- $PE \geq 120 \Rightarrow quantum = (140 - PE) * 5[ms]$

6.2.1.2. Prioridad dinámica

La prioridad dinámica es calculada según el comportamiento del proceso en el sistema, donde, según como sea este, recibirá un *bonus* para mejorar o empeorar su prioridad. Al igual que con la prioridad estática su valor va entre 100 y 139. El planificador buscará por esta prioridad al momento de elegir un proceso para que entre a la CPU.

Se define como: $PD = \max(100, \min(PE - bonus + 5, 139))$, donde el bonus es un valor de 0 a 10, relacionado con el tiempo de *sleep* promedio del proceso.

Ejemplo: $PD = \max(100, \min(120 - 5 + 5, 139)) = 120$

El **tiempo de sleep** corresponde a un promedio del tiempo que el proceso ha pasado durmiendo (sin entrar a la CPU). Este tiempo decrece mientras un proceso esta corriendo y será como máximo 1000[ms], independientemente del tiempo real que lleve sin entrar a CPU. El bonus se define utilizando el tiempo de *sleep* como : $bonus = \text{floor}(\text{tiempo}/100)$.

Utilizando este tiempo de *bonus* se puede determinar si un proceso es interactivo o batch utilizando la siguiente fórmula $bonus - 5 \geq PE/4 - 28 \Rightarrow \text{interactivo}$. Por la fórmula anterior tenemos que un proceso con prioridad estática *default* (o sea 120) al iniciar no será

considerado interactivo, solo aquellos con una prioridad mejor (o sea 119 o menos). ¿Cuándo un proceso con prioridad *default* será considerado interactivo?

6.2.2. nice

nice permite empeorar (o mejorar) la prioridad estática de un proceso, esto permitirá ajustar el valor entre 100 y 139, sin embargo para mejorar la prioridad se requieren privilegios de administrador (usuario *root*). Lo anterior ya que la idea original de *nice* era que un usuario fuese amable (*nice*) con otros al bajar la prioridad de sus procesos.

Los parámetros de la instrucción van de -20 a 19, donde, como ya se mencionó valores negativos solo puede asignarlos *root*. La nueva prioridad se calculará como $P_{Enueva} = P_{Edefault}(120) + nice$.

6.2.2.1. Ejemplo

A continuación se adjunta un ejemplo en C que afecta la prioridad estática de un proceso. Se sugieren distintas formas de ejecución con diferentes resultados.

```
1 #include <sys/time.h>
2 #include <sys/resource.h>
3 #include <stdlib.h>
4 #include <stdio.h>
5 int main (int n, char* args[]) {
6     // crear variables para guardar las prioridades
7     int prioridadActual, prioridadNueva;
8     // determinar prioridad nueva
9     if (n==2) prioridadNueva = atoi(args[1]);
10    else prioridadNueva = 0;
11    // mostrar prioridad con que se llamo al proceso
```

6.3. EJERCICIOS Y PREGUNTAS 6. PLANIFICACIÓN DE MONOPROCESADORES

```
12     prioridadActual = getpriority(PRIO_PROCESS, 0) + 120;
13     printf("Prioridad original: %d\n", prioridadActual);
14     // cambiar prioridad y mostrarla
15     setpriority(PRIO_PROCESS, 0, prioridadNueva);
16     prioridadActual = getpriority(PRIO_PROCESS, 0) + 120;
17     printf("Prioridad cambiada: %d\n", prioridadActual);
18     // salir del programa
19     return EXIT_SUCCESS;
20 }
```

- I. Compilar: \$ gcc -Wall prioridad.c -o prioridad
- II. Ejecutar normal: \$./prioridad
- III. Ejecutar con nice: \$ nice -n 10 ./prioridad
- IV. Pasar prioridad como parámetro: \$./prioridad 5
- V. Ejecutar con nice y pasar prioridad como parámetro: \$ nice -n 7 ./prioridad 3

¿Qué pasa en el último caso con el valor 3?

6.3. Ejercicios y preguntas

1. ¿Cuál es la función de las políticas de *scheduling*?
2. ¿Qué tipo jerarquías de planificación existen? Explíquelas.
3. ¿Qué jerarquía es la encargada de elegir el proceso que entrará a la CPU?.
4. ¿Qué condición se debe dar en la ejecución de un proceso para que se considere que dicha ejecución corresponde a solo una ráfaga de CPU?.

5. ¿Qué es el tiempo de despacho de un proceso?
6. Nombre tres criterios que se pueden utilizar para determinar la estrategia de planificación a utilizar.
7. ¿Cuál es la diferencia entre políticas de planificación apropiativas y no apropiativas?
8. ¿FCFS es apropiativo o no apropiativo?
9. ¿Cuál es el gran problema de FCFS?
10. En FCFS, los procesos ¿son todos atendidos?
11. El orden en que llegan las ráfagas en FCFS ¿afectará el tiempo de despacho? Explique con un ejemplo.
12. “Primero el trabajo más corto”, “primero el de menor tiempo restante” y “primero el de mayor tasa de respuesta” ¿son casos particulares de qué estrategia de planificación?.
13. En SJF, ¿qué es y qué representa el predictor de la ráfaga?.
14. La tasa de respuesta (o *response ratio*) de un proceso, ¿con qué esta relacionada?.
15. ¿Por qué la estrategia de prioridades presenta hambruna?, explique.
16. ¿Qué técnica se puede utilizar en una planificación con prioridades para evitar la hambruna?.
17. ¿Por qué es crítico definir de forma correcta la tajada de tiempo en Round Robin?.
18. ¿Qué tipo de planificación se utiliza en el sistema operativo Linux?.
19. ¿Para qué es utilizada la prioridad estática de un proceso en Linux?.
20. ¿Para qué es utilizada la prioridad dinámica de un proceso en Linux?.

21. En la prioridad dinámica se utiliza un *bonus* para mejorar o empeorar esta, ¿de qué depende este *bonus*?.
22. ¿Cuál es la prioridad estática por defecto?.
23. ¿Qué usuarios pueden mejorar la prioridad estática de sus procesos?.

6.4. Referencias

- Sistemas Operativos, Segunda Edición, Andrew Tanenbaum, Capítulo 2.4.
- Sistemas Operativos, Quinta Edición, Abraham Silberschatz y Peter Baer Galvin, Capítulo 5.
- Sistemas Operativos, Segunda Edición, William Stallings, Capítulo 8.

Capítulo 7

Memoria principal

Todo programa que se quiera ejecutar dentro del sistema, o sea convertirse en un proceso, requerirá como mínimo dos recursos, utilizar la CPU para ejecutar su código y utilizar memoria RAM para almacenar su código y datos. El primer tema fue discutido en el capítulo 6, la asignación de RAM será discutida en este capítulo donde se abordarán los dos temas principales de administración de memoria principal, correspondientes a direcciones virtuales y memoria virtual.

Es importante recalcar que todo proceso que quiera pasar por la CPU requiere estar cargado en memoria principal. Esto implicará que al inicio del proceso el programa debe ser llevado desde el disco (almacenamiento secundario) a la RAM (memoria principal). Adicionalmente es importante recordar las velocidades de operación de los tipos de almacenamientos existentes y sus capacidades se verán afectadas dependiendo de si se trata de discos, ram, cache o registros de cpu, ver figura 7.1.

Finalmente se deberán considerar aspectos relacionados con la protección de la memoria principal, donde el sistema operativo deberá garantizar que solo los procesos legítimos puedan hacer uso de cierto espacio de direcciones, evitando que cualquiera pudiese leer o escribir en cualquier área de la memoria.

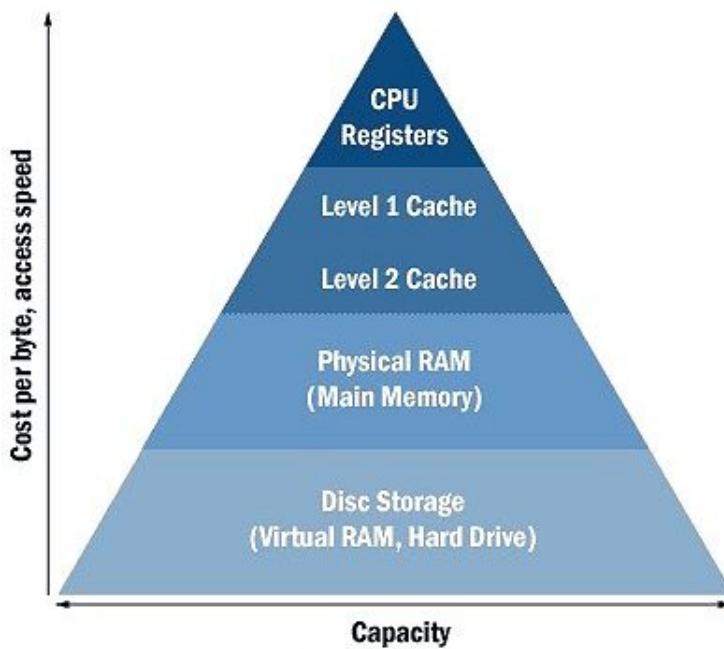


Figura 7.1: Tipos de memorias (tiempos, tamaños y costos)

7.1. Espacio de direcciones

Originalmente en los “sistemas operativos” la administración de la memoria era bastante simple ya que al existir un solo proceso ejecutándose al mismo tiempo este se podía copiar completamente a la memoria principal y acceder a los registros de la misma de forma directa. Sin embargo lo interesante es ver lo que ocurre en un sistema que utiliza multiprogramación, donde pueden existir diversos procesos residentes en memoria principal y a todos se les debe asignar un espacio de memoria para poder ser ejecutados.

La forma más simple de asignar la memoria a diversos procesos es simplemente tomar todo el programa existente en disco y copiarlo como un solo bloque a la memoria principal, en cuyo caso para determinar donde se encuentra el proceso guardado en la memoria basta conocer la dirección inicial, llamada **base**, y el tamaño del bloque, llamado **límite**, en la figura 7.2 tenemos una $base = 300040$ y un $limite = 120900$. Para movernos dentro del bloque del proceso en la memoria utilizamos un *offset* o **desplazamiento**, el cual debe dar

como resultado siempre una dirección dentro del bloque que el proceso tiene asignado, de forma contraria ocurriría un error de protección o ***segmentation fault***, o sea si la dirección es menor a la base o mayor a la base más el límite ocurrirá un error, ver figura 7.3

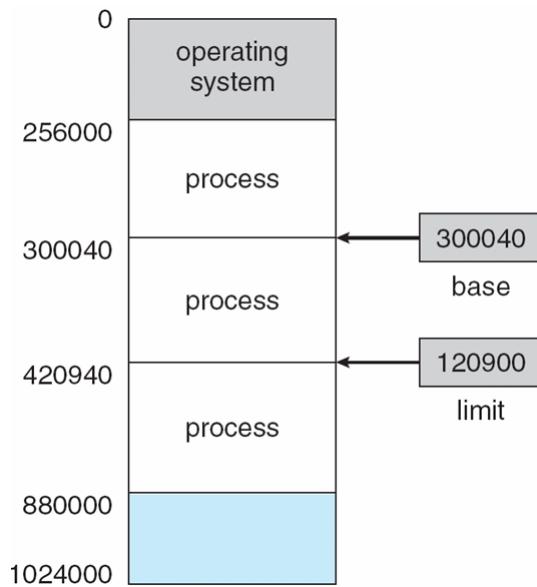


Figura 7.2: Registro base y límite de un bloque de memoria

El tipo de asignación descrito anteriormente y mostrado en la figura 7.2 corresponde a la **asignación contigua**, donde todo el proceso es ubicado en un único bloque de memoria física. De esta forma se ubicarán diversos procesos en la memoria, pero todos en un único bloque, esto implica que si no existe espacio contiguo para un proceso deberá sacarse alguno de los existentes, ver figura 7.4, esto último se conoce como intercambio y será discutido más adelante.

7.1.1. Enlace de direcciones

Una vez se escribe un programa debe ser traducido a código de máquina, ya que el computador solo entiende direcciones de memoria, no sabe de nombres de variables ni mucho menos de su semántica. Esto significa que cada una de las variables que se utilizan dentro

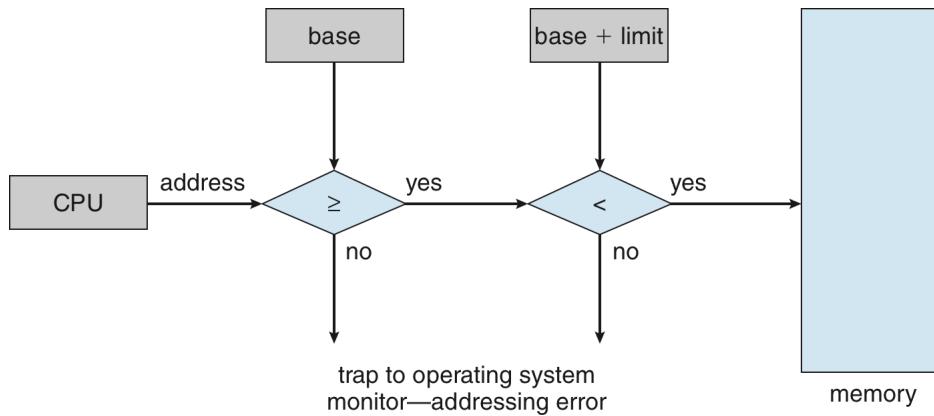


Figura 7.3: Protección de memoria, se limita el acceso al bloque del proceso

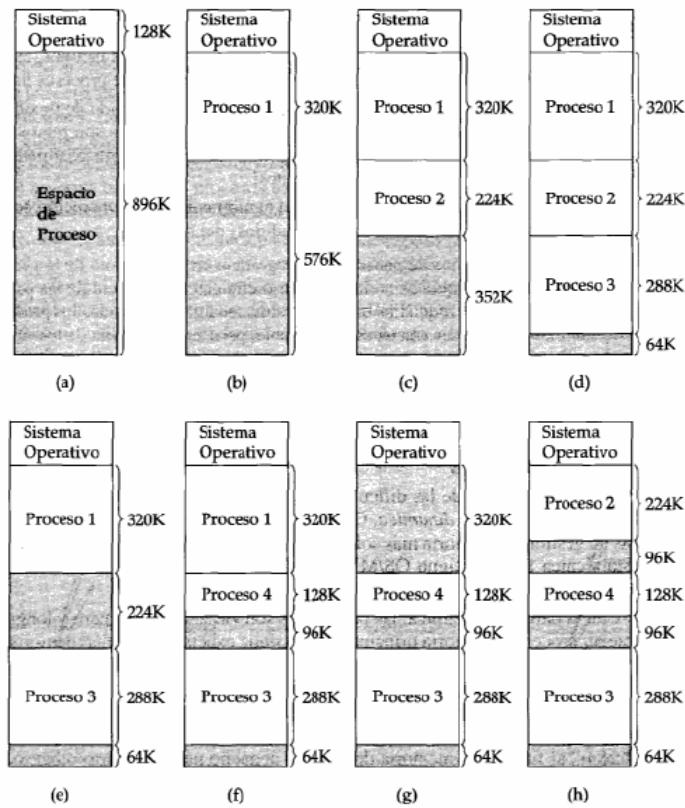


Figura 7.4: Asignación contigua

del código, y el mismo código, debe ser mapeado a direcciones de memoria para poder ser utilizado. Lo anterior se conoce como **enlace de direcciones** y corresponde al mapeo entre lo que forma el programa (código y datos) y las direcciones donde se encuentran dichos componentes. Existen diferentes momentos donde realizar el enlace, cada uno de ellos será mencionado a continuación y se pueden observar en la figura 7.5.

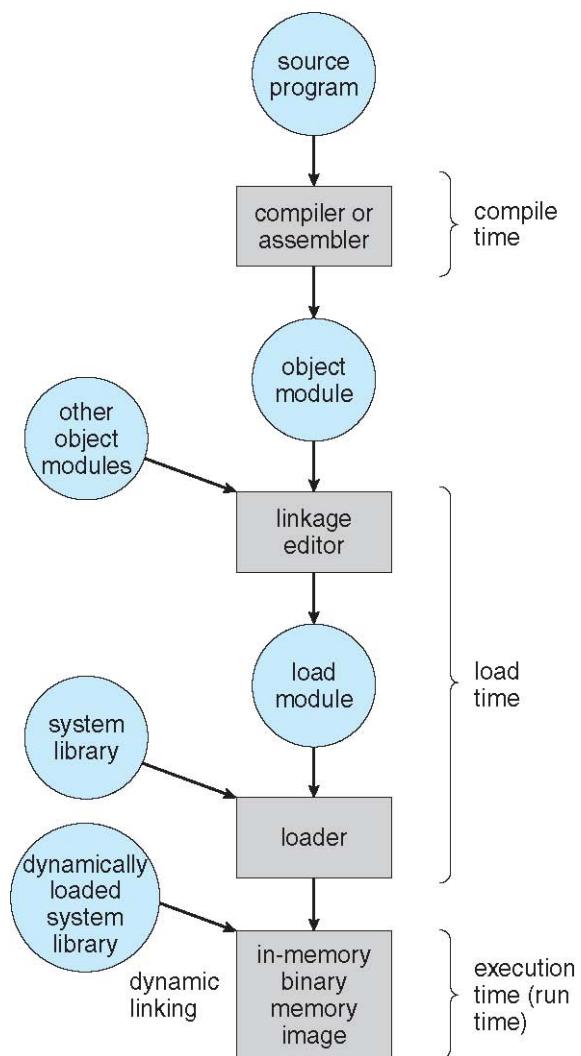


Figura 7.5: Tipos de enlaces de direcciones de memoria principal

7.1.1.1. Enlace en tiempo de compilación

Las direcciones de memoria principal son especificadas al momento de compilar el programa, aquí se indicará donde será cargada cada una de las instrucciones y variables del código del programa.

Esto tiene diferentes problemas:

- El proceso no podrá ser cargado si el bloque que requiere está parcialmente o totalmente ocupado.
- El proceso no podrá ser movido a otro bloque de memoria una vez sea cargado.
- Se deberán conocer las características físicas de la memoria principal disponible.

Este método tenía sentido en sistemas sin multiprogramación, donde solo un proceso se encontraba cargado en la ram en todo momento.

7.1.1.2. Enlace en tiempo de carga

En este método el programa generado al compilar no tiene las direcciones de memoria especificadas, será el sistema operativo el que al cargar el proceso (estado inicio) asignará las direcciones. De esta forma en diferentes ejecuciones el programa podría ser ubicado en bloques de memoria diferentes.

El problema aquí seguirá siendo que una vez cargado el programa no podrá ser movido a otro bloque de memoria. Si por ejemplo se quisiera utilizar intercambio (al igual que en el caso anterior) el proceso deberá volver al mismo espacio de direcciones físicas de donde fue sacado, lo cual obviamente representará una tremenda ineficiencia ya que dicho bloque no podría estar nunca más disponible y el proceso podría morir de hambruna.

7.1.1.3. Enlace en tiempo de ejecución

En los dos casos anteriores el proceso una vez era cargado en memoria no podía ser movido a otro bloque, lo que dificultaba que el proceso pudiese crecer (solo lo haría si hubiesen direcciones contiguas, después de su límite, libres) y prácticamente imposible hacer intercambio (ya explicado anteriormente).

En este tipo de enlaces el mapeo entre direcciones y los componentes del programa se realiza en tiempo de ejecución y a medida que el programa va cambiando (porque necesita crecer o porque es intercambiado) las direcciones de memoria irán cambiando.

El problema de este método es que la administración de la memoria netamente por software sería muy complicada y costosa, por las grandes referencias que se deberían manejar e ir modificando a lo largo de la vida del proceso. Para solucionar esto y permitir que el enlace pueda ser realizado en tiempo de ejecución se requiere soporte del hardware, específicamente de la MMU.

7.1.2. Direcciones virtuales y direcciones físicas

Uno de los problemas de la multiprogramación es que al existir múltiples procesos residentes en memoria, si cada proceso utilizará las direcciones de memoria física para sus enlaces las referencias serían complicadas.

Pensemos por un momento que existe una forma de dividir el programa, de tal forma que su asignación en memoria no es contigua, el programa se encontraría dividido por toda la memoria principal, sin embargo para ejecutarse necesita un espacio de direccionamiento contiguo ya que no se puede cortar un bloque de datos en un punto para luego continuarlo en otra parte de la memoria física.

Otro escenario interesante donde existen problemas es que sucede si necesitamos ejecutar un programa que requiere más memoria principal de la que disponemos.

Los casos anteriores se simplifican enormemente al utilizar el concepto de direcciones

virtuales y físicas, donde el espacio de direccionamiento virtual será contiguo para cada proceso, sin embargo el mapeo de dicho espacio virtual no será necesariamente contiguo en la memoria física.

Adicionalmente, al ser espacio virtual, se podrían tener más direcciones virtuales que las físicamente soportadas. Esto tendrá sentido más adelante cuando se vean paginación y segmentación, sin embargo se puede adelantar que un proceso solo mantendrá cargado lo que necesita y no todo el código o datos del programa. Esto permitirá cargar programas más grandes que la memoria principal disponible mediante el uso de direcciones virtuales.

Este mismo concepto es el requerido para el enlace en tiempo de ejecución, donde un proceso verá el mismo esquema de direcciones virtuales siempre, pero será la MMU la que se encargará de actualizar las referencias físicas de dichas direcciones virtuales. Este proceso es totalmente transparente para el proceso y no se enterará en caso que hayan cambios o lo muevan entre bloques físicos de memoria.

7.1.3. Unidad de administración de memoria MMU

La unidad de administración (o gestión) de memoria o MMU (del inglés *Memory Management Unit*) corresponde a la unidad que permite traducir direcciones virtuales a direcciones reales de memoria principal. De esta forma un proceso tiene una vista virtual de la memoria, donde varios procesos podrían tener visión de una misma dirección virtual, la cual es mapeada a direcciones reales diferentes.

Procesadores de computadores personales iniciales, 8088 o 68000, no contaban con MMU, por lo cual al no contar con MMU dichas máquinas no podían correr sistemas operativos Unix (ya que no había fork lo que implica que no hay shell). Esta característica (MMU) si estaba disponible en los *mainframes* (desde fines de los 60s). Con los procesadores 386 (Intel) y 68030 (Motorola) estuvo disponible una MMU en computadores personales a mediados de los 80s.

Sistemas embebidos, como DVD, lavadoras, etc, generalmente no incluyen MMU, ya que

existen menores posibilidades de caídas en el software. En caso que este se “caiga” se tiene que reiniciar el sistema, apagándolo, desenchufando o presionando el botón de encendido por X¹ segundos.

7.2. Asignación no contigua

A continuación se discutirán métodos de asignación no contigua de la memoria principal. Recordar que esto aplica a las direcciones físicas, ya que el espacio de direcciones virtuales siempre será contiguo.

7.2.1. Segmentación

Este tipo de división ya no se utiliza pero es pedagógicamente interesante ya que es una forma simple de dividir la memoria. Cada proceso necesita al menos 4 segmentos para funcionar, los cuales corresponden al código, los datos, la pila y el espacio del sistema.

En la figura 7.6 se puede apreciar la división del espacio de memoria de un proceso, notar que en esta imagen el espacio de datos se encuentra separado entre las variables globales y el espacio para solicitud dinámica de memoria (*malloc*). También notar que hay un espacio para bibliotecas compartidas, de esto se hablará más adelante y tendrá sentido al ver paginación.

A continuación se describe cada uno de los cuatro segmentos:

I. Código

- Instrucciones binarias del programa.
- Solo lectura.
- Inicio cercano a la dirección 0, por ejemplo 128 Kb.

II. Datos

¹Típicamente 4 segundos.

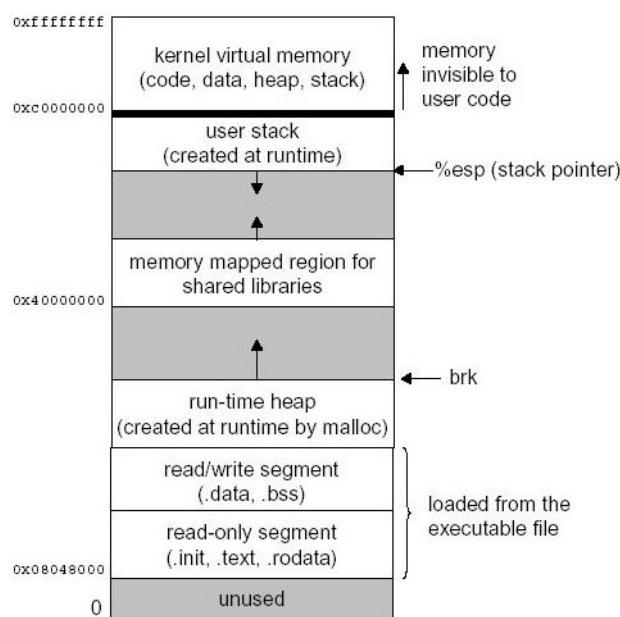


Figura 7.6: Registro base y límite de un bloque de memoria

- Variables globales inicializadas.
- Variables globales no inicializadas.
- heap para malloc y free.
- Lectura y escritura.

III. Pila

- Variables locales.
- Información para volver a la rutina que llamo a la función.
- Lectura y escritura.

IV. Sistema

- No se puede leer ni escribir en modo usuario.
- Segmento compartido entre todos los procesos.

- Inicia en todos los procesos en la misma dirección de memoria.

Cada uno de los cuatro segmentos será almacenado de forma contigua en la memoria física, o sea si el programa esta dividido en cuatro segmentos en la memoria física habrán cuatro segmentos. Se pueden tener los cuatro segmentos separados en la memoria real, ya que al utilizar direcciones virtuales todo el proceso se verá continuo en este espacio lógico.

Notar que todos los procesos comparten la memoria correspondiente a la parte de sistema (núcleo), sin embargo solo pueden acceder a ella en modo sistema, no en modo usuario.

Entre la pila y el sistema se encuentra una zona de espacio de memoria no asignada utilizada para crecer al ir solicitando más memoria con sbrk². Por razones históricas existe el *segmentation fault*, sin embargo actualmente debiese ser *page fault*.

El sistema real, la máquina, tendrá una cierta cantidad de memoria física, por ejemplo 4 Mb, entonces un proceso verá sus direcciones virtuales (descritas anteriormente) mapeadas a las direcciones reales en la memoria física disponible.

La traducción de direcciones virtuales a reales será realizada mediante la tabla de segmentos, donde existe una tabla por cada proceso (que pertenece a su contexto). Dependiendo de la implementación de la tabla de segmentos se pueden guardar diferentes registros (direcciones) todas apuntando a poder obtener la dirección física en memoria principal a partir de una dirección virtual.

En la figura 7.7 se puede observar el proceso de traducción de una dirección virtual utilizando segmentos a una dirección física. Importante mencionar que la dirección lógica entregará el segmento al que corresponde y un desplazamiento dentro de dicho segmento. De esta forma podríamos tener una dirección lógica dentro del segmento de datos y otra dentro de la pila, ambas con el mismo desplazamiento. Una vez se tiene el segmento se busca en la tabla de segmentos si dicho segmento tiene asociada una dirección base (física) asignada, si lo tiene se suma a su desplazamiento (previa verificación que este dentro del límite del segmento) y se obtiene la dirección real en la memoria secundaria.

²man sbrk

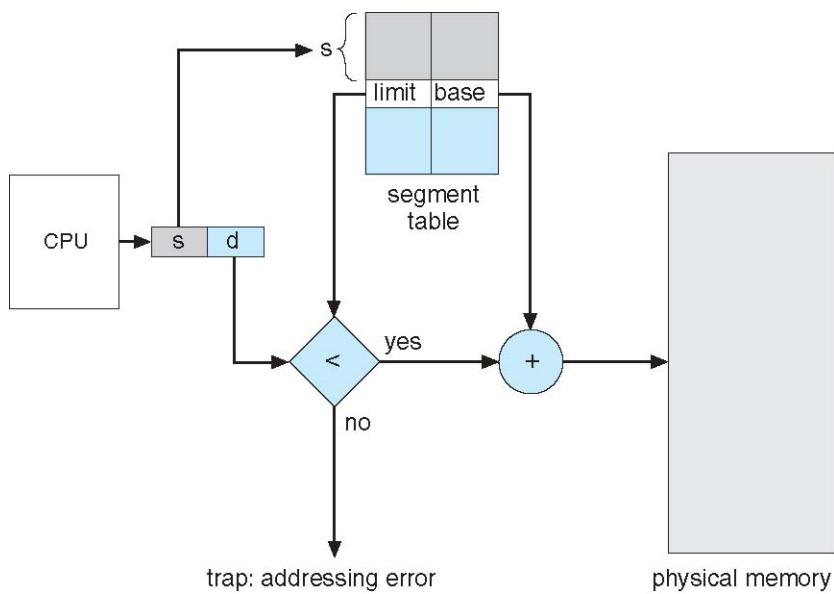


Figura 7.7: Traducción de segmentos

Esta traducción se realiza por hardware en la MMU, lo cual toma aproximadamente un ciclo del reloj. La MMU mantiene la tabla del proceso en ejecución solamente, ya que sería costoso (para el hardware) mantener todas las tablas.

Al existir un cambio de contexto la tabla de segmentos se almacena en el descriptor del proceso y se debe cambiar la tabla del proceso saliente por la del proceso entrante.

7.2.1.1. Administración de segmentos

Se deben crear tres segmentos al crear un proceso (`fork`), al hacer `exec` o al hacer `sbrk`, importante recordar que el segmento de sistema no se crea en la memoria principal, ya que dicho segmento es compartido entre todos los sistemas y corresponde el código y datos del sistema operativo. La destrucción de los segmentos se realiza cuando el proceso termina.

El núcleo mantiene un *heap* de memoria para segmentos, el cual es utilizado para los segmentos que se van creando y destruyendo. Esta se administra de forma similar a como se maneja `malloc` y `free`.

Por ejemplo para la administración de los segmentos se puede utilizar una lista enlazada de pedazos de segmentos disponibles. Esta lista representa trozos de memoria contigua, en la cual cuando se crea un segmento se debe buscar cual es el “mejor” pedazo para ubicar el segmento del proceso. Cuando se destruye un segmento se debe devolver a esta lista el espacio liberado, y de ser necesario unirlo a un segmento contiguo que ya estuviese libre.

El mayor problema es el elegir que segmento entregar, ya que se debe considerar el tamaño del mismo. Históricamente existen 3 estrategias:

- **First-fit o primer ajuste:** recorrer secuencialmente la lista desde el inicio hasta encontrar un espacio que alcance para el segmento, se divide el segmento usando el tamaño preciso y devolviendo lo que no se ocupará a la lista enlazada. Esto achica trozos disponibles dejando cada vez tamaños más pequeños libres.
- **Best-fit o mejor ajuste:** busca en toda la lista cual es el tamaño más cercano suficiente para el segmento. El problema es que debe recorrer toda la lista y va dejando pedazos demasiado pequeños cada vez.
- **Worse-fit o peor ajuste:** busca en toda la lista y entrega el peor caso, es teórico y no se usa.

De los anteriores el más eficiente es *first-fit*, ya que *best-fit* deja muchos segmentos pequeños, tan pequeños que no sirven para ningún segmento. Adicionalmente se debe recorrer la lista completa cada vez, lo cual no es bueno.

Una mejora a *first-fit*, quedando como el mejor mecanismo para la asignación, es el uso de una lista circular. De esta forma no se parte cada vez del inicio de la lista, sino que se parte desde donde se dejó la última vez . Esto evita que los pedazos pequeños queden todos al inicio y la búsqueda sea más rápida, distribuyendo uniformemente los pedazos dentro de la lista.

Las técnicas anteriores producen el problema de **fragmentación externa**, los métodos anteriores producen pedazos de memoria pequeñas, que no pueden ser utilizados, sin embargo

la suma de estos pedazos pueden servir para atender a un segmento. Ejemplo, 5 trozos de 1K, se necesita un segmento de 3K, con los 5 trozos no se puede atender al segmento, pero si se pudieran unir si se podría.

Como solución al problema de fragmentación externa existe la **compactación**, la cual une los segmentos libres cambiando las direcciones reales, se deben corregir las tablas de segmentos de los procesos afectados. Las direcciones virtuales no cambian, por lo cual el proceso de compactación es transparente para los procesos. El problema de esta solución, es que se introduce una pausa al momento de ser realizada la compactación por el tiempo requerido para realizar la copia desde un lado de la memoria principal a otro, aún así esto es mejor que no poder entregar memoria.

Otra alternativa a la falta de memoria, es el cambio de memoria entre RAM y disco, mediante el uso de memoria virtual o *swap* lo cual será visto más adelante.

7.2.1.2. Potencial de la segmentación

Si bien segmentación no es lo que actualmente se utiliza, tiene la ventaja de ser más sencillo y fácil de implementar.

I. Incremento del espacio para *heap*.

Supongamos tenemos un área de datos en un proceso, donde se encuentra su heap. malloc pide memoria pero no hay, por lo cual es llamado sbrk para aumentar el tamaño del segmento de datos.

Para implementar la llama a sistema sbrk el núcleo debe:

- a) Solicitar un segmento más grande.
- b) Copiar el contenido.
- c) Liberar el antiguo segmento.
- d) Actualizar la tabla de segmentos.

Lo que el proceso ve es lo “mismo”, solo cambia el límite virtual del segmento, lo cual es natural ya que el proceso está pidiendo memoria. Sin embargo los cambios de direcciones reales del segmento, lo cual si cambio, no es visible por el proceso.

Se puede utilizar una optimización que no copie, realloc permite hacer esto extendiendo el segmento a un trozo contiguo libre. De hecho malloc automáticamente utilizará sbrk o realloc dependiendo de la situación, sin embargo si no hay un segmento contiguo libre necesariamente se debe copiar hacia otro lado ya que en segmentación cada segmento se debe encontrar de forma contigua en la memoria física.

II. Desborde de la pila

Esto ocurre automáticamente al ir llamando funciones (por ejemplo de forma recursiva). Para evitar que se produzca un *overflow*, el núcleo pide más memoria de forma transparente para el proceso más memoria. Esto se hará adivinando cuando se podría requerir más memoria para la pila, y el núcleo interrumpirá al proceso (sin que este se entere) y la aumentará.

En caso que ocurriese un desborde de la pila y se tratase de acceder a un área que no ha sido asignada ocurrirá un *segmentation fault*. Donde si el proceso no atrapa la señal del *segmentation fault*, el núcleo enviará una señal al proceso padre indicando el error y será el padre (la shell) quien imprimirá el mensaje.

III. Implementación de fork

```
int pid = fork();  
if (pid==0) {  
    // hijo  
} else {  
    // padre  
}
```

Como el segmento de código es de solo lectura y común para todos los hijos, se puede mapear para todos los hijos el mismo trozo físico en la memoria real para dicho segmento. Esto evita tener que copiar el código y ahorra memoria principal.

Si un proceso hace fork y hay una variable global que es modificada por el hijo, el padre no verá este cambio, ya que los procesos no comparten memoria o sea son procesos pesados. Cada proceso tendrá su propio segmento de datos.

IV. Swapping

Cuando la memoria escasea el scheduler de mediano plazo lleva procesos completos a disco. El estado del proceso cambia a SWAPPED y se guardan copias al bit de los segmentos que se envían a disco. En algún momento se llevarán a disco otros procesos y se restaurará este, lo más probable en otra dirección física de memoria, esto sigue siendo transparente para el proceso.

Esto es crítico en procesos interactivos. Sigue siendo mejor a no tener memoria y no poder atender a nuevos o tener que matar procesos. Se trata de evitar llevar procesos interactivos a disco.

7.2.2. Paginación

La paginación corresponde a otro método de asignación de memoria física no contigua, el cual es el actualmente utilizado. Básicamente es muy similar a la segmentación, sin embargo aquí la memoria no se divide por segmentos si no que se divide en páginas.

Específicamente es el proceso el que se divide en páginas de igual tamaño cada una de ellas y la memoria principal se divide en marcos de igual tamaño que las páginas. De esta forma cuando se crea un proceso y se copia el código más los datos estos son llevados a marcos en la memoria principal los cuales no necesariamente estarán contiguos. Al igual que en segmentación, desde el punto de vista de direcciones lógicas las páginas están contiguas.

La paginación no presenta el problema de fragmentación externa, por lo cual no requiere compactación. Sin embargo el problema que aparece es el de la **fragmentación interna**, ya que los marcos al ser de tamaño fijo puede ocurrir que una página de un proceso no complete el marco quedando espacio libre dentro de dicho marco. Este espacio puede ser utilizado para crecer eventualmente, pero de no ser utilizado será desperdiciado.

En la figura 7.8 se puede observar como tres procesos han sido divididos en páginas del mismo tamaño, la memoria principal ha sido dividida en marcos del mismo tamaño que las páginas. Cada página se ubica en un marco dentro de la memoria principal. Es la tabla de páginas la que dice que página se encuentra en que marco.

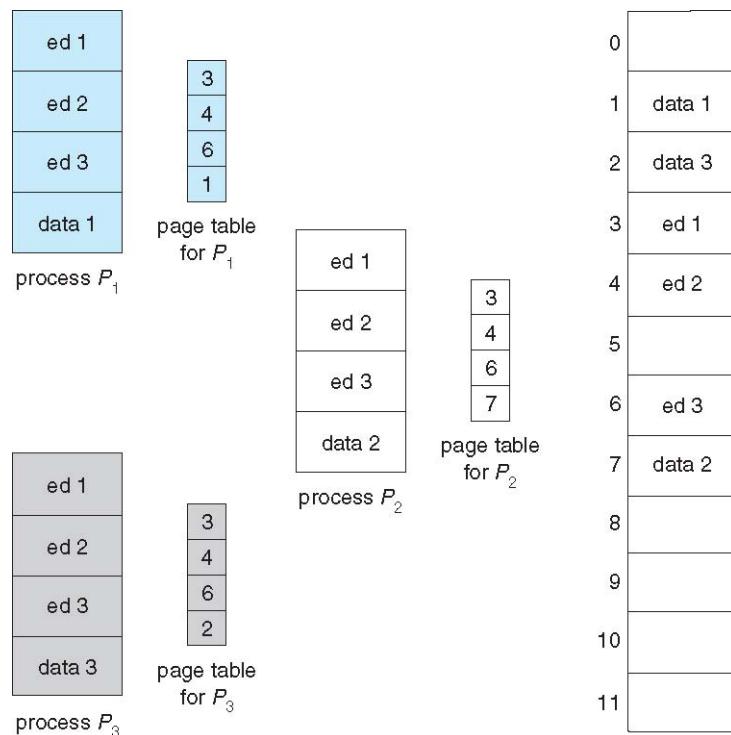


Figura 7.8: Ejemplo de paginación con 3 procesos

7.2.2.1. Carga dinámica

El hecho de tener ahora páginas permitirá llevar a la memoria principal partes del código o datos del programa, no teniendo que llevar todo el código (como ocurría con segmentación) de esta forma se cargarán bajo demanda las páginas del proceso y si existen algunas que nunca se lleguen a utilizar entonces nunca serán cargadas en la memoria principal.

Esto lo que busca es reducir el espacio, específicamente marcos, utilizados por un proceso en la memoria principal, cargando solo lo que es necesario.

Para utilizar la carga dinámica el sistema operativo deberá vaciar la memoria cache L1 (de nivel 1) para que se produzcan los fallos de página del proceso que está en ejecución cada vez que ocurra un cambio de contexto. La acción de vaciar la caché es la que finalmente representa gran parte del costo del cambio de contexto.

7.2.2.2. Enlace de bibliotecas

Cuando un programa requiere hacer uso de alguna biblioteca que esta en el sistema existen dos alternativas para el enlace de las mismas, un enlace estático y otro dinámico.

En el **enlace estático** las bibliotecas son agregadas al ejecutable al momento de la compilación. Esto implica un mayor tamaño en el programa resultante y eventualmente una mayor cantidad de páginas que cargar en memoria cuando el programa se ejecute. Esto tiene además el problema que se repetirán páginas de la biblioteca entre varios procesos que la usen y la tengan enlazada de forma estática en su código.

En el **enlace dinámico** al momento de compilar solo se deja una referencia a las bibliotecas, las cuales deberán ser cargadas por el sistema al momento de la ejecución del programa pero en un espacio diferente al código del programa. De esta forma se podrán compartir varias bibliotecas entre varios procesos en ejecución, en el fondo se compartirán las páginas cargadas correspondientes a dichas bibliotecas. Si la página requerida no esta cargada cuando la solicita un proceso P1 se cargará en un marco M1, cuando un proceso P2 requiera la misma página de la biblioteca no se volverá a cargar si no que se hará referencia al marco M1 ya

cargado. Este espacio para cargar bibliotecas compartidos puede apreciarse en la figura 7.9.

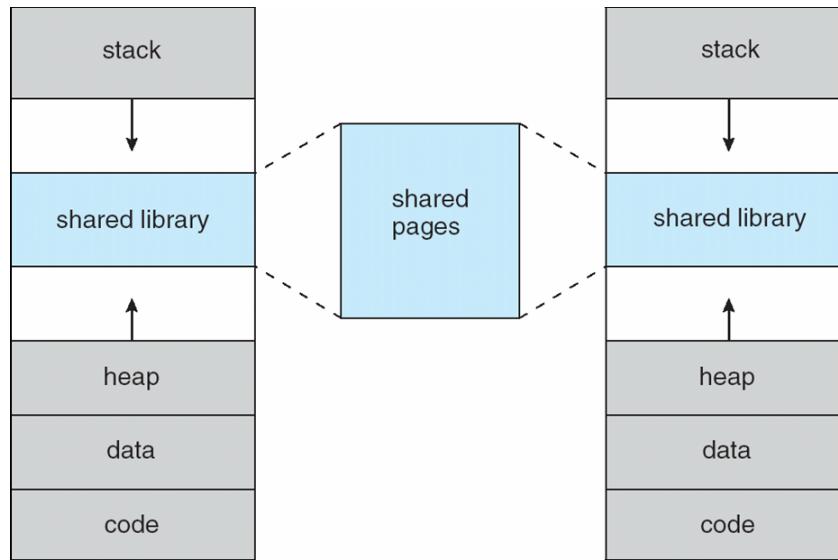


Figura 7.9: Biblioteca compartida entre dos procesos

Uno de los problemas con el enlace de bibliotecas de forma dinámica es el uso de variables globales en las bibliotecas. Por lo cual si se esta desarrollando un software con múltiples hebras no se deberán utilizar bibliotecas que no se han desarrollado para ser utilizadas en procesos ligeros, ya que pueden aparecer problemas de *dataraces*.

7.2.2.3. Procesos semi-pesados

Adicionalmente a la idea de bibliotecas compartidas, se pueden tener áreas de datos compartidas entre procesos. Hasta ahora se habían visto procesos livianos y procesos pesados. Un proceso semi-pesado corresponde a un proceso que puede compartir ciertas áreas de memoria con otro proceso.

Supongamos un proceso pesado con Código (C), Datos (D) y Pila (P). Un proceso semi-pesado podría corresponder al proceso pesado anterior (C, D y P) más un área para datos (D') que es compartida con otros procesos.

Existen funciones que permiten retornar puntos a espacios de memoria compartida, como por ejemplo *mmap*.

7.2.2.4. Implementación tabla de páginas

Las tablas de páginas son considerablemente más grandes comparadas con las tablas de segmentos, esto ya que un programa al ser dividido en páginas por lo general contendrá más páginas que si fuese dividido en segmentos. Por lo anterior las tablas de páginas no pueden ser almacenadas completamente en la MMU, y deben ser mantenidas en la memoria principal.

La situación explicada anteriormente implica que por cada acceso “útil” a la memoria principal se requiere un acceso para buscar la tabla de páginas (para saber donde está lo solicitado) y otro para acceder a lo solicitado. Este doble acceso para llegar a la memoria principal significa tiempos mayores de acceso a los datos requeridos.

Como solución al problema de doble acceso se utiliza *TLB* (*translation lookaside buffer*), lo cual es un *buffer* para la tabla de páginas. Por lo cual cuando se hacen consultas primero se revisa en la *TLB* por la página solicitada, y solo si no está ahí, se va a la memoria principal por ella.

7.2.2.5. Traducción de páginas

La traducción de páginas corresponde a un proceso similar al de la traducción de segmentos, de esta también se encarga la MMU. La dirección lógica en este caso entregará el número de página del proceso y el desplazamiento dentro de la página. Por ejemplo en un sistema con direcciones de 32 bits se podrían utilizar 20 bits para la página (y marco) y 12 bits para el desplazamiento.

En la figura 7.10 se observa el proceso de traducción, donde una vez obtenida la página y el desplazamiento se busca en la tabla de páginas el marco correspondiente a dicha página. En caso que la página no estuviese cargada en un marco en la memoria ocurrirá un fallo de página y la página deberá ser traída desde el disco, si la página no existe o el desplazamiento

esta fuera de rango se produce un *segmentation fault*. Una vez se tiene el marco y se sabe que el desplazamiento es válido se va a la memoria principal con la dirección física.

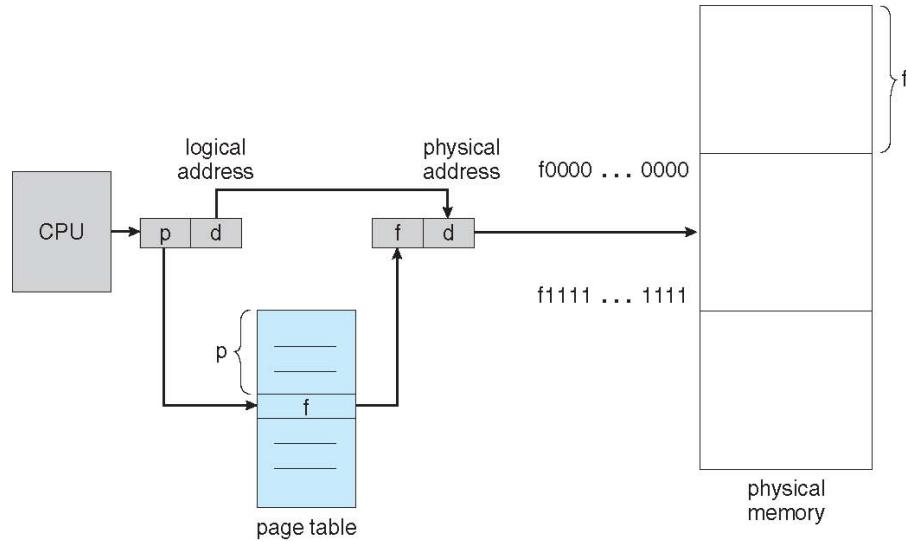


Figura 7.10: Traducción de páginas

7.2.2.6. *Copy on write*

El objetivo de *copy on write* es hacer un *fork* más eficiente, donde las páginas serán copiadas en la memoria principal si y solo si son modificadas. Mientras no sean modificadas las páginas seguirán estando compartidas entre el padre y el hijo.

En la implementación cuando se realiza *fork* de un proceso las páginas no son copiadas si no que son marcadas como solo lectura y además se indica que cuando se quieran escribir primero se deberán copiar. Para esto se utilizarán campos de los atributos de las páginas para indicar que se debe copiar la página cuando uno de los procesos (padre o hijo) la deseé modificar.

Cada vez que se quiera hacer una modificación se producirá una interrupción que verificará si la página puede ser escrita, si no lo es se verificará si existe un bit que indique que se debe copiar, si existe se copiara y luego se modificará. Si muchos procesos quisieran modificar

muchas páginas se producirán muchas interrupciones, para evitar esto se puede utilizar una heurística que determine que si se han copiado X páginas a la siguiente interrupción se copien todas las páginas del proceso a un nuevo espacio, asumiendo que seguirán ocurriendo interrupciones.

Este método también es conocido como duplicación perezosa de las páginas o copia bajo demanda.

7.2.2.7. Tablas de páginas de dos niveles

Estos ejemplos son para x86 (32 bits), para procesadores de 64 bits existen tablas de páginas de 3 niveles asociadas con procesadores de 64 bits.

Para el caso de 32 bits, la dirección de memoria virtual se dividirá en 2 partes. Donde la primera parte a su vez se divide en un bloque b (10 bits) y página p (10 bits). La segunda parte es el desplazamiento d (12 bits).

En 32 bits el tamaño máximo de memoria que se puede direccionar son 4G, en este caso la memoria es dividida en 1024 bloques, entonces cada bloque es de 4M. Y cada bloque es dividido en 1024 páginas, entonces cada página es de 4K.

Existe un directorio de bloques que indica para cada bloque:

- **b**: el bloque.
- **t**: utilizado para entregar la dirección de una tabla de páginas asociada al bloque b (dirección de 10 bits).
- **Atributos**: similares a los de las páginas, pero valido para todo el bloque (o sea para todas las páginas).

Con lo anterior se construye una nueva dirección, formada por $t + p$, que entrega un puntero directo a la entrada en la tabla de página t del bloque b . Con este puntero a la entrada y p se puede construir la dirección real de la página, ya que la entrada de la tabla de

páginas (apuntada por t) contiene el marco m al cual se le suma d , obteniendo la dirección física.

La idea de usar estas dos tablas (por eso se llama tabla de páginas en dos niveles), es que la mayoría de los procesos requieren menos de 4M. Donde un proceso ocupará 1 bloque para su código y datos y otro bloque para la pila. Todas los otros bloques se encontrarán inválidos (no asignados). El espacio que queda en el bloque es dejado para crecimiento futuro.

El sobre-costo para un típico proceso (menor a 4M) es:

- Una tabla para código más datos.
- Una tabla para la pila.
- Un directorio para manejar los bloques.

Tanto el directorio como la tabla de páginas requieren (cada una) 4K, ya que cada entrada ocupa 32 bits. Por lo cual un proceso menor a 4M requerirá 12K para manejar su memoria principal. Sobre-costo es muy bajo, menor al 1% (lo que se requiere *versus* lo que se podrá direccionar).

Cuando se requiere más memoria, llamando a *sbrk*, se entregarán páginas dentro del mismo bloque de memoria ya asignado, una vez el bloque se llena se van pidiendo nuevos bloques de memoria.

Cuando los datos llegan a la dirección de la pila, significa que no hay más espacio que asignar y *malloc* retornará *NULL* y el proceso no podrá continuar, asumiendo que terminará cuando *malloc* retorne *NULL*.

7.2.2.8. Tablas de páginas de tres niveles

Los directorios se referencian de a 4G (o sea se agregan 10 bits más a la dirección virtual), donde existe otra tabla (tercer nivel) que administra estos directorios.

En sistemas de 64 bits, se dispone de 64 bits para la dirección, sin embargo direcciones de 64 bits permitirían referenciar mucha más memoria de la que actualmente se encuentra

disponible. Por lo anterior, de los 64 bits solo se usan 10 bits más, o sea 42 bits de los 64. Arquitecturas futuras deberán considerar utilizar más de 42 bits para poder referenciar más memoria.

El problema con este esquema, es que se requieren 3 accesos a memoria para llegar a la memoria física requerida. Para solucionar esto aquí entra la TLB que almacena la tabla de páginas y el directorio.

Hoy en día los sistemas operativos modernos asignan bloques completos de memoria cuando un proceso lo solicita, no páginas. O sea, ya sea que requiera 10K o 3M se le asignarán 4M. La asignación de páginas de forma individual esta reservada para actividades especiales, como las del núcleo.

7.3. Memoria virtual

La memoria virtual utiliza espacio de la memoria secundaria (disco) para almacenar temporalmente los procesos que por algún motivo han sido enviados fuera de la memoria principal por el planificador de mediano plazo.

El esquema de memoria virtual es utilizado con las páginas, no se llevan bloques completos a disco solo páginas de dichos bloques. Se podría utilizar este esquema en segmentación, o en bloques, pero sería muy costoso porque el acceso a disco es lento. Se hablará de memoria virtual con páginas por ser el esquema utilizado en los sistemas operativos modernos.

A la tabla de páginas se le agregará un **bit de validez** (o invalidez, dependiendo de la implementación), que indicará si la página está o no en memoria principal. Cuando la página solicitada no se encuentra en un marco se produce un **fallo de página** (o *page fault*), indicado por su bit de invalidez en la tabla de páginas, el sistema operativo captura esto y si determina que la página esta en el disco procede a llevarla a un marco libre. Una vez la página ha sido copiada a un marco en la memoria principal, la página es marcada como válida en la tabla de páginas y se reinicia el proceso desde el punto donde requería acceder

a la página que se cargó. Este proceso puede ser apreciado en la figura 7.11 y es importante mencionar que es totalmente transparente para el proceso.

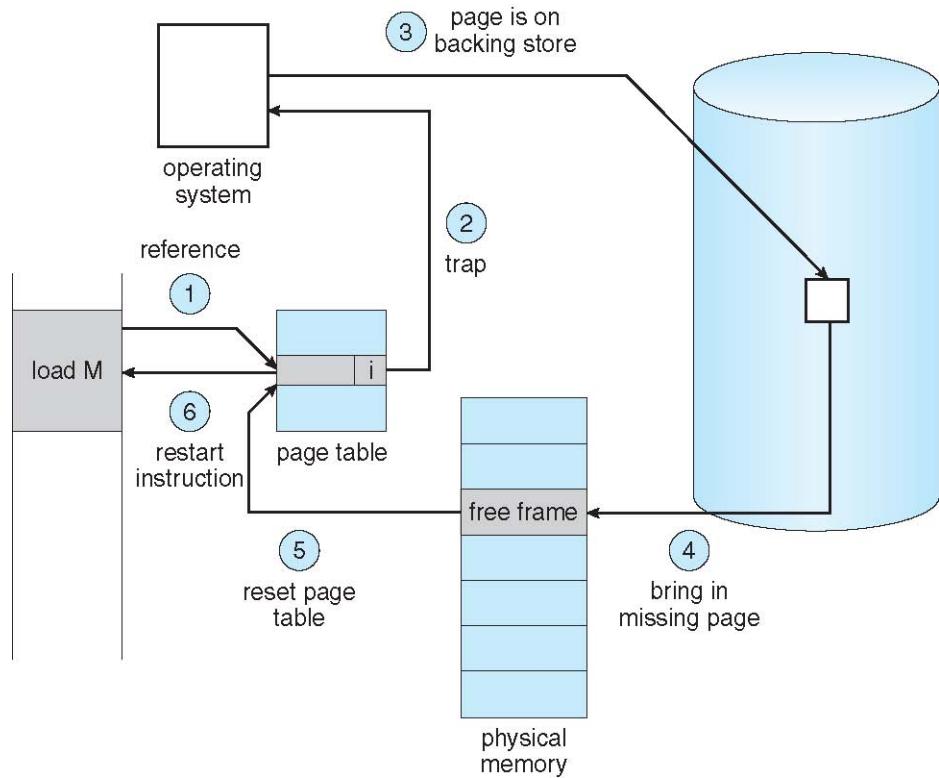


Figura 7.11: Proceso que ocurre al existir un fallo de página

El proceso de **intercambio** ocurre cuando es detectada una página inválida y está se encuentra en el disco, pero no hay marco libre en la memoria principal para ubicarla. Es en este punto donde ocurre lo descrito en la figura 7.12, donde primero se debe llevar a la “víctima” desde el marco en la memoria principal al disco, marcar su página como inválida, luego llevar la página requerida del disco al marco liberado y marcar esta última página como ahora válida. Luego se continúa la ejecución con la página cargada en un marco.

Un uso lógico de la memoria virtual es cuando el sistema no dispone de más memoria principal para la creación de nuevos procesos. Adicionalmente el uso de memoria virtual permitirá disponer en el sistema de un espacio de direccionamiento virtual mayor al real para

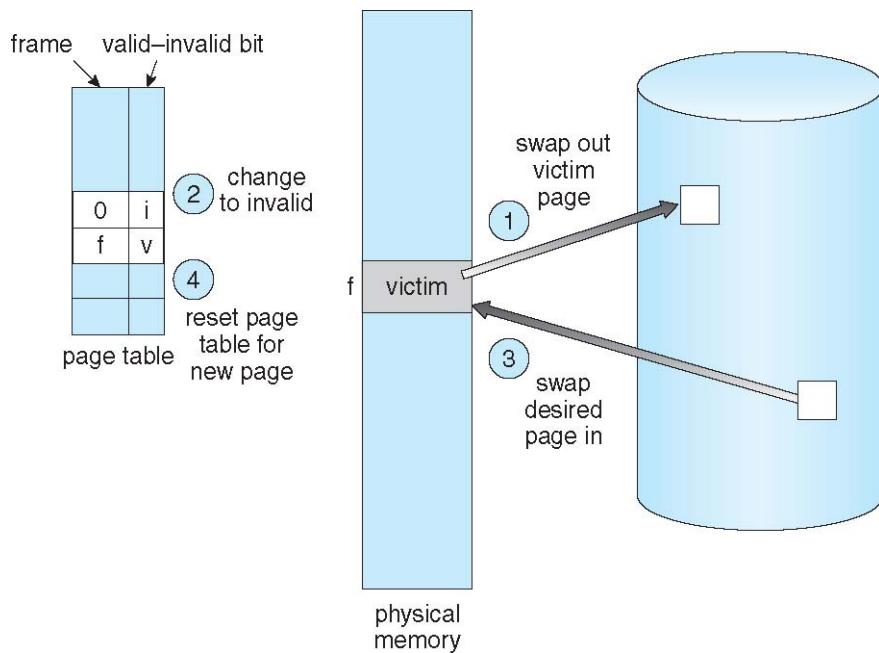


Figura 7.12: Proceso de intercambio de páginas

los procesos, ya que a pesar de no tener marcos para todas las páginas del proceso este podría tener aquellas páginas que no este utilizando en la memoria virtual y solo traerlas a principal cuando las necesite usar.

7.3.1. Algoritmos de reemplazo de páginas

Para el proceso de intercambio lo crucial es decidir a quién se sacará de la memoria principal para ser llevado a disco. En estricto rigor se debe determinar la página que será sacada desde la memoria principal. Para esto existen diversos algoritmos los cuales serán mencionados a continuación, la idea es encontrar un algoritmo que en el sistema genere la menor cantidad de fallos de páginas. Lo último porque el proceso de intercambio es lento ya que implica acceso al almacenamiento secundario.

En cualquiera de los algoritmos se puede utilizar una optimización que implica el uso de un **bit de modificación** (o **bit dirty**) en la tabla de páginas, este indicará si la página

(que ya está en disco) fue modificada al estar en la memoria principal (en un marco). Si fue modificada se copia a la memoria virtual al existir el intercambio, sin embargo si no fue modificada no se copia, con esto se reducen los tiempos que dura el fallo de página.

El objetivo de estos algoritmos es ser capaz de elegir una página que será usada en el futuro más lejano, sin embargo determinar esto es imposible ya que no se sabe que pasará en el futuro. Sin embargo se puede usar la historia pasada para poder determinar que páginas podrían usarse en el futuro más lejano. De esta forma se asumirá, por ejemplo, que una página que no se ha utilizado en el último minuto no se usará hasta al menos dentro de otro minuto más.

El **tiempo de acceso efectivo** corresponde al tiempo real que tomará acceder a la memoria principal si existe un fallo de página. Se define como $T_{ef} = (1 - r) * t_M + r * t_p$, donde:

- t_M es el tiempo de acceso a memoria física, por ejemplo 60 ms.
- t_p es el tiempo de reemplazo de la página, por ejemplo 12 ms.
- r es la tasa de fallos de página.

7.3.1.1. Ideal: “oráculo”

La estrategia ideal sería consultar el “oráculo” y se lleva a disco la página que permanecerá más tiempo sin ser usada. Lamentablemente esto no se puede implementar, ya que no se puede ver el futuro. Sin embargo sirve como referencia.

Ejemplo:

- Marcos: 3.
- Páginas: 8.
- Cadena de referencia: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

¿Cuántos fallos de página generará la cadena de referencia indicada?

7.3.1.2. FIFO

En este caso se van ubicando las páginas en la memoria en el mismo orden que son referenciadas, la más “antigua” será la que se elegirá como “víctima” para ser sacada en caso de requerir un marco libre. Esta estrategia puede ser considerada como la peor, sin embargo será la más simple de implementar.

Ejemplo:

- Marcos: 3.
- Páginas: 8.
- Cadena de referencia: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Luego de la ejecución total del proceso, o sea, cuando se haya procesado toda la cadena de referencia de páginas habrán ocurrido un total de 15 fallos de página, ver figura 7.13. Sin embargo la cantidad de fallos de página dependerá de las solicitudes de páginas, esto significa que con otra cadena de referencia de memoria podrían haber más o menos fallos de página.

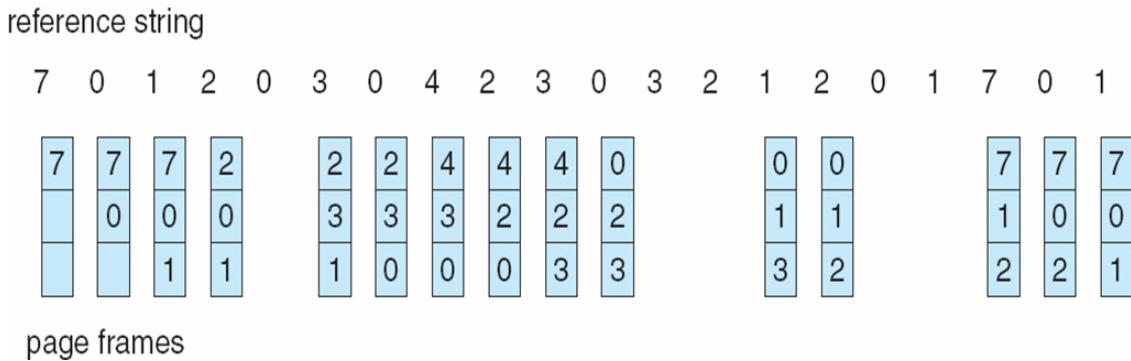


Figura 7.13: Ejemplo de intercambio con algoritmo FIFO

7.3.1.3. LRU

En este caso la página que es reemplazada es aquella que no ha sido usada en el último tiempo (*Least Recently Used*), o dicho de otra forma aquella que fue usada hace más tiempo.

Para esto se debe llevar por cada página una asociación con el tiempo de la última vez que la página fue referenciada.

Implementación:

- a) **Con un contador:** cada entrada de página en la tabla de páginas tiene un contador, cada vez que la página es referenciada el reloj es copiado al contador. Cuando una página necesita ser cambiada se mira el contador y se busca el valor más pequeño. Este método es sencillo de implementar pero tiene el problema de requerir buscar en toda la tabla de páginas para encontrar la usada hace mas tiempo, por lo cual resulta muy ineficiente y cara de implementar. En la práctica no es utilizada.
- b) **Con una pila:** se mantiene una pila con los números de página del sistema, cada vez que una página es referenciada se mueve al tope de la pila. Por lo cual cuando se requiere una página para intercambiar se saca aquella que esta más abajo en la pila.

7.3.1.4. Second chance o estrategia del reloj

Esta es una aproximación de LRU donde se elige una página que lleve bastante tiempo sin ser utilizada. Se podría cometer un error y no sacar la que lleva más tiempo sin ser usada, pero esto es una aproximación que busca aumentar la velocidad.

El núcleo coloca el bit “r” o **bit de referencia** en 0. Si la página es referenciada el hardware del procesador (la MMU) coloca el bit “r” en 1 cuando la página se usa. El sobre costo en esto es muy bajo, ya que solo implica ir a la tabla de página y cambiar un bit. Este cambio se realiza en la TLB, por lo cual cada cierto tiempo se deberá actualizar el bit en la memoria principal, pero no es común tener que actualizar de la TLB a la memoria principal.

La página se puede llevar a disco cuando “r” permanece en 0 por “suficiente” tiempo. ¿Cómo se considera el tiempo? Para esto se utiliza una lista circular donde cada vez que se quiere sacar una página de la memoria principal se consulta el bit “r”, si es 1 la página es dejada en memoria principal (dando una segunda oportunidad) y su bit es puesto a 0, se

continúa así hasta encontrar una página con su bit de referencia en 0. Se marca el bit de referencia en 1 cada vez que la página es referenciada y se parte revisando a contar de la página que se cambio la vez anterior.

En la figura 7.14 se puede apreciar el funcionamiento del algoritmo, al lado izquierdo desde donde se inicia la búsqueda y al lado derecho el estado final de los bits de referencia y la “victima” que fue elegida.

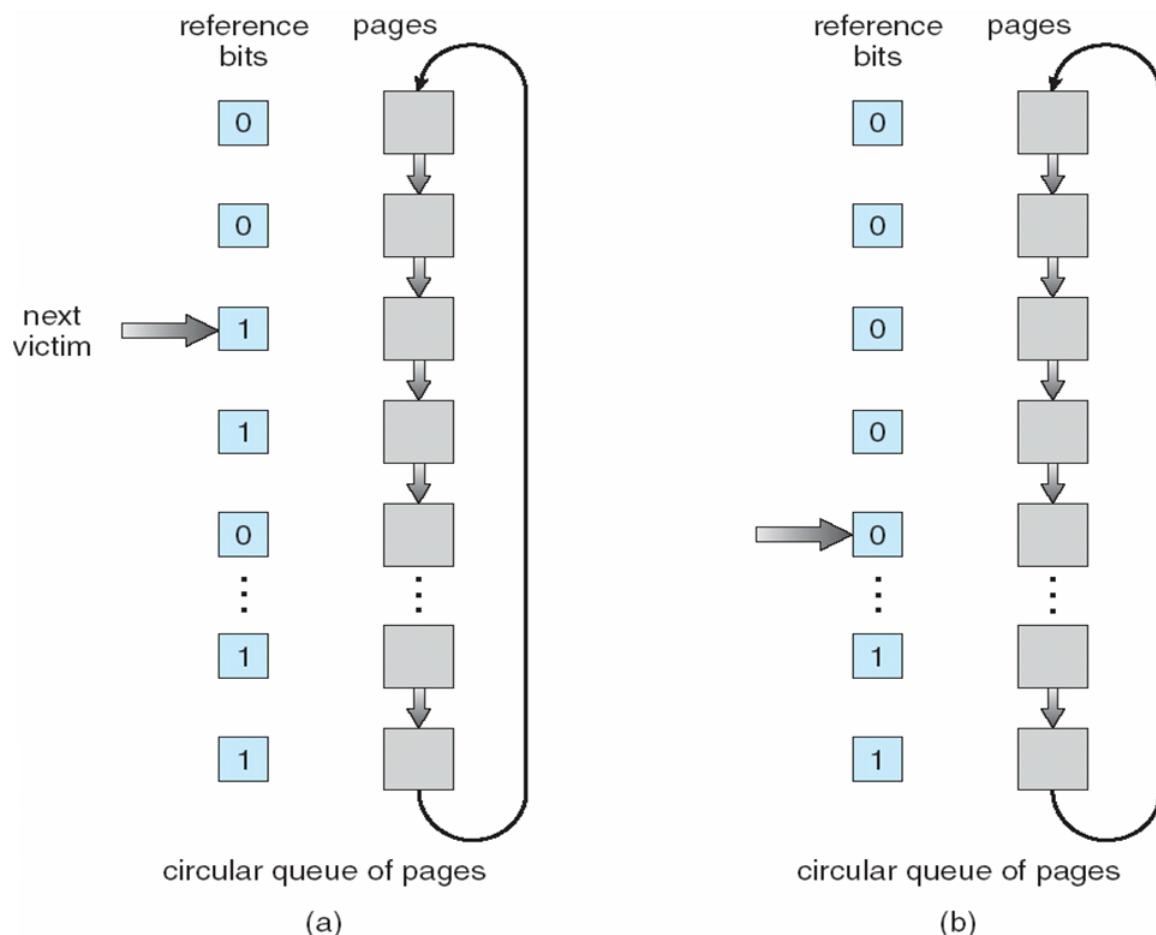


Figura 7.14: Ejemplo de intercambio con algoritmo Second Chance

Esta estrategia es utilizada hoy en día, su seudo código se muestra a continuación:

```
while (bitR(cursor())==1) {
```

```

bitR(cursor()) = 0
avanzarCursor()
}
reemplazarCursor()
AvanzarCursor

```

Si todas las páginas tienen su bit de referencia r en 1 se deberá dar la vuelta completa poniendo los bits en 0 hasta volver a la primera página revisada la cual ahora tendrá el bit en 0 y se escogerá. Esto es una situación anómala ya que en estricto rigor no se está determinando lo esperado, ya que todas estaban en iguales condiciones y se terminó sacando la primera consultada. Se da cuando el tiempo en que se demora en dar la vuelta a la lista es muy grande, sin embargo en la práctica es muy poco probable que ocurra y no es considerado como un problema real.

Un problema real es el **thrashing**, que corresponde a la situación cuando la tasa de fallos de páginas es muy alta. Esto ocurre cuando ocurren fallos de páginas muy seguidos, el cursor avanza muy rápido y los discos al ser lentos (tiempo de acceso alrededor de 10 ms) se podrían atender aproximadamente 100 fallos de página por segundo. En esta situación el sistema operativo trabajará llevando procesos de disco a RAM y de RAM a disco, pero no se hará trabajo útil. La CPU pasa el 99 % del tiempo ociosa y el disco de *paging* pasa el 100 % del tiempo ocupado, lo anterior significa que los procesos no avanzan. Recuperarse de esto es muy complicado de forma automática, por lo cual la solución es matar procesos, lo cual no se hace automáticamente. El problema de hacerlo manualmente es que se requiere memoria para conectarse al equipo, luego memoria para ver los procesos y matar alguno que ocupa mucha RAM. Caso extremo, la solución es reiniciar.

7.3.1.5. Estrategia del *working set*

La idea de este método es utilizar la combinación de paginamiento por demanda con *swapping*, basado en un *working set* por proceso evitando el *thrashing*.

El *working set* son las páginas que ha usado recientemente el proceso. La estrategia del reloj no hace una distinción por proceso, compara todas las páginas de los procesos juntas. En este caso el tiempo considerado para correr el reloj es solo cuando el proceso está avanzando o sea el tiempo virtual del proceso.

Más formalmente, $WS_P(t, \Delta t)$ es el conjunto de páginas usadas por P en el intervalo $(t - \Delta t, t)$ de tiempo virtual. Se debe mantener el WS de cada proceso actualizado en memoria, para esto se calcula el WS para cada proceso cada Δt unidades de tiempo virtual. Aquellas páginas que no pertenecen al WS se pueden llevar a disco.

La principal desventaja de este método es que siempre se debe estar calculando el *working set*, a pesar de que haya mucha memoria física disponible. En el caso de Linux, cuando la CPU no tiene “nada” que hacer, el núcleo se pone a actualizar tablas para la estrategia del reloj.

Para el cálculo del WS_P , que se realiza cada Δt segundos de uso de CPU (tiempo virtual), tenemos C como el conjunto de páginas candidatas para ir a disco. Con este C se define la forma de realizar el WS_P según el siguiente algoritmo:

```
WSp = vacío
para toda página q perteneciente a P residentes en memoria física {
    if(bitR(q)==1) {           // si el bit r es 1
        WSp = WSp U {q}       // se agrega la página q al WS
        bitR(q) = 0            // se pone el bit r en 0 (para quitarla a futuro)
    } else {
        C = C U {q}           // si el bit r era 0 se agrega al conjunto C
    }
}
```

Cuando se deba elegir una página para llevar a disco bastará recorrer el conjunto de páginas C y revisar su bit de referencia r (que podría haber sido modificado en el Δt).

Cuando ocurre un fallo de página se debe seleccionar una página para intercambiar, esto se puede realizar con el siguiente algoritmo:

```

while C no esté vacío { // mientras vayan quedando elementos en C
    Sea q ∈ C // por cada elemento q perteneciente a C
    C = C \ {q} // quitar q del conjunto C
    if(bitR(q)==0) { // si el bit de referencia es 0
        return q // se retorna q para reemplazar
    }
    WSp = WSp ∪ {q} // si el bit r estaba en 1 se coloca a q en el WS
}
Swap() // se cambia la página q por la requerida

```

Cuando la sumatoria de los *working set* de cada proceso, $\sum_{i=1}^n WS_{P_i}$, es mayor a la memoria física disponible se debe empezar a hacer intercambio de procesos completos. Un proceso no puede correr eficientemente si su *working set* es más grande que la memoria física del computador.

Para el caso del sobre costo en los fallos de páginas definamos:

- pf : fallos de página.
- ta : tiempo de acceso al disco
- tt : tiempo total que tomaron los fallos de página

Para determinar el tiempo total se utilizará: $pf * ta = tt$, si consideramos un proceso que causa 10 fallos de página en un segundo de uso de CPU, en este caso el sobre costo sería $10 * 10 = 100$. Estos 100 ms representan un 10% de sobre costo sobre el segundo de CPU utilizado.

7.4. Ejercicios y preguntas

1. ¿Cuál es la forma más simple de asignar memoria principal?.
2. Describa los conceptos: base, límite y desplazamiento, y como están relacionados.
3. ¿Qué es un *segmentation fault*?.
4. ¿Qué implica asignar la memoria de forma contigua?.
5. Explique los tres tiempos en que puede ser realizado el enlace de direcciones.
6. ¿Cuál es el inconveniente de utilizar intercambio con los enlaces en tiempo de compilación o en tiempo de carga?, explique.
7. ¿Cuál es la relación entre direcciones virtuales y físicas?.
8. ¿Las direcciones virtuales son contiguas?.
9. ¿Qué ventajas presenta el uso de direcciones virtuales?.
10. ¿Por qué no era posible instalar Unix en los ordenadores con procesadores 8088?.
11. Fundamente la siguiente premisa: la asignación no contigua de memoria física no implica asignación no contigua de direcciones virtuales.
12. ¿Cómo se divide el proceso al utilizar segmentación?.
13. ¿Cuántos segmentos como mínimo se deben asignar al inicio del proceso?.
14. ¿Cuál es la funcionalidad de la tabla de segmentos?.
15. Explique el proceso de traducción de dirección virtual a dirección física en un esquema con segmentación.

16. Cuando un proceso esta en estado inicial, ¿qué método de asignación de segmentos (First-Fit, Best-Fit o Worse-Fit) se utiliza para asignar el segmento de sistema?.
17. Explique en que consiste la fragmentación externa, como se soluciona.
18. ¿Cuál es el problema de la compactación?.
19. Cuando un segmento necesita crecer, explique las dos situaciones que pueden ocurrir.
20. Fundamente la siguiente premisa: el segmento de código puede ser compartido entre proceso hijos.
21. ¿Por qué el usar *copy on write* se considera una optimización en la creación de páginas al hacer **fork**?.
22. ¿Cuál es la principal diferencia entre segmentación y paginación?.
23. Los tamaños de las páginas y marcos, ¿son iguales o diferentes?, fundamente.
24. La cantidad de páginas y marcos, ¿son iguales o diferentes?, fundamente.
25. Explique el problema de fragmentación interna en las páginas.
26. Explique el concepto de carga dinámica de páginas.
27. ¿Qué ventaja tiene el enlace dinámico de bibliotecas?.
28. Explique el proceso de traducción de dirección virtual a dirección física en un esquema con paginación.
29. ¿Qué indica el bit de validez (o invalidez)?.
30. Explique en que consiste y como se maneja un fallo de página.
31. Explique cuando ocurre el proceso de intercambio.

32. Explique el proceso de intercambio.
33. Explique los tres algoritmos de reemplazo de páginas.

7.5. Referencias

- Sistemas Operativos, Segunda Edición, Andrew Tanenbaum, Capítulo 4.
- Sistemas Operativos, Quinta Edición, Abraham Silberschatz y Peter Baer Galvin, Capítulo 8 y 9.
- Sistemas Operativos, Segunda Edición, William Stallings, Capítulo 6 y 7.

Capítulo 8

Memoria secundaria

Durante este capítulo se abordaran temas generales relacionados con la memoria secundaria, específicamente relacionados con el sistema de archivos y el uso del mismo.

La memoria secundaria corresponde a aquella memoria de gran capacidad, bajo costo pero a la vez más lenta. Una característica importante versus otros medios de almacenamiento (como la RAM) es que corresponde a un medio persistente para guardar datos. Estos datos serán guardados con alguna estructura dentro del disco, la unidad mínima para trabajar con memoria secundaria, de una forma eficiente, es el archivo.

8.1. Archivo

Un **archivo** corresponderá a un conjunto de datos y/o instrucciones relacionadas que son guardadas en un dispositivo de almacenamiento secundario, como un disco duro o una memoria flash. Los clasificaremos en dos grupos:

- **Datos:** caracteres (ASCII) o binarios.
- **Programas:** generalmente binarios, podrían ser *scripts* también.

El sistema operativo será el encargado de mapear el archivo a un conjunto de direcciones física en el dispositivo, esto será muy similar al concepto de mapeo de un proceso a memoria principal visto en el capítulo 7.

8.1.1. Estructura

Un archivo deberá estar almacenado siguiendo algún formato o **estructura**, de esta forma podremos encontrar diferentes clasificaciones:

- **Sin estructura:** solo una secuencia de bytes.
- **Estructura simple:** un archivo de texto con diferentes líneas, que puede ser de largo fijo o variable.
- **Estructura compleja:** documentos con formatos (ejemplo: pdf).

Es posible simular la estructura simple o compleja utilizando la primera, esto definiendo caracteres de control que permiten definir el archivo. Será el proceso que crea el archivo el que deberá decidir que estructura tendrá.

Es importante mencionar que existen estructuras ya bien definidas y formatos de archivos estandarizados, por ejemplo las imágenes PNG ya tienen un formato definido, de la misma forma un gran número de archivos. En la figura 8.1 se pueden observar algunos de ellos, revisar /etc/mime.types para obtener una lista mayor de tipos de archivos.

8.1.2. Atributos

Adicionalmente al contenido del archivo, este podrá tener diferentes **atributos**, tales como:

- **Nombre:** información en formato “humano”.
- **Identificador:** etiquete (número) que lo identifica en el sistema de archivos.

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Figura 8.1: Diferentes tipos de archivos

- **Tipo:** necesario para sistemas que soportan diferentes tipos de archivos.
- **Ubicación:** puntero a la ubicación del archivo en el dispositivo.
- **Tamaño:** tamaño actual del archivo.
- **Protección:** controla quien puede leer, escribir o ejecutar el archivo.
- **Hora, fecha e identificación del usuario:** datos para protección, seguridad y monitoreo de su uso.

A continuación se adjunta una lista con diferentes atributos de archivos mediante la salida del comando `ls`.

```
delaf@goku:~/unab/iet110$ ls -lh
total 208K
drwxr-xr-x 3 delaf delaf 4,0K jun  4 04:00 apunte
drwxr-xr-x 3 delaf delaf 4,0K abr  9 21:01 diapos
-rw-r--r-- 1 delaf delaf  15K jun  2 15:51 iet110.ods
-rw-r--r-- 1 delaf delaf 166K nov 22  2011 IET110 Sistemas Operativos.pdf
drwxr-xr-x 5 delaf delaf 4,0K may 15 18:41 pruebas
drwxr-xr-x 5 delaf delaf 4,0K mar 25 02:38 so_juguete
drwxr-xr-x 2 delaf delaf 4,0K dic  1  2011 trabajos
-rw-r--r-- 1 delaf delaf  998 nov 22  2011 trabajo_v
```

8.1.3. Operaciones

Sobre un archivo se pueden definir diversas operaciones, tales como: crear, abrir, escribir, leer, reposicionar, eliminar, truncar o cerrar. `Open(File)` busca en la estructura de directorios la entrada `File` y mueve el contenido de la entrada a la memoria. De forma contraria `Close` moverá de memoria a disco. ¿Qué sucedería si tuviese un archivo de 100MB y solo 64MB de memoria principal?

8.1.3.1. Abrir

Para manejar un archivo abierto se requieren varias piezas de datos, estas en conjunto permitirán traer el contenido deseado del archivo a la memoria principal. Estas son:

- **Puntero de archivo:** apunta a la última ubicación donde se leyó o escribió el archivo, por cada uno de los procesos que tiene el archivo abierto.
- **Contador de apertura:** cuenta el número de veces que el archivo es abierto, permite remover los datos de la tabla de archivos una vez el último proceso cierra el archivo.
- **Ubicación en el disco del archivo:** información para acceder de forma física al archivo.
- **Permisos de acceso:** información respecto al modo de acceso por proceso al archivo.

Algunos sistemas bloquean el acceso a los archivos si ya han sido abiertos por otro proceso, esta es una forma de mediar con el acceso concurrente a un mismo archivo. Cuando un archivo es abierto por primera vez el archivo es bloqueado, luego cuando se quiere realizar una segunda apertura dos situaciones pueden ocurrir: se puede denegar directamente el acceso al archivo o bien se puede avisar que el archivo está abierto y que el usuario elija que hacer, por ejemplo abrirlo como solo lectura.

8.1.4. Métodos de acceso

Los métodos de acceso definen como se trabajara con el archivo, ya sea para operaciones de lectura o de escritura.

8.1.4.1. Acceso secuencial

En este tipo de acceso el archivo es abierto y leído de forma continua hasta que el archivo termina. Si se quiere leer algo nuevamente se debe volver al inicio y empezar nuevamente.

```
/* operaciones */
read next
write next
reset
```

En la figura 8.2 se puede apreciar como el acceso secuencial opera sobre un archivo. Donde desde una posición actual solo se pude leer/escribir lo siguiente o volver al inicio.

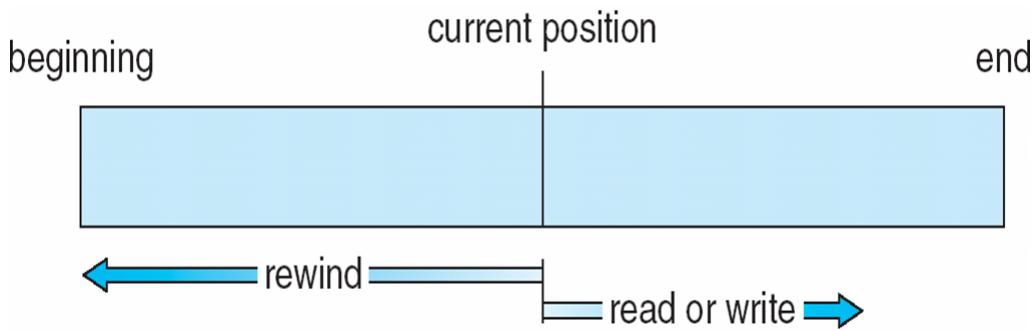


Figura 8.2: Acceso secuencial a un archivo

8.1.4.2. Acceso directo

En el acceso directo se puede controlar directamente la posición del cursor dentro del archivo, de esta forma se puede ir a una posición específica sin tener que recorrer (leer) todo el archivo desde el inicio.

```
/* operaciones */
read n
write n
position to n
```

Utilizando el acceso directo se puede simular el acceso secuencial como se ve en la figura 8.3.

sequential access	implementation for direct access
<i>reset</i>	$cp = 0;$
<i>read next</i>	<i>read cp;</i> $cp = cp + 1;$
<i>write next</i>	<i>write cp;</i> $cp = cp + 1;$

Figura 8.3: Acceso secuencial simulado con acceso directo

8.2. Estructura del disco

Un disco puede ser dividido en particiones. Eventualmente estos discos o particiones pueden ser protegidos ante fallas utilizando algún sistema, como RAID.

Las particiones, que son las que se formatean, también son conocidas como minidisks o slices (FreeBSD). En general la información de los discos es guardada en una tabla de particiones, la cual permite a lo más guardar la información de 4 particiones. Por esta razón un disco duro puede contener como máximo 4 particiones primarias o bien 3 primarias y 1 extendida, donde la extendida puede contener más particiones, pero lógicas.

Para utilizar el disco existen dos alternativas, escribir directamente en el disco secuencias de bytes, en este caso se habla de un disco tipo RAW (sin un sistema de archivo) o bien formateado con un sistema de archivos (como ext4, reiserfs, jfs, xfs, fat, ntfs). La entidad que contiene al sistema de archivos es conocida como un volumen, donde cada uno lleva un historial de la información del sistema de archivos en una tabla de contenido del volumen. Así como hay sistemas de archivos de propósito general hay aquellos que son sistemas de archivo de propósito especial o específico, frecuentemente relacionados a un sistema operativo

8.3. Estructura de directorios

Una estructura de directorios no es más que una colección de nodos conteniendo información acerca de todos los archivos que contienen. Tanto la estructura de directorio como los archivos residen en el disco, y ambas son necesarias para el correcto funcionamiento del sistema de archivos. Por lo anterior, un respaldo debe incluir las dos: archivos y estructura de directorios.

8.3.1. Operaciones

Sobre un directorio se pueden definir diversas operaciones, tales como: buscar un archivo, crear un archivo, borrar un archivo, listar el contenido de un directorio, renombrar un archivo o navegar por el sistema de archivos.

8.3.2. Organización de los directorios

El objetivo de dar una organización, orden o jerarquía a los directorios tiene relación con:

- **Eficiencia:** poder ubicar un archivo rápidamente.
- **Nombres:** dos o más usuarios podrían tener el mismo nombre para diferentes archivos o bien el mismo archivo tener diferentes nombres.
- **Agrupar:** agrupar los archivos por propiedades, usos o tipos (ej: binarios, configuraciones, datos variables, etc).

8.3.2.1. Nivel simple

En este caso solo existe un nivel de directorios, ver figura 8.4. Aquí los mayores problemas son los nombres de los archivos y la baja capacidad de agrupamiento.

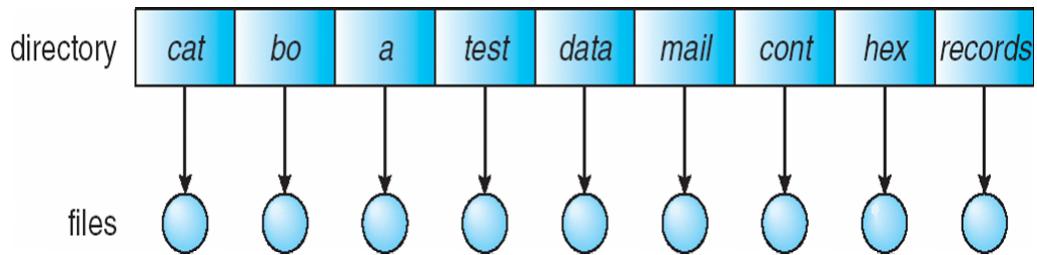


Figura 8.4: Jerarquía de directorios simple

8.3.2.2. Dos niveles

En este caso se definen directorios separados para cada usuario, ver figura 8.5. Esto permite tener el mismo nombre de archivo para diferentes usuarios y provee de una búsqueda eficiente. Sin embargo las capacidades de agrupar son muy limitadas.

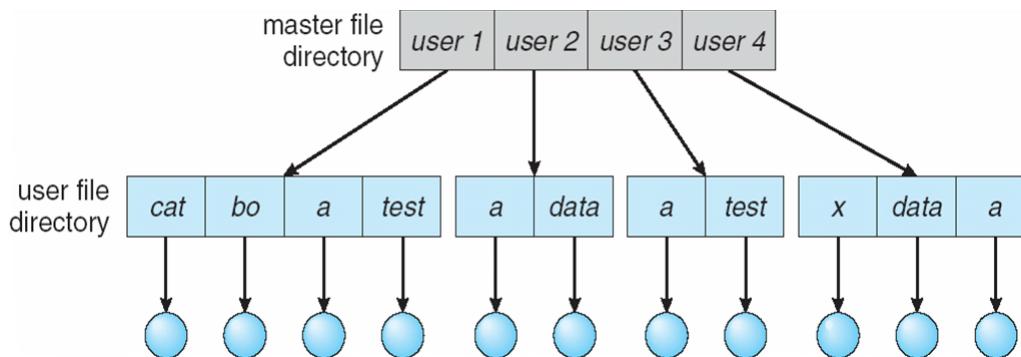


Figura 8.5: Jerarquía de directorios con dos niveles

8.3.2.3. Jerarquía en árbol

En este tipo se crean “infinitos” niveles de subdirectorios, tantos como sean necesarios, ver figura 8.6. Provee de una búsqueda eficiente, permite tener nombres de archivos repetidos y permite entregar capacidades de agrupamiento.

En este tipo de estructura se definen dos formas de navegar por el árbol de directorios, mediante **rutas absolutas** que parten desde la raíz del árbol o mediante **rutas relativas**

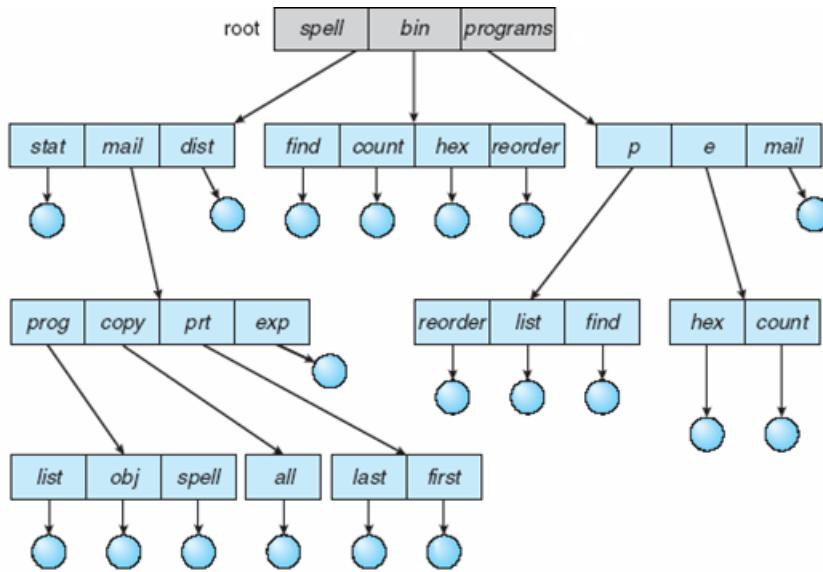


Figura 8.6: Jerarquía de directorios en árbol

que parten desde donde uno se encuentra “parado” en el árbol.

Respecto al borrado de archivos o directorios, esta debe ser realizada de forma recursiva, ya que se deben ir quitando las referencias entre directorios y archivos de tal forma de no dejar nodos o hojas sin un nodo padre.

8.4. Montaje

Un sistema de archivos debe ser montado antes de poder ser accedido. Este proceso se lleva acabo “tomando” el volumen no montado y “ubicándolo” en un punto de montaje. Este punto de montaje corresponderá a algún directorio dentro del sistema de archivos.

Si se llega a montar un volumen sobre un directorio que ya contiene archivos u otros directorios estos no se perderán, solo quedarán inaccesibles hasta que el volumen sea desmontado.

8.5. Compartición

Para compartir un archivo (o directorio) dentro del árbol de directorio se pueden utilizar *aliases* para nombrar a un mismo elemento en diferentes partes del árbol.

En la figura 8.7 se puede apreciar la compartición donde un mismo archivo se encuentra en dos carpetas `count` y de forma similar dos carpetas `w` y `words` apuntan a una misma llamada `list`.

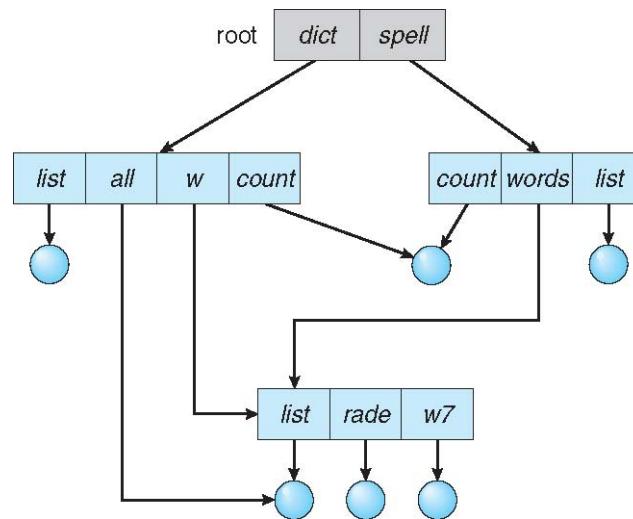


Figura 8.7: Enlaces

Lo anterior es conocido como un enlace, lo cual será otro nombre (un puntero) a un archivo ya existente. Cuando se quiera acceder al enlace se deberá resolver siguiendo el puntero para localizar el archivo. Se definen dos tipos de enlaces:

I. Enlaces simbólicos:

- Tamaño del enlace, tamaño de la ruta hacia el archivo.
- Apunta a una ruta en el sistema de archivos.
- Puede ser utilizado entre diferentes sistemas de archivos.

II. Enlaces duros

- Apunta al dispositivo de almacenamiento.
- Tamaño del enlace, tamaño del archivo.
- Solo puede ser utilizado en un mismo sistema de archivo.

8.5.1. Protección

Compartir archivos en un sistema multiusuarios es una tarea deseable, pero se debe considerar un esquema de protección para evitar que cualquier usuario pueda realizar cualquier tipo de operación sobre los archivos.

El propietario del archivo debe ser capaz de controlar ¿qué puede ser hecho? y ¿por quién puede ser hecho?. Para esto se utiliza el identificador del usuario (UID) y de grupo (GID) permitiendo definir los permisos que disponen cada uno de los usuarios del sistema sobre los archivos y directorios. Se definen tres tipos básicos de permisos: lectura (read), escritura (write) y ejecución (execute)

A continuación se explica la salida del comando `ls -l` que involucra los permisos recién descritos:

```
-rw-r--r-- 1 usuario grupo 289 nov 20 23:08 archivo
- - - - - - -
| | | | | | | | +----- fecha de modificación
| | | | | | +----- tamaño
| | | | | +----- enlaces duros
| | | | +----- permisos otros
| | +----- permisos grupo
| +----- permisos usuario
+----- tipo (común '{' o 'd')
```

8.6. Ejercicios y preguntas

1. Nombre 5 atributos de un archivo.
2. Explique las partes involucradas en la apertura de un archivo.
3. ¿Cuál es la diferencia entre el método de acceso secuencial y directo hacia un archivo?.
4. ¿Se puede simular el acceso secuencial a un archivo con acceso directo?, explique.
5. ¿Se puede simular el acceso directo a un archivo con acceso secuencial?, explique.
6. ¿Cuántas particiones primarias se pueden crear en un disco? ¿por qué?.
7. ¿Qué implica usar un disco como RAW?.
8. ¿Cuales son los objetivos de la organización del disco?.
9. ¿Por qué la estructura jerárquica simple no cumple con los objetivos?.
10. ¿Cuál es la diferencia entre rutas relativas y rutas absolutas?.
11. ¿Qué es el proceso de montaje?.
12. ¿Cuál es la diferencia entre enlaces simbólicos y duros?.
13. ¿Cuáles son los tres permisos definidos para un archivo o directorio?.

8.7. Referencias

- Sistemas Operativos, Segunda Edición, Andrew Tanenbaum, Capítulo 5.
- Sistemas Operativos, Quinta Edición, Abraham Silberschatz y Peter Baer Galvin, Capítulo 10 y 11.
- Sistemas Operativos, Segunda Edición, William Stallings, Capítulo 10 y 11.

Capítulo 9

Protección y seguridad

Controlar el acceso a los recursos es una de las tareas más importantes del sistema operativo, sin embargo no solo debe asegurar que estos accesos sean sincronizados y sin problemas de concurrencia (como ya se discutió anteriormente), sino que también debe asegurar que solo accedan a los recursos quienes estén autorizados a hacerlo.

En un modelo de protección el computador simplemente es un conjunto de objetos, ya sean estos físicos (hardware) o lógicos (software o datos). Donde cada objeto puede ser identificado por un nombre único y tendrá asociado un conjunto definido de operaciones.

Entonces, el problema de la protección consiste en asegurar que cada objeto sea accedido correctamente, mediante las operaciones definidas, y solo por aquellos procesos que tienen permitido hacerlo.

En este capítulo se hablará de conceptos de protección genéricos, los cuales pueden aplicarse a un sistema operativo o bien a cualquier otro tipo de sistema, ya sea o no computacional.

9.1. Principios de protección

En protección se considera generalmente el **principio del menor privilegio**, esto implica que un programa, usuario o sistema en general deberá tener los privilegios mínimos para poder

realizar las tareas que deba hacer.

Supongamos una secretaria en un banco, sería un error dar a ella acceso a la bóveda si de lo único que se tiene que encargar es atender las consultas de los clientes. Para realizar su tarea, “atender clientes”, la secretaria solo necesita acceso a una determinada área del banco, no a todo este.

Esto tiene mucha relación con la idea de **denegar todo y permitir algunos**, donde se evita que el usuario pueda hacer cualquier cosa en el sistema y solo se le conceden accesos específicos según lo que debe realizar.

Este tipo de política limita el daño si la ejecución del proceso tiene algún error, ya que de haberlo a lo más podrá afectar a aquellos objetos con los que tenía permitido interactuar.

La forma de utilizar esta política es concediendo los privilegios de forma **estática**, o sea el proceso durante toda su ejecución tendrá los mismos privilegios. O bien de forma **dinámica** donde el proceso podrá cambiar sus privilegios según vaya requiriendo, este concepto es conocido como **escalada de privilegios**.

Se deberá determinar a qué nivel de detalle se manejarán los privilegios, un nivel más tosco implicará un manejo más simple, sin embargo se deberá realizar a grandes rasgos o abarcando muchas operaciones u objetos. Un manejo más fino será más complejo de administrar, pero permitirá definir con precisión los privilegios para los procesos.

9.2. Dominios

Un dominio definirá un conjunto de derechos de acceso que el proceso (o usuario) tienen sobre determinados objetos, especificando para cada uno de estos objetos las posibles operaciones que el proceso puede realizar sobre el mismo.

En la figura 9.1 se puede observar un ejemplo con tres dominios, cada uno con un conjunto de derechos de acceso y las operaciones que pueden realizar sobre cada uno de los objetos. Notar que los dominios D_2 y D_3 comparten una misma operación sobre un objeto.

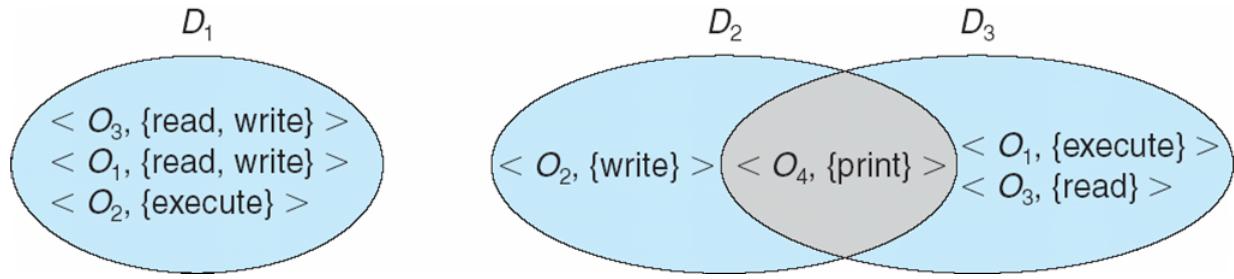


Figura 9.1: Ejemplo de dominios y sus conjuntos de derechos de acceso

9.2.1. Dominios en sistemas *like Unix*

En sistemas operativos *like Unix*, los dominios están definidos por el identificador del usuario (UID) y el o los grupos a los que pertenece el usuario (GID).

El cambio de dominio, específicamente el cambio de dominio definido por el UID, puede ser realizado de diferentes maneras:

- I. **Vía sistema de archivos:** cada archivo tiene asociado un bit (*setuid bit*) que permite indicar que cuando el archivo sea ejecutado se realice con el dominio del dueño del archivo y no con el dominio de quien ejecuta el archivo.
- II. **Vía contraseñas:** comando `su` que permite cambiar a otro dominio de usuario.
- III. **Vía comandos:** comando `sudo` que ejecuta un comando específico utilizando otro dominio. Generalmente el otro dominio es el del usuario con UID 0 (root), pero no necesariamente tiene que ser ese dominio.

9.3. Matriz de acceso

Las políticas de protección se pueden ver como una matriz (tabla) donde las filas representan los dominios, las columnas representan los objetos y cada casilla_{ij} de la tabla corresponde a las operaciones que un proceso ejecutándose en el dominio i puede realizar sobre el objeto j .

En la figura 9.2 se puede observar un ejemplo de una matriz de acceso donde, por ejemplo, el dominio D_3 puede leer el fichero F_2 pero no el F_1 . De la misma forma el único que puede imprimir es el dominio D_2 .

object domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Figura 9.2: Ejemplo de matriz de acceso

En general la forma de revisar la matriz de acceso es: si un proceso del dominio D_i trata de hacer una operación X en el objeto O_j , entonces la operación X debe estar en la *casilla* i,j en la matriz de acceso.

Se deben definir operaciones para agregar o quitar derechos de uso, algunas son:

- Definir propietario de $Objeto_j$.
- Copiar operación desde $Objeto_j$ a $Objeto_k$.
- Control: $Dominio_i$ puede modificar los accesos de $Dominio_h$.
- Transferencia: del dominio $Dominio_i$ al $Dominio_h$.

Para definir las operaciones sobre dominios es necesario que el dominio sea tratado como un objeto más dentro de la matriz de acceso, de esta forma se tendrá algo similar al ejemplo de la figura 9.3.

object domain \	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Figura 9.3: Ejemplo de matriz de acceso usando dominios como objetos

Finalmente es importante mencionar que la matriz de acceso separa políticas de mecanismos, teniendo que:

- Mecanismos:
 - Sistema operativo provee de matriz de acceso más reglas.
 - Asegura que la matriz sea solo manipulada por agentes autorizados.
 - Reglas de la matriz aplicadas estrictamente.
- Políticas:
 - Usuario indica las políticas.
 - Se define **¿quién puede acceder?** a que objeto y **¿en qué modo?**.

9.4. Ejercicios y preguntas

1. Explique el principio del menor privilegio.

2. ¿Por qué es recomendable denegar todo y solo asignar los permisos específicos que son requeridos?
3. Explique los métodos de asignación de privilegios estáticos y dinámicos.
4. ¿En qué consisten los dominios?.
5. ¿Qué son los derechos de acceso?.
6. ¿Cuáles son las tres vías para hacer un cambio de dominio en Unix?.
7. ¿Qué es la matriz de acceso?.
8. ¿Qué representan las filas y columnas en la matriz de acceso?.
9. ¿Por qué se podría querer tratar a los dominios como objetos en la matriz de acceso?.

9.5. Referencias

- Sistemas Operativos, Quinta Edición, Abraham Silberschatz y Peter Baer Galvin, Capítulo 19 y 20.
- Sistemas Operativos, Segunda Edición, William Stallings, Capítulo 14.

Anexo A

Máquinas virtuales

Una máquina virtual entrega una abstracción del hardware de la máquina hacia el sistema operativo, proporcionando una interfaz de hardware virtual similar a la de la máquina real. Los discos duros son emulados, por ejemplo, mediante imágenes de discos. El sistema operativo que corre sobre la máquina virtual desconoce tal condición, o sea, no sabe que funciona sobre una máquina virtual y no una real. Este tipo de sistemas permite correr múltiples sistemas operativos sobre una misma máquina. Ejemplos de sistemas de virtualización son KVM, XEN y VirtualBox.

El sistema operativo que corre sobre la máquina virtual también posee un modo de ejecución usuario y de sistema, sin embargo estos son modos virtuales que corren sobre un modo usuario real. Esto significa que si en el sistema operativo virtual hay una solicitud a una llamada del sistema a través de un programa que corre en modo usuario virtual, esta será procesada por el sistema operativo en modo sistema virtual y se entregará a la máquina virtual, la cual, en modo usuario real, atenderá la interrupción mediante el hardware virtualizado y entregará la respuesta al sistema operativo. En caso que se requiera acceso al hardware real, la máquina virtual deberá hacer uso de la API del sistema operativo real para acceder al recurso solicitado.

Es importante mencionar que los tiempos de respuesta en máquinas virtuales serán más

lentos que en máquinas reales. Lo anterior debido a la emulación que se debe realizar del hardware y por la posibilidad de que existan múltiples máquinas virtuales en ejecución en un mismo sistema real.

Las principales ventajas de esta solución es que permite realizar una protección por aislamiento de los recursos del sistema, ya que el sistema virtualizado solo verá dispositivos virtuales y en caso de cualquier problema solo podrá afectar a la máquina virtual quedando la máquina real protegida. Adicionalmente son un medio ideal para la realización de pruebas de sistemas operativos, como la prueba de módulos en desarrollo o la prueba de servicios que se desean implementar en una máquina real. También permiten aprovechar mejor el hardware disponible, entregando servicio en un mismo equipo a diferentes sistemas operativos que en conjunto comparten de forma eficiente el hardware disponible.