Apunte de Sistemas Operativos

Una visión general

(borrador)

Esteban De La Fuente Rubio

esteban[at]delaf.cl

1 de abril de 2014

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.

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 oyente[1] 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.

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

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

Índice de cuadros

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 **??** 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 esta interesado en ejecutar sus aplicaciones.

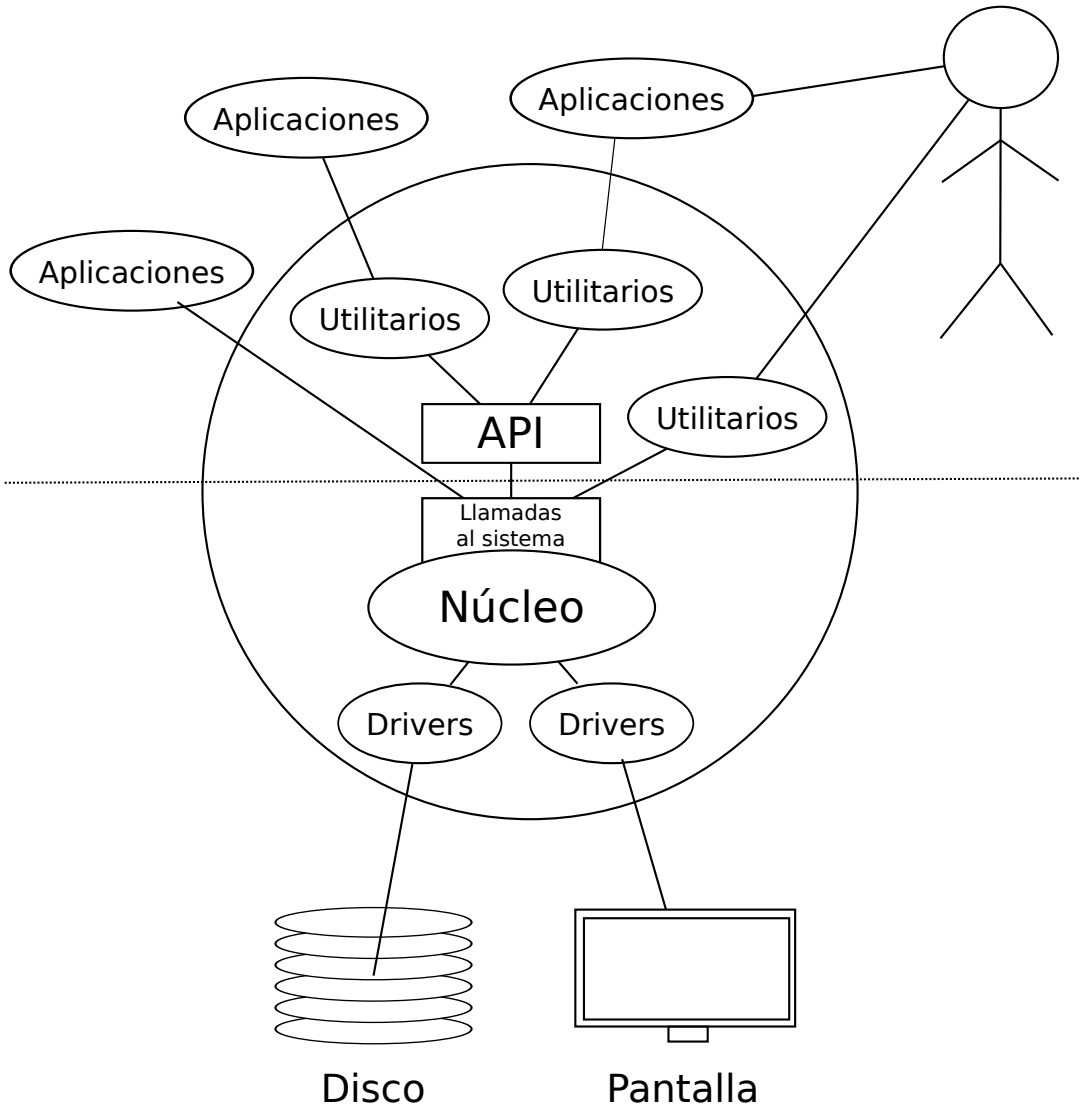


Figura 1.1: Visión global del Sistema Operativo

En la figura **??** se observa una línea divisora 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 **??**.

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 **??**, 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 **??**.

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 al 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 mantención 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 **??**, 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-

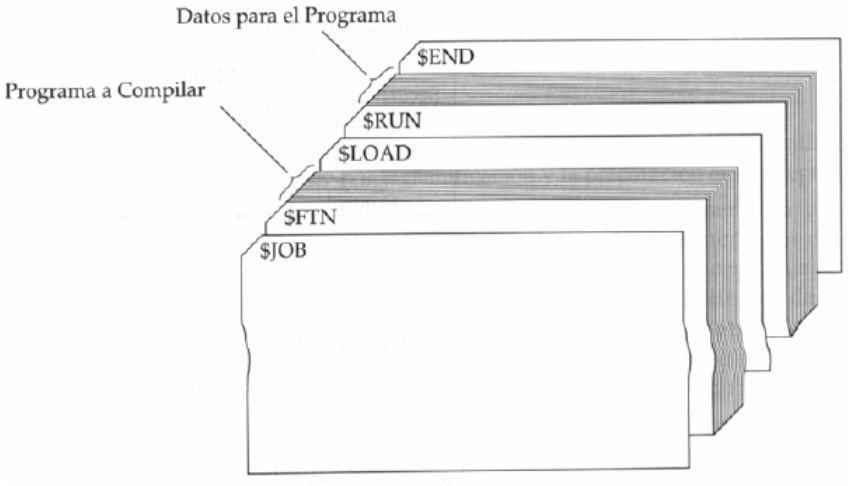


Figura 2.1: Tarjetas de un sistema por lotes

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 **??**.

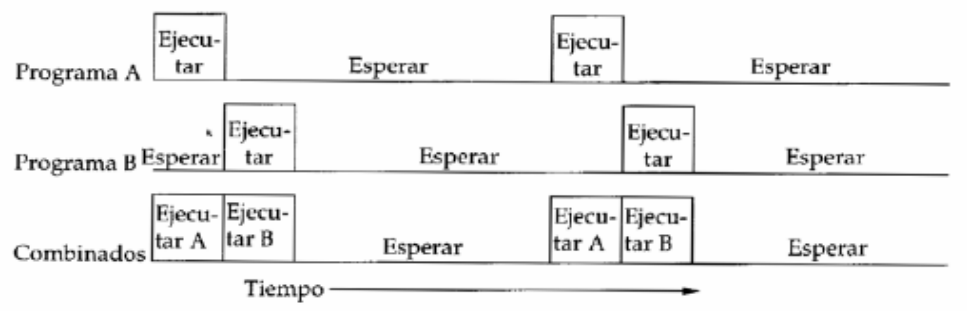


Figura 2.2: Sistemas por lotes vs sistema multiprogramado

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 FSF[1].

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,

[1]Free 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 **??** 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 mamyoría 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 comándos 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

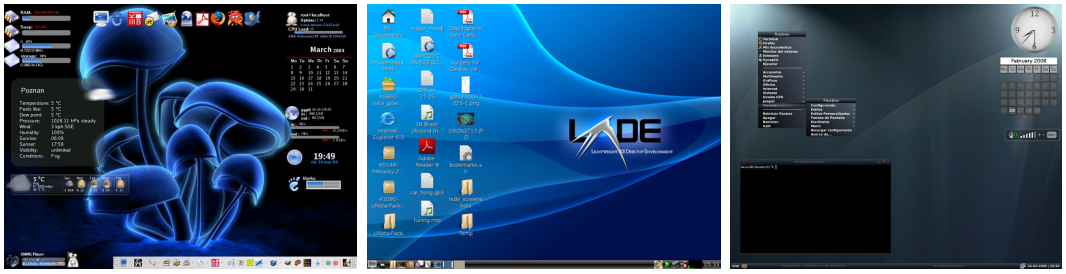
La interfaz de usuario gráfica corresponde al entorno de ventanas, el cual permite tener diversas aplicaciones encapsuladas dentro de un cuadro (ventana) y de esta forma compartir de manera fácil un único recurso, la pantalla, con múltiples procesos que quieren dibujar en ella. En sistemas *like Unix* es conocido como X en honor a Xerox que lo ideo en los años 70s[1].

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

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

(a) KDE 3.5 (b) KDE 4 (c) Gnome



(d) XFCE (e) LXDE (f) Fluxbox

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 *Aplication Program Interface* corresponde al conjunto de intrucciones 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 POSIX[2], 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.

[2]Portable 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 si, análogamente si se escribe en esa dirección de memoria se hará escritura en el disco. Esto es básicamente los 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 **??**. 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 esta 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** esta preocupado por que el sistema operativo sea cómodo de utilizar, fácil de aprender y usar, fiable, seguro y rápido.

El **sistema** esta 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 restriccioens 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 **??** 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, revocándoselo 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.

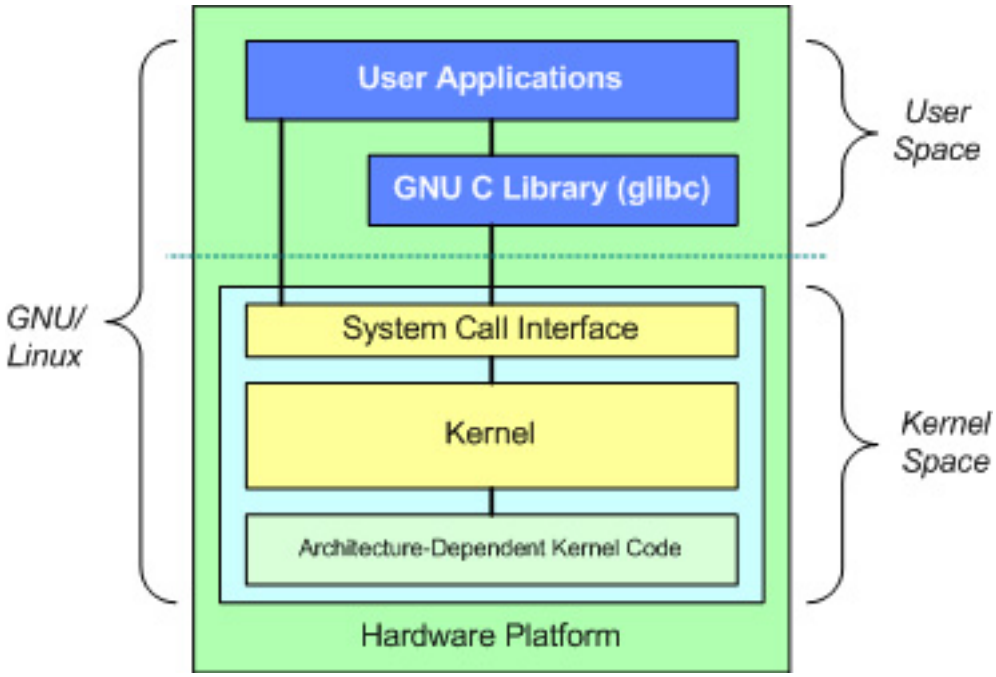


Figura 3.2: Componentes presentes en el modo usuario y el modo sistema

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 **??**.

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 multiusuarios. 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 esta 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.

[3]Computadoras 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 `rmmod`.

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 que 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 **??**.

- **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.

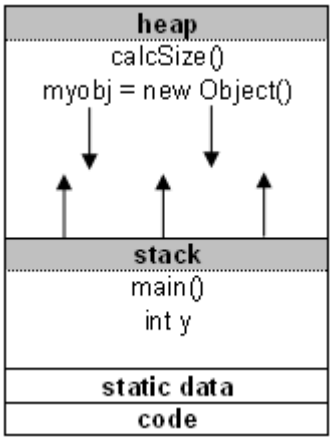


Figura 4.1: Distribución de la memoria

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 esta 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 esta 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 **??**. 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.

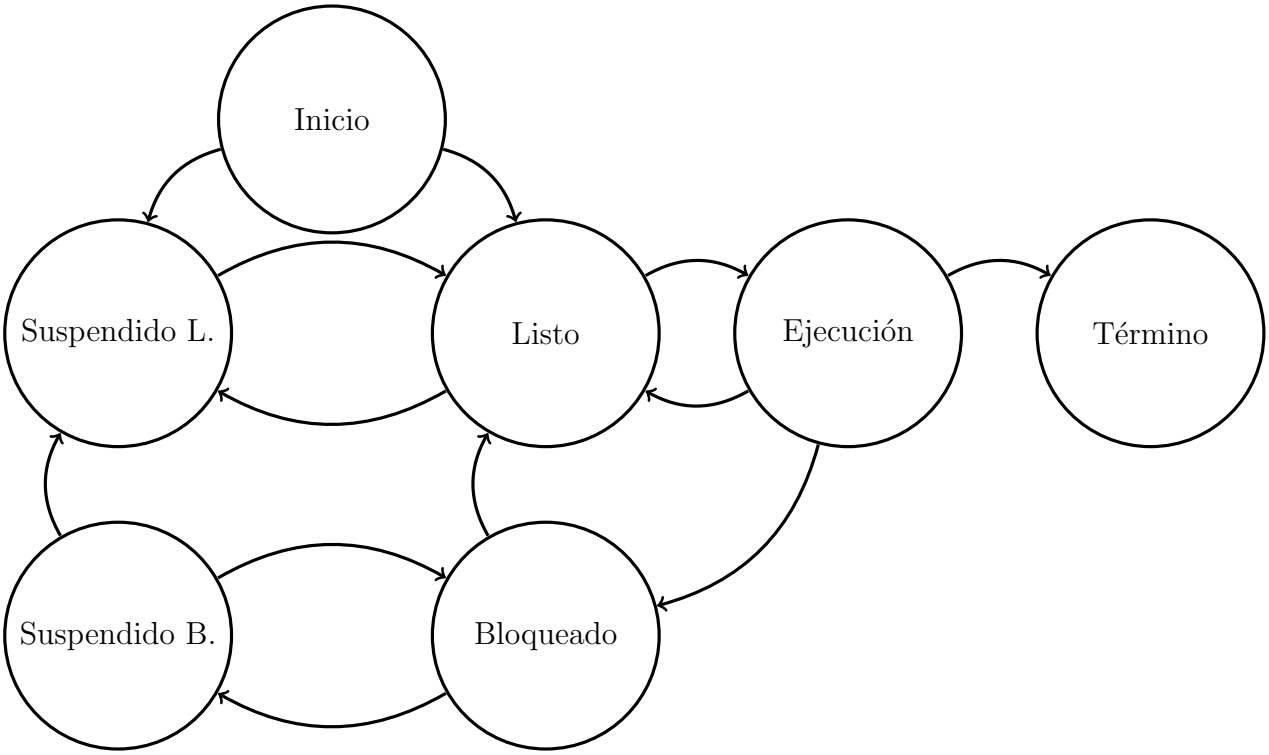


Figura 4.2: Estados de un proceso

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 asigna 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 **??** 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
Ejeplos 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 estás 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 *paralelismo*. 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 **??** 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 código[1]. Al existir la posibilidad que el sistema operativo interrumpa la ejecución del

[1]Se 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 problema 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 sillas[10]; /* arreglo inicializado en 0 */
2 void sentarme ()
3 {
4   int i;
5   for(i=0; i<10; ++i) {
6     if(sillas[i]==0) {
7       sillas[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 **??**.

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

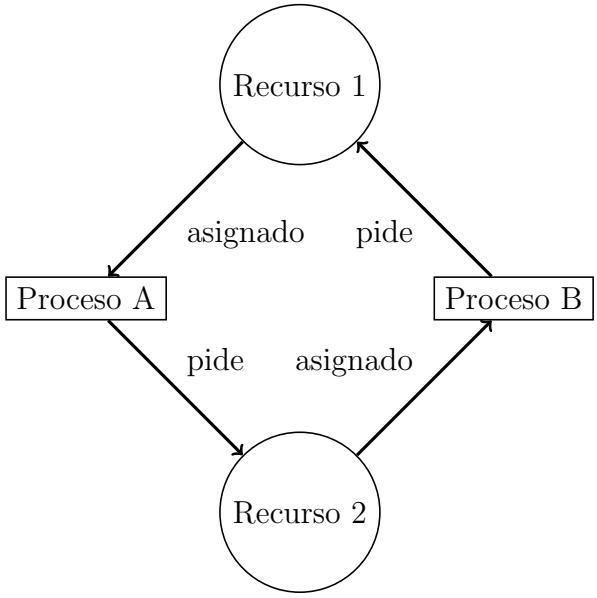


Figura 4.3: Interbloqueo, espera circular entre proceso A y B

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 seccion critica */
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 sucedera 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 que se utiliza el espacio de memoria *stack*?.
5. ¿Qué es un *segmentation fault*?.
6. Describa a que 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: *dataraces*, *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 interactuan 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
```

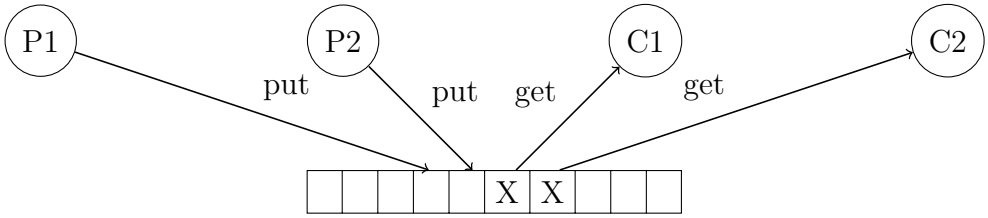


Figura 5.1: Interacción entre productores y consumidores

```
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).

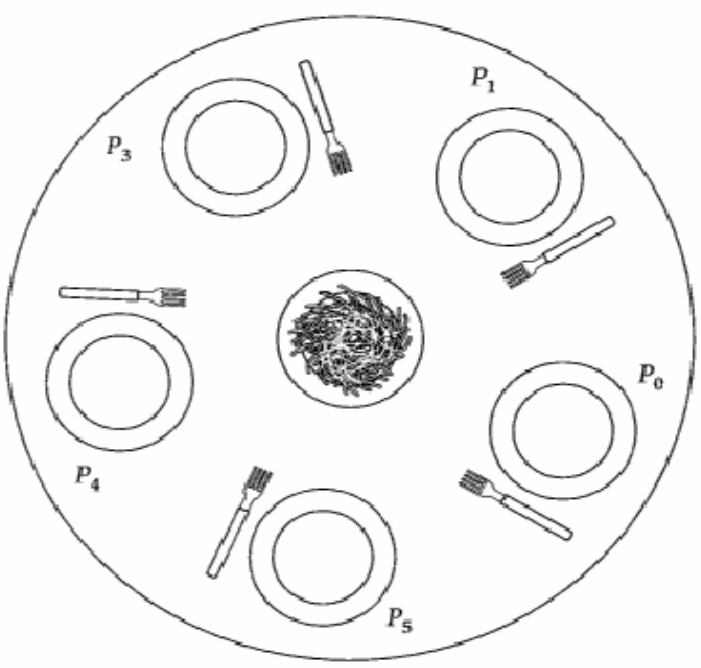


Figura 5.2: Problema cena de filósofos

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
```

[1]Sistema 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 (0) */
6  struct semaphore *full;  /* = semaphore_make (N) */
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 Hansen[2] 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.

[2] `http://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  // ejecuci n de secci n cr tica
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 propieda
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

quedar 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 que 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 modifica
9                                  datos que afecten la condicion de otros qu
10             esperan (adem s podr a 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 palito
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 esta 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 esta 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 liberá 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 en
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 paras las funciones `enterRead` y `enterWrite`. El resto es igual.

```
1 int visor = 0;          /* la idea es que del distribuidor se saca un "tur
2 int distribuidor = 0;      solo se avanza cuando el visor asi 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 [
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 desbloquear  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 **??** 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.

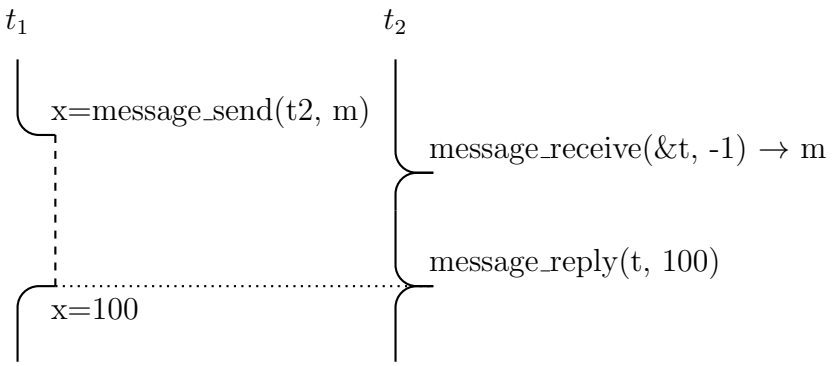


Figura 5.3: Ejemplo 1 de uso de mensajes

En la figura **??** 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.

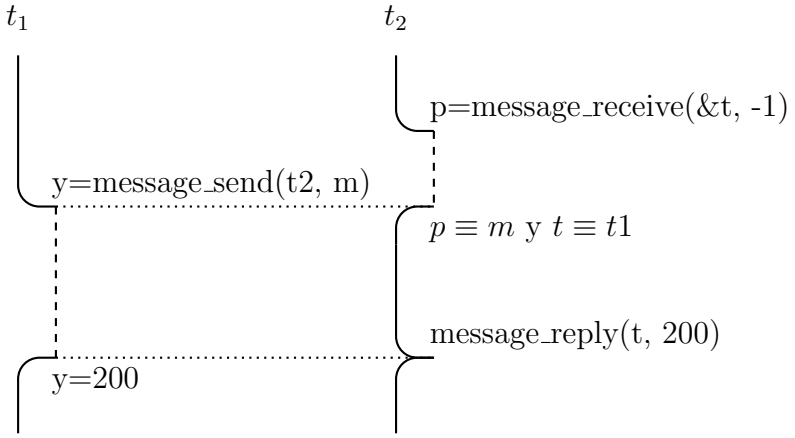


Figura 5.4: Ejemplo 2 de uso de mensajes

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 funcion 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 **??**

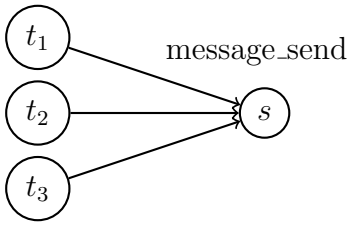


Figura 5.5: Ejecución de la función f en el servidor

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

[3] `http://es.wikipedia.org/wiki/X_Window_System`

```
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 funci n 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
7  Sem semMake (int ini)
8  {
```

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

```
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 ineficiente 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 eemplazado 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 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 que 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.

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.