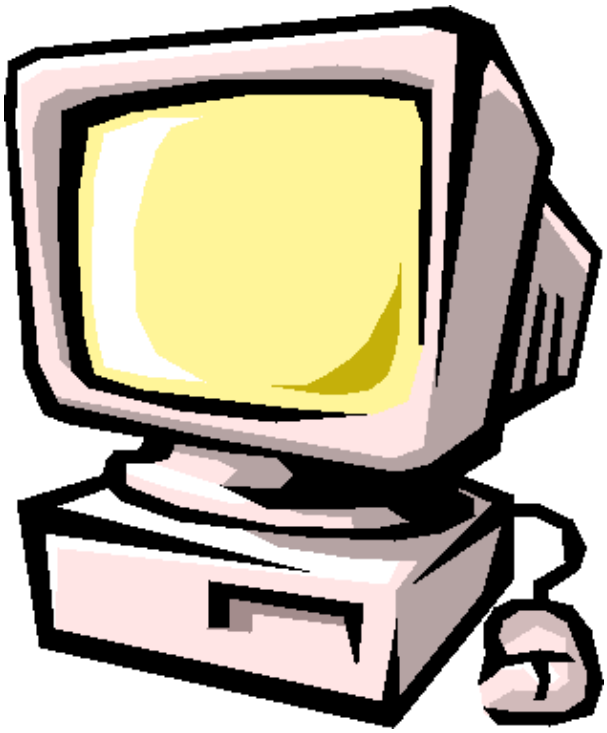
SISTEMAS OPERATIVOS

(Resumen)



UNIVERSIDAD MARIANO GALVES DE GUATEMALA (UMG)

INGENIERIA EN SISTEMAS DE INFORMACION

Elaborado el: 24 de Marzo 2003

UNIDAD 1 : Introducción

DIVISIONES DEL SOFTWARE

- *Programas de Sistema:* Controlan la operación de la PC.
- *Programas de Aplicación:* Hacen tareas que el usuario desea.

DEFINICIÓN DE UN SISTEMA OPERATIVO (S.O.)

Es el *programa de sistema* más fundamental y es el que controla todos los recursos de la computadora. Evita que los programadores tengan que ocuparse del manejo de dispositivos como disquetes.

MISIÓN DEL SISTEMA OPERATIVO

Asegurar un reparto ordenado y controlado de los procesadores, memorias y dispositivos de E/S entre los diferentes programas que compiten por ellos, como el manejo de impresiones para varios usuarios haciendo ordenado el envío al dispositivo.

COMPONENTES PRINCIPALES

- Manejo de Procesos
- Manejo de E/S
- Manejo de Memoria
- Sistema de archivos

DEFINICIÓN DE UNA MAQUINA VIRTUAL (MAQUINA EXTENDIDA)

Así se denomina la capa de software, a manera e interfaz, encima del hardware, llamada sistema operativo, que evita la complejidad de dispositivos y recursos a los programadores, ofreciendo facilidad en el uso de dispositivos.

EL ARRANQUE DE UN SISTEMA OPERATIVO

Cuando se enciende la computadora, el hardware lee el primer sector de la primera pista del disco de arranque y lo coloca en la memoria, luego ejecuta el código que encuentra ahí. Los detalles varían entre un disco duro y un disquete. En un disquete este sector contiene el programa de **auto arranque (bootstrap)**, que es muy pequeño, pues debe caber en un sector. Este programa cara un programa más grande, **boot**, que luego carga al sistema operativo propiamente dicho.

En cambio, los discos duros requieren un paso intermedio. Un disco duro está dividido en **particiones**, y el primer sector de un disco duro contiene un pequeño programa y la **tabla de particiones** del disco. Colectivamente, éstos se conocen como **registro maestro de arranque**. La parte de programa se ejecuta para leer la tabla de particiones y seleccionar la **partición activa**, la cual tiene un auto arranque en su primer sector, mismo que se carga y ejecuta para encontrar e iniciar una copia de boot en la partición, exactamente como se ha cuando se arranca con un disquete.

En ambos casos, boot busca un archivo multi partes en el disquete o la partición y carga las partes individuales en las posiciones apropiadas de la memoria. Estas partes incluyen el Kernel, administrador de memoria, sistema de archivos e init, el primer proceso del usuario. Una vez que todos se han ejecutado e inicializado en orden, se bloquean, esperando algo que hacer. Una vez que todos están bloqueados, se ejecuta init para comenzar a trabajar.

EVOLUCION DE LAS COMPUTADORAS

Primera generación (1945−1955): tubos de vacío

Lo cierto es que el primer computador digital fue diseñado por el matemático inglés Charles Babbage hace cerca de siglo y medio. Era un computador totalmente mecánico, que Babbage nunca pudo construir, principalmente debido a que la tecnología de la época no era capaz de producir las piezas con la precisión requerida. Después de eso, poco se hizo hasta la segunda guerra: alrededor de 1940 se construyeron las primeras máquinas calculadoras usando tubos de vacío. Estas máquinas de varias toneladas de peso eran diseñadas, construidas, programadas y operadas por el mismo grupo de personas. No había ni lenguajes de programación, ni compiladores; mucho menos sistema operativo. Cada programa se escribía en lenguaje de máquina, usando tableros con enchufes e interruptores y tenía que manejar todo el sistema (lo que era factible gracias a que el programador era el mismo que diseñó y construyó la máquina). Con el tiempo se introdujeron las tarjetas perforadas, para reemplazar al tablero, pero el sistema era esencialmente el mismo.

Segunda generación (1955−1965): transistores y procesamiento por lotes

La introducción de los transistores permitió aumentar sustancialmente la confiabilidad de los computadores, lo que a su vez hizo factible construir máquinas comerciales. Por primera vez hubo una separación entre diseñadores, constructores, y programadores.

La aparición de los primeros compiladores (de FORTRAN) facilitó la programación, a costa de hacer mucho más compleja la operación de los computadores. Por ejemplo, para probar un programa en FORTRAN recién escrito, el programador debía esperar su turno, y:

Cargar compilador de FORTRAN, típicamente desde una cinta magnética.

Poner el alto de tarjetas perforadas correspondientes al programa FORTRAN y correr el compilador.

El compilador genera código en assembler, así que hay que cargar ahora el ensamblador para traducirlo a lenguaje de máquina. Este paso requiere poner otra cinta con el ensamblador. Ejecutar el ensamblador, para generar el programa ejecutable.

Al Ejecutar el programa.

Si hay errores en cualquiera de estos pasos, el programador debía corregir el programa y comenzar todo de nuevo. Obviamente, mientras el programdor ponía cintas y tarjetas, y mientras se devanaba los sesos para descubrir por qué el programa no funciona, la CPU de millones de dólares de costo se mantenía completamente ociosa. Más que rápido, se idearon mecanismos para mantener a la CPU siempre ocupada. El primero fue separar el rol de programador del rol de operador. Un operador profesional demoraba menos en montar y desmontar cintas, y podía acumular lotes de trabajos con requerimientos similares: por ejemplo, si se acumula la compilación de varios programas FORTRAN, entonces el compilador de FORTRAN se carga una sola vez para todos los trabajos.

Aún así, en la transisión de un trabajo a otro la CPU se mantenía desocupada. Aparecieron entonces los *monitores residentes*, que fueron los precursores de los sistemas operativos. Un monitor residente es un pequeño programa que está siempre en la memoria del computador, desde que éste se enciende. Cuando un programa termina, se devuelve el control al monitor residente, quien inmediatamente selecciona otro programa para ejecutar. Ahora los programadores, en vez de decirle al operador qué programa cargar, debían informárselo al monitor (mediante tarjetas de control especiales):

```
$JOB

$FTN

programa FORTRAN

$LOAD

$RUN

datos para el programa

$END
```

Esto se conoce como procesamiento por lotes: el programador deja su alto de tarjetas, y después vuelve a retirar la salida que se emite por la impresora (y que puede ser nada más que la notificación de que el programa tenía un error de sintaxis).

Tercera generación (1965−1980): circuitos integrados y multiprogramación

El procesamiento por lotes evita que la CPU tenga que esperar tareas ejecutadas por lentos seres humanos. Pero ahora el cuello de botella se trasladó a los dispositivos mecánicos (impresoras, lectoras de tarjetas y de

cinta), intrínsecamente más lentos que que las CPUs electrónicas. Para resolver esto, aparece, dentro de la tercera generación de computadores, la multiprogramación: varios trabajos se mantienen permanentemente en memoria; cuando uno de ellos tiene que esperar que una operación (como grabar un registro en cinta) se complete, la CPU continúa con la ejecución de otro trabajo. Si se mantiene un número suficiente de trabajos en la memoria, entonces la CPU puede estar siempre ocupada.

Pero el sistema sigue siendo esencialmente un sistema de procesamiento por lotes; los programadores no interactúan *en línea* con el computador, los tiempos de respuesta desde que se deja un trabajo para ejecución hasta conocer el resultado son demasiado grandes. (¡En cabio, los computadores de primera generación eran interactivos!) De ahí nace el concepto de *tiempo compartido* que es una variante de la multiprogramación en la cual una CPU atiende simultáneamente los requerimientos de varios usuarios conectados en línea a través de terminales. Ya que los usuarios humanos demoran bastante entre la emisión de un comando y otro, una sola CPU es capaz de atender, literalemente, a cientos de ellos simultáneamente (bueno, en realidad, uno después de otro, pero los usuarios tienen la ilusión de la simultaneidad). Al mismo tiempo, cuando no hay ningún comando que ejecutar proveniente de un usuario interactivo, la CPU puede cambiar a algún trabajo por lote.

Cuarta generación (1980−): computadores personales

Con el advenimiento de la integración a gran escala, que permitió concentrar en un solo chip miles, y luego millones de transistores, nació la era de la computación personal. Los conceptos utilizados para fabricar los sistemas operativos de la tercera generación de computadores siguen siendo, en general, adecuados para la cuarta generación. Algunas diferencias destacables:

Dado los decrecientes costos de las CPUs, ya no es nada de grave que un procesador esté desocupado.

La creciente capacidad de las CPUs permite el desarrollo de interfaces gráficas; buena parte del código de los sistemas operativos de hoy es para manejar la interfaz con el usuario.

Los sistemas paralelos (un computador con varias CPUs), requieren sistemas operativos capaces de asignar trabajos a los distintos procesadores.

Las redes de computadores y sistemas distribuidos abren nuevas posibilidades e imponen nuevas obligaciones a los sistema operativo.

EVOLUCION DE LOS SISTEMAS OPERATIVOS (ESTRUCTURAS)

(1) MONOLÍTICOS

- Todo el sistema es un conjunto de procedimientos, sin privacidad que pueden llamarse uno a otro cuando se quiera. Todo compilado en un solo archivo objeto unido.
- Solo tienen una pequeña estructuración donde para las llamadas al sistema se hacen colocando parámetros en lugares previamente definidos, como registros o pilas y después ejecutando una instrucción especial (trampa o interrupción) de Kernel para ejecutar un procedimiento (llamada supervisora). Una vez hecha la llamada, se detecta en los parámetros si el nuevo proceso correrá en modo Kernel o modo usuario.
- Existen 3 capas: (1) procedimientos principales (2) procedimientos de servicio (3) procedimientos de utilidades.

(2) SISTEMAS DE ESTRATOS (CAPAS)

Con esto se pretende establecer un sistema operativo más sólido sobre una clasificación jerárquica de procedimientos en estratos o capas. Por ejemplo:

CAPA	FUNCIONES	Cada capa sabe que puede contar con las otras para realizar sus funciones yendo del 4 al 0
4	Programas del usuario.	
3	Administración de E/S.	
2	Comunicaciones.	
1	Administración de Memoria.	
0	Multiprogramación.	

(3) MAQUINAS VIRTUALES

Se creo sobre la observación que un sistema operativo de tiempo compartido se puede dividir en: (1) Multiprogramación y (2) Interfaz amigable con el usuario. Sobre esto se dividieron las dos áreas, teniendo un corazón llamado *Monitor de Máquina Virtual* que ofrece varias máquinas virtuales a la siguiente capa de interfaz de programación, siendo cada máquina una copia exacta de la máquina real con su modo Kernel / Usuario, E/S, interrupciones y demás.

Esto permite la ejecución de diferentes sistemas operativos sobre la misma computadora real. Simulando ASAMBLER o MICROPROGRAMACIÓN a nivel de Hardware. Por e*jemplo: Intel puso esto en Pentium para emular el procesador del D.O.S. o ejecutar Windows.*

Actualmente se pueden tener por ejemplo 2 máquinas virtuales con recursos diferentes para su ejecución y un EXOKERNEL que se encarga de coordinar y administrar recursos de ambas.

Ventajas: facilita el desarrollo de sistemas operativos (cuando la máquina es cara y no se puede entregar una a cada programador). Sin este esquema, es necesario bajar la máquina cada vez que quiere probar una modificación al sistema. También permite resolver problemas de compatibilidad. Por ejemplo, si queremos desarrollar un nuevo sistema operativo, pero queremos que los miles de programas DOS que existen sigan corriendo, podemos usar esta solución, asignando una máquina virtual a cada programa DOS que se ejecute. Se puede ir más lejos, y crear una máquina virtual distinta a la original. Así es como Windows para arquitectura Intel se puede ejecutar, por ejemplo en una Silicon Graphics. El problema es que, en este caso, hay que interpretar cada instrucción (en lugar de ejecutarla directamente).

(4) CLIENTE / SERVIDOR

La idea es desplazar el código del sistema a capas superiores, dejando el código del Kernel al mínimo. El método más usado es que las funciones del sistema operativo se hagan en modo usuario y que las terminales envíen la solicitud a un servidor que haga el trabajo, buscando que el Kernel se ocupe solo de comunicaciones entre los clientes y servidores.

En teoría ofrece la ventaja que al ejecutarse los procesos en el servidor en modo usuario, si ocurre una falla, el hardware no se cae, únicamente el servicio.

Otra ventaja es su adaptabilidad a sistemas distribuidos, donde al cliente no le afecta quién atiende su solicitud. Únicamente recibe la tarea realizada de

vuelta.

MODOS DE EJECUCIÓN

Existen 2 modos:

- *Modo Usuario:* Es cuando un usuario hace una llamada al sistema operativo para realizar una tarea. En este modo sólo se permiten algunas operaciones por la protección que tiene el sistema operativo.
- *Modo Supervisor (Kernel) :* Es cuando el sistema se encarga de hacer lo que el usuario solicitó. Toda operación sobre dispositivos se permite.

Estos dos modos cambian constantemente entre si, con lo que se da un cambio de contexto.

TIPOS DE PROTECCION

- PROTECCIÓN E/S: Las instrucciones E/S solo se pueden ejecutar en modo sistema para prevenir un daño por el usuario y asi el sistema filtra los datos.
- PROTECCIÓN DE MEMORIA: Como se verá más adelante en administración de memoria se divide en base y límite para protegerla.
- CONTROL DEL CPU: Se previene que no se quite el control al sistema operativo si sucediera algo como un ciclo infinito en un programa de usuario.

La idea es que el usuario le pide al sistema operativo que haga estas tareas por él a través de llamadas al sistema.

LLAMADAS AL SISTEMA

Son instrucciones extendidas que el sistema operativo ofrece como una interfaz entre los programas de usuario y el sistema operativo como tal.

Modo Usuario Llamada al sistema *Modo Supervisor*

Hace pasa a

INTERRUPCIONES

Existen interrupciones en software, análogas a las del hardware. El sistema operativo se vale de interrupciones para notificar a los procesos mensajes de error o envíos de información requerida.

Las llamadas al sistema, en la mayoría de los sistemas operativos se hace por medio de una interrupción de software o *trap* a una ubicación específica del vector de interrupciones. Ejemplo hipotético, (parecido a MSDOS), para borrar un archivo:

- Poner en registro A el código de la operación: 41h = borrar archivo
- Poner puntero al nombre del archivo en registro B
- INT 21h (generar interrupción de software)

Como la llamada es a través de una interrupción, se hace lo de siempre: cambiar a modo protegido, y pasar control a la dirección en la posición 4*21h del vector de interrupciones, o sea, a la rutina que maneja las llamadas al sistema (o tal vez, la rutina que maneja las llamadas que tienen que ver con archivos; puede que otras clases de llamadas sean atendidas por medio de otras interrupciones). Dicha rutina examina el registro A para determinar qué operación debe ejecutar, y después determina el nombre del archivo a través del registro B. En seguida, decide si la operación es válida: puede que el archivo no exista o pertenezca a otro usuario; según eso, borra el archivo o "patalea" de alguna manera.

Cuando uno hace esto en un lenguaje de alto nivel, como C, simplemente escribe:

```
char* Nombre = "Datos";

remove(Nombre);
```

El compilador de C es quien se encarga de traducir esto a una llamada al sistema, ocultando el detalle de la interfaz con el sistema operativo.

VECTOR DE INTERRUPCIONES

Área de memoria que guarda la dirección inicial de un proceso al ocurrir una interrupción.

Dado que existe una cantidad limitada (y reducida) de tipos de interrupciones posibles, se usa una tabla con las direcciones de los servidores de cada interrupción, típicamente en las primerísimas posiciones de la memoria.

Esta tabla se conoce como *vector de interrupciones*. Así por ejemplo, si las direcciones ocupan 4 bytes, la dirección del servidor de la interrupción *i* se guarda en los 4 bytes a partir de la posición *4i* de la memoria.

Se require la protección de modo sistema porque un programa de usuario podría poner una dirección que apunte a una rutina propia en el vector de interrupciones. Así, cuando se produzca la interrupción, el hardware cambiaría a modo sistema, y pasaría el control a la rutina del usuario.

CATEGORÍAS PRINCIPALES

- Referentes a Procesos
- Referentes a archivos

LLAMADAS REFERENTES A ARCHIVOS

La referente a procesos se ve en la siguiente unidad. Básicamente la de archivos se refiere a que el sistema operativo maneja los archivos con rutas en forma de árbol donde la raíz es el primer nodo de un dispositivo E/S, que a su vez es un nodo de dispositivos E/S que tiene la computadora.

El usuario continuamente hace llamadas referentes a archivos como escribir o leer datos en archivos o dispositivos E/S.

CLASIFICACION DE LOS SISTEMAS OPERATIVOS (CAPAS)

<table>
<tr><td>Sistema bancario</td><td>Reservaciones aéreas</td><td>Navegador WEB</td><td>Programas de Aplicación</td><td></td></tr>
<tr><td>Compiladores</td><td>Editores</td><td>Intérprete de comandos−SHELL</td><td rowspan="5">Programas de Sistema</td><td></td></tr>
<tr><td colspan="3">Sistema Operativo</td><td></td></tr>
<tr><td colspan="3">Lenguaje de máquina</td><td>Hardware</td></tr>
<tr><td colspan="3">Microprogramación</td><td></td></tr>
<tr><td colspan="3">Dispositivos físicos</td><td></td></tr>
</table>

DISPOSITIVOS FISICOS

Circuitos integrados (CHIPS), alambres, fuentes de potencia y otros aparatos físicos similares.

MICROPROGRAMACIÓN

Es una programación en los dispositivos físicos, normalmente de sólo lectura que ejecuta cada instrucción, del lenguaje máquina definido en la siguiente capa, en varios pasos de operaciones sobre el dispositivo físico.

LENGUAJE MAQUINA

Es el conjunto de instrucciones para hacer funcionar a los dispositivos, pero no todas las computadoras hacer pasan por la microprogramación, por ejemplo el procesador IBM POWER PC o computadoras RISC *(computadoras con conjunto de instrucciones reducido)* ejecuta las instrucciones de lenguaje máquina directamente, a diferencia del MOTOROLA 680X0 que si tiene microprogramación.

PROGRAMAS DE SISTEMA

Estos programas no son parte del sistema operativo, aunque generalmente son dados por el fabricante de la computadora.

El sistema operativo es la porción de software que corre ya sea en **modo** *Kernel o modo supervisor* y está protegido contra la intervención del usuario.

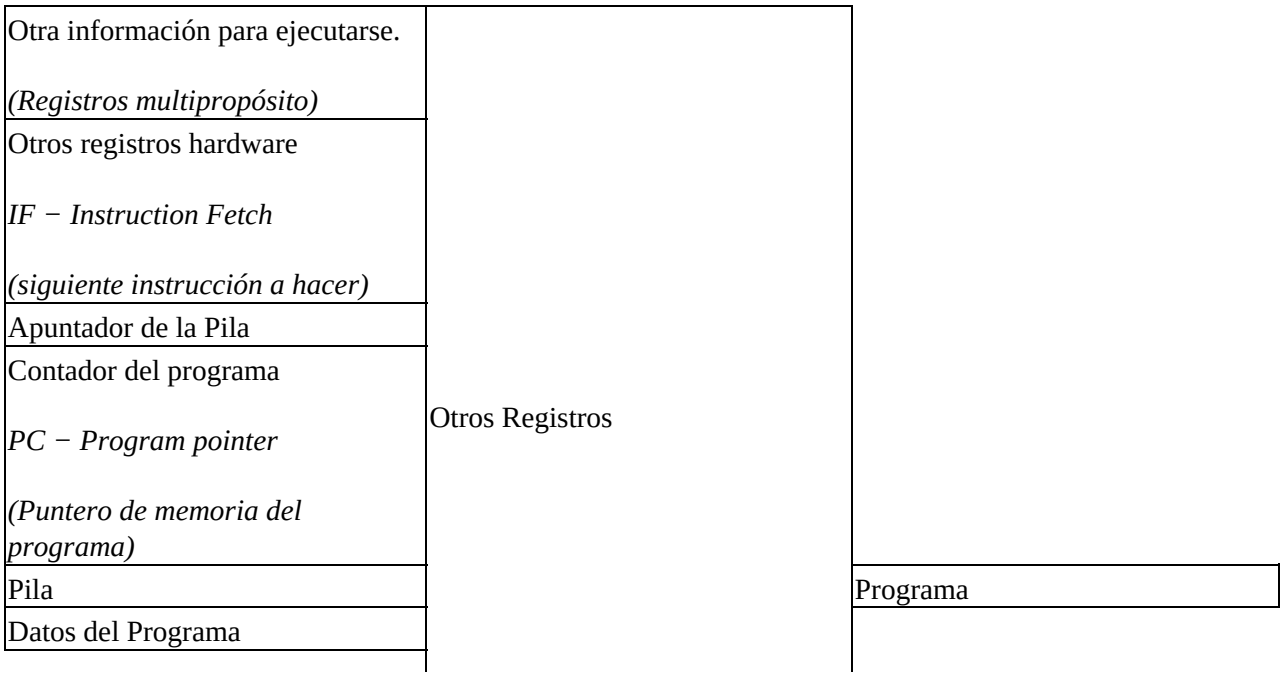
PROGRAMAS DE APLICACION

Los usuarios compran o escriben estos programas para resolver problemas particulares.

UNIDAD 2 : Procesos

DEFINICIÓN DE PROCESO

Es un programa en ejecución. Cada proceso tiene asociado un espacio de direcciones en memoria con un mínimo, generalmente 0, hasta un máximo, donde puede leer o escribir, teniendo en este espacio la siguiente información:



Programa Ejecutable *IR − Instruction Register* *(Instrucciones del programa)*	

MULTIPROGRAMACION (TIME−SHARING)

Significa que se pueden ejecutar varias tareas o procesos, mediante el cambio de una proceso a otro (conmutar o ***context−switch***), aprovechando el tiempo desperdiciado en esperas E/S. También hace posible el trabajo **multiusuario.**

Con la conmutación de procesos ya no se ejecutan todos los procesos con una misma duración o un tiempo dado, menos reproducible. Es por esta razón que en adelante ***Nunca se hacen suposiciones de tiempo de ejecución*** en procesos.

LLAMADAS AL SISTEMA DE ADMINISTRACIÓN DE PROCESOS

Las Principales son:

- Creación de Procesos
- Terminación de Procesos

Algunas otras que se dan son:

- Solicitar más memoria.
- Liberar memoria no utilizada.
- Esperar a que un proceso hijo termina.
- Superponer otro programa al actual.

Todo el manejo de las interrupciones y de los detalles de echar a andar y suspender procesos están escondidos en el administrador de procesos, bastante pequeño y que además planifica la ejecución de los mismos.

Algunos procesos están ejecutando programas en respuesta a comandos emitidos por los usuarios (o sea, son **procesos de usuario o interactivos**) y otros procesos forman parte del sistema; se encargan, por ejemplo de atender peticiones de otros procesos para manipular archivos, o de manejar algún dispositivo.

ESTADOS DE PROCESOS

- Ejecutándose (usando al CPU).
- Listo (se puede ejecutar, pero se suspendió temporalmente para dejar que otro proceso se ejecute).
- Bloqueado (no puede ejecutarse en tanto no ocurra algún evento externo.

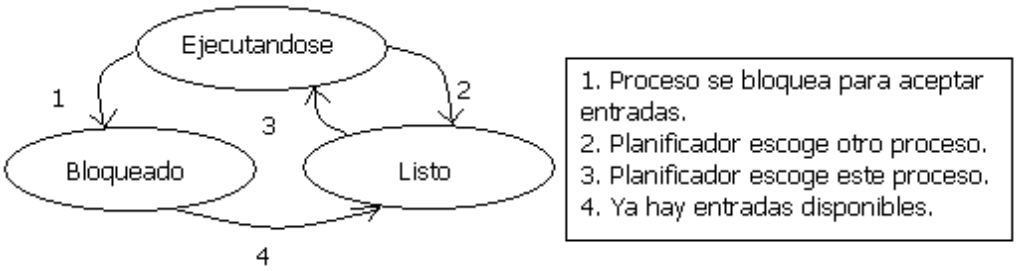


TABLA DE PROCESOS

Esta tabla existe porque periódicamente, el sistema operativo decide dejar de ejecutar un proceso y comenzar otro, volviéndose necesario el tener un lugar para guardar todos los datos al momento de suspender un proceso para después reanudarlo en el mismo lugar donde se dejó. *Por ejemplo: El apuntador de un registro en un archivo abierto para leer la información en ese registro al reanudar el proceso*.

En esta tabla se guardan estos datos junto con la dirección y espacio en memoria que ocupa el proceso suspendido. Todo se guarda en forma de una lista:

Info. Del Nodo Info. Del Nodo Info. Del Nodo

Imagen del Núcleo Imagen del Núcleo Imagen del Núcleo

Algunos campos importantes para los procesos son:

- Registros
- Contador de programa
- Estado del programa
- apuntador a la pila
- Hora de inicio del proceso.
- Tiempo del CPU usado
- Tiempo de CPU de los hijos
- Identificador del proceso.

se le llama **IMAGEN DEL NÚCLEO** al espacio de direcciones guardado en la tabla de procesos para un proceso suspendido.

PROCESOS HIJO

Es cuando un proceso crea a otro. Esto va formando una estructura de árbol para los procesos en la tabla de procesos (lista enlazada):

P1 P2 (Tabla de Procesos)

P1A P1B (Árboles binarios con hijos)

P1A1 P1A2

HILOS DE CONTROL (THREADS)

Los procesos hijos pueden corren en espacios de memoria diferente o en el mismo espacio que el padre. La ventaja es compartir memoria y recursos entre procesos como conexiones a servidores o que queden en espacios de memoria independientes, conexiones independientes. La tabla que guarda la conmutación de procesos varía dependiendo del manejo de hilos en el mismo espacio en memoria o distinto.

No hay protección entre threads: nada impide que un thread pueda escribir, por ejemplo, sobre el stack de otro.

Las ventajas de resolver esta clase de problemas usando dos procesos, es que tenemos:

- Modularidad, ya que cada proceso realiza una función bien específica.

- Disminución en el tiempo de ejecución en caso que contemos con más de un procesador, ya que los procesos pueden trabajar en paralelo.

Las desventajas son:

- Los procesos deben sincronizarse entre sí, y deben compartir el buffer. ¿Cómo puede hacerse esto si los procesos tiene espacios de direccionamiento disjuntos? (Recordemos que el sistema operativo no permite que un proceso escriba en la memoria que le corresponde a otro proceso). Veremos formas de hacerlo, pero, por ahora, lo importante es que no es "natural": hay que hacerlo explícitamente.
- Para que la CPU suspenda un proceso y retome otro, debe hacer lo que se conoce como *cambio de contexto (context−switch*), y para eso, se requiere guardar el estado del proceso que está ejecutando, y cargar el estado que se había guardado del nuevo proceso. El estado de un proceso comprende el PC y los registros de la CPU. Además, si se usan las técnicas de administración de memoria que veremos más adelante, hay más información involucrada. Este cambio es puro *overhead*, puesto que entretanto la CPU no hace trabajo útil (ningún proceso avanza). Considerando que la CPU hace varios cambios de contexto en un segundo, su costo es relativamente alto. Además, al cambiar de un proceso a otro, como los espacios de direccionamiento son disjuntos, hay otros costos indirectos, que tienen relación con el caché de la CPU.

Ejemplo: servidor de archivos, que recibe solicitudes para leer y escribir archivos en un disco. Para mejorar el rendimiento, el servidor mantiene un caché con los datos más recientemente usados, en la eventualidad que reciba algún requerimiento para leer esos datos, no es necesario accesar el disco. Cuando se recibe una solicitud, se le asigna a un thread para que la atienda. Si ese thread se bloquea porque tuvo que accesar el disco, otros threads pueden seguir atendiendo otras solicitudes. La clave es que el buffer debe ser compartido por todos los threads, lo que no podría hacerse si se usa un procesos pesados para cada solicitud.

UNIDAD 3: Planificación de Procesos

La parte del sistema operativo que decide la ejecución entre varios procesos se llama **planificador** y el algoritmo que usa se llama **algoritmo de planificación**.

Casi todas las computadoras tienen incorporado un cronómetro o reloj electrónico que genera interrupciones periódicamente. Es común una frecuencia de 60 o 60 interrupciones por segundo (50 o 60 Hz.), pero hoy en día, en muchas computadoras el sistema operativo puede ajustar la frecuencia del cronómetro.

NIVELES DE PLANIFICACIÓN

Es cuando los procesos tienen parte de sus datos en memoria secundaria, con un acceso más lento por lo que tenemos los siguientes niveles:

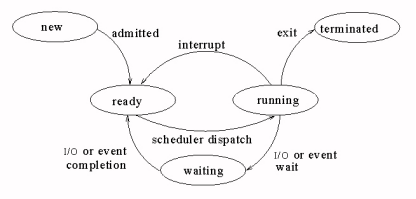
- A CORTO PLAZO: Es la ejecución de varios procesos en memoria principal.
- A MEDIANO PLAZO: Es el cambio de memoria principal a secundaria para la ejecución de otro proceso.
- A LARGO PLAZO: Es planificar de antemano y sobre datos como estadísticas, los procesos que habrá que cargar a memoria principal de la secundaria.

FORMAS DE PLANIFICACIÓN (APROPIATIVA)

- PLANIFICACIÓN NO APROPIATIVA: Se daba al principio y los procesos se ejecutaban hasta terminar.
- PLANIFICACIÓN APROPIATIVA: Estrategia de suspender procesos lógicamente ejecutables.

CRITERIOS LOS ALGORITMOS

- EQUITIVIDAD: Cada proceso recibe una parte justa del tiempo de CPU.
- EFICIENCIA: Mantener el CPU ocupado todo el tiempo.
- TIEMPO DE RESPUESTA: Minimizar el tiempo de respuesta para usuarios interactivos.
- TIEMPO DE ESPERA: Minimizar el tiempo medio de espera en la cola de listos en los procesos.
- VOLUMEN DE PRODUCCIÓN: Maximizar el numero de trabajos procesados por hora.



ALGORITMOS DE PLANIFICACIÓN:

(1) ALGORITMO DE PLAZO FIJO

Se establece la duración de cada proceso no apropiativo.

(2) ALGORITMO DE FIFO

FCFS (First−come, first−served) por sus siglas en inglés. Es un algoritmo que no usa expropiación, y que consiste en atender a los procesos por estricto orden de llegada a la cola READY. Cada proceso ejecuta hasta que termina, o hasta que hace una llamada bloqueante (de I/O), o sea, ejecuta su fase de CPU completa. La gracia es que se trata de un algoritmo muy simple: la cola READY se maneja como una simple cola FIFO.

- No apropiativo
- Equitativo
- Eficiente
- El tiempo global no es aceptable.

(3) ALGORITMO DE TORNEO (ROUND ROBIN)

Es el más sencillo, equitativo y antiguo. A cada proceso se le asigna un intervalo de tiempo llamado **Quantum**, durante el cual se le permite ejecutarse. Al terminarse el tiempo y aún se ejecuta, el sistema operativo se apropia del CPU y se lo da a otro proceso.

Si el proceso termina antes de su tiempo, el cambio se CPU se hace cuando el proceso se bloquee o termine. La implementación básicamente es una lista donde al hacer un cambio de procesos, se extrae el proceso actual y se inserta al final:

B F D G A F D G A B

(Lista inicial) (Nueva)

Suspender 1 proceso e iniciar otro, requiere tiempo por almacenar datos de estado. Este tiempo se suma al Quantum lo que lo hace mayor. A Quantum de menor tiempo, más suspensiones de procesos y más tiempo del CPU desperdiciado en cambios de procesos. En Quantum mayor, se desperdicia menos el CPU en cambios de procesos, pero el rendimiento es pobre con muchos usuarios o procesos que se ejecuten por lo que no es productivo.

Por ejemplo: Un Quantum de 20 Ms. Con 5 fijados para cambio es un 20% de CPU desperdiciado, un buen tiempo medio razonable es 100 Ms. Por Quantum.

(4) ALGORITMO POR ROUND ROBIN (CON PRIORIDADES, SJN)

SJN es un caso especial de planificación por prioridad, en la cual a cada proceso se le asigna una prioridad, y la CPU se asigna al proceso con mayor prioridad en la cola READY. Se asignan prioridades a los procesos, dando más quantum a los de mayor prioridad. Para evitar que dejen en espera permanente a los otros, se va reduciendo su prioridad en la medida que el reloj los ejecuta, bajándolos a la capa inmediatamente inferior hasta terminar.

Cada capa de prioridades se ejecuta con ROUND ROBIN, pero siempre garantizando el desplazamiento de procesos a otras capas al irse ejecutando. Por ejemplo:

ROUND ROBIN

Prioridad 4 Cambios de Nivel

prioridad

Prioridad 3

Prioridad 2

Prioridad 1

(5) ALGORITMO SJF (PRIORIDAD AL PROCESO MAS CORTO)

SJF es planificación por prioridad donde la prioridad es función del estimador de la duración de la próxima fase de CPU. Una variante se hizo, haciendo que a todos los proceso se daba 1 quantum y si el proceso consumía todo su Quantum, se le duplicaba la cantidad del tiempo o sea 2 Quantums y así sucesivamente, degradándose a su vez de prioridad, pero a la larga ejecutándose en menos intercambios. **No es equitativo.**

Pero si el proceso se degradaba y se volvía interactivo con el usuario, digamos con la tecla ENTER, no era correcto dejarlo congelado, por lo que se optó por subirlo de prioridad al detectar una interacción.

De esta forma se iba atendiendo más rápidamente a estos procesos. Todo iba bien hasta que un usuario comentó que cuando presionaba ENTER todo le corría más rápido...

LAS PRIORIDADES EN ALGORITMOS

Este tema da para mucho, pues hay muchas formas de definir la prioridad. La prioridad puede definirse de manera estática o dinámica; interna o externa.

Ejemplos:

- Según categoría del usuario (externa).
- Según tipo de proceso: sistema, interactivo, o por lotes; o bien, CPU−bound o I/O bound (interna).
- Según cuanto hayan ocupado la CPU hasta el momento (dinámica).
- Para evitar que un proceso de baja prioridad sea postergado en demasía, aumentar prioridad mientras más tiempo lleve esperando: *aging* o envejecimiento (dinámica).
- Para evitar que un proceso de alta prioridad ejecute por demasiado tiempo, se puede poner un límite de tiempo, o ir bajando la prioridad.

(6) ALGORITMO AL PRIMER TRABAJO MAS CORTO

SJN (shortest−job−next) por sus siglas en inglés. Supongamos que tenemos tres procesos cuyas próximas fases de CPU son de a, b y c milisegundos de duración. Si ejecutan en ese orden, el tiempo medio de espera es:

0 + a + (a + b) = 2a+b

O sea, el primer proceso que se ejecute es el que tiene mayor incidencia en el tiempo medio, y el último, tiene incidencia nula. En conclusión, el tiempo medio se minimiza si se ejecuta siempre el proceso con la menor próxima fase de CPU que esté LISTO. Lo malo es que para que esto funcione, hay que adivinar el futuro, pues se requiere conocer la duración de la próxima fase de CPU de cada proceso. Lo que se hace es predecir la próxima fase de CPU en base al comportamiento pasado del proceso, usando un promedio exponencial. Supongamos que nuestra predicción para la n−sima fase es Tn, y que en definitiva resultó ser tn. Entonces, actualizamos nuestro estimador para predecir Tn+1:

Tn+1 = (1−a) tn + a Tn

El parámetro a, entre 0 y 1, controla el peso relativo de la última fase en relación a la historia pasada.

Tn+1 = (1−a)tn + a(1−a)tn−1 + ... + aj(1−a)tn−j+ ... + an+1T0

O sea, mientras más antigua la fase menos incidencia tiene en el estimador. Un valor atractivo para a es 1/2, ya que en ese caso sólo hay que sumar los valores y dividir por dos, operaciones especialmente fáciles en aritmética binaria. Por ejemplo, con a = ½ tenemos:

T0, T0/2 + T1/2, T0/4 + T1/4 + T2/2, T0/8 + T1/8 + T2/4 + T3/2

Después de tres nuevas ejecuciones, el peso de T0 en el nuevo estimado se ha reducido a 1/8.

(7) PLANIFICACIÓN POR LOTERÍA

La idea es dar a los procesos boletos de lotería para los diversos recursos del sistema, como el tiempo de CPU. Para tomar decisiones de planificación se escoge al azar un boleto para que obtenga el recurso, pudiendo realizar el sistema una lotería 50 veces por segundo, concediendo 20 Ms. De tiempo de CPU como premio.

Se pueden dar más boletos a los procesos más importantes, a fin de aumentar sus posibilidades de ganar. Algunas de sus propiedades interesantes son por ejemplo el que cada vez que aparezca un proceso nuevo, se le conceden boletos, con lo que ya tendrá una probabilidad de ganar proporcional al número de boletos recibidos.

En procesos cooperativos pueden intercambiar boletos entre sí como en servicios cliente−servidor donde el cliente puede prestarle temporalmente al servidor sus boletos para una mayor posibilidad de ganar tiempos.

(8) PLANIFICACIÓN DE TIEMPO REAL

Por lo regular en un sistema de **tiempo real** uno o más dispositivos físicos externos a la computadora generan estímulos y la computadora debe reaccionar a ellos de forma apropiada en un plazo fijado, como la música de los datos leídos en un CD−ROM o aplicaciones tan serias como un reactor nuclear.

El programa se divide en varios procesos y sus plazos es conocido de antemano por el planificador, generalmente de menos de 1 seg. Se asegura cumplir todos los procesos en su tiempo, pudiéndose dar los procesos **periódicos** y **no periódicos**.

La planificación puede ser estática, sabiendo de antemano los tiempos o dinámica, asignando tiempos conforme se ejecutan los procesos. Un ejemplo de los estáticos es asignar tiempos basándose en la proporción de la cantidad de veces que se ejecutará el proceso del total de eventos establecidos. A esta forma de planificar se le llama **algoritmo de tasa monotónica**.

Otro algoritmo es el de **primer plazo más próximo** que funciona evaluando el nuevo proceso contra los que quieren tiempo y les asignar prioridades de ejecución. Por ejemplo, podría evaluar un nuevo proceso interactivo contra un proceso periódico que necesita ejecutar inmediatamente.

El algoritmo de **Holgura** siempre elige al proceso con menos holgura como el caso de uno que requiera 200 Ms. y debe terminar en 250, tiene una holgura de 50 Ms., eligiendo siempre al que tenga menor holgura.

(9) PLANIFICACIÓN DE MULTIPLES COLAS

Podemos agrupar los procesos en distintas clases, y usar distintos algoritmos de planificación intra−clase, más algún algoritmo ínter−clases. Por ejemplo, los procesos interactivos y los procesos por lotes tienen distintos requerimientos en cuanto a tiempos de respuesta. Entonces, podemos planificar los proceso interactivos usando RR, y los procesos por lotes según FCFS, teniendo los primeros prioridad absoluta sobre los segundos.

Una forma de implementar este algoritmo es dividiendo la cola READY en varias colas, según la categoría del proceso. Por ejemplo, podemos tener una cola para:

- Procesos de sistema.
- Procesos interactivos.
- Procesos de los alumnos.
- Procesos por lotes.

Cada cola usa su propio algoritmo de planificación, pero se necesita un algoritmo de planificación entre las colas. Una posibilidad es prioridad absoluta con expropiación. Otra posibilidad: asignar tajadas de CPU a las colas. Por ejemplo, a la cola del sistema se le puede dar el 60% de la CPU para que haga RR, a la de procesos por lotes el 5% para que asigne a sus procesos según FCFS, y a las otras el resto.

Por otra parte, podríamos hacer que los procesos migren de una cola a otra. Por ejemplo: varias colas planificadas con RR, de prioridad decreciente y quantum creciente. La última se planifica con FCFS. Un proceso en la cola que no termina su fase de CPU dentro del quantum asignado, se pasa al final de la siguiente cola de menor prioridad, pero con mayor quantum. Un proceso en la cola que sí termina su fase de CPU dentro del quantum asignado, se pasa al final de la siguiente cola de mayor prioridad, pero con menor quantum. Ejemplo:

Cola 0: quantum=10 ms, 40% de CPU.
Cola 1: quantum=20 ms, 30% de CPU.
Cola 2: quantum=35 ms, 15% de CPU.
Cola 3: FCFS, 5% de CPU.

Así los procesos de fases más cortas tienen prioridad. Este algoritmo es uno de los más generales, pero también uno de los más complejos de implementar. También es difícil de afinar, pues hay múltiples parámetros que definir.

(10) PLANIFICACIÓN EN 2 NIVELES

Hasta ahora de alguna manera hemos supuesto que todos los procesos ejecutables están en memoria. Pero si hay poca memoria disponible y muchos procesos, entonces algunos procesos deben mantenerse en disco, lo

que cambia radicalmente la problemática de la planificación, porque el tiempo requerido para hacer un cambio de contexto que involucre traer un proceso del disco es muchísimo mayor que el tiempo de un cambio de contexto entre procesos en memoria.

Las cosas se simplifican si se divide el problema en dos, y se usa un scheduler distinto para cada caso. Un scheduler *de* **corto plazo** se encarga sólo de decidir a qué proceso se le asigna la CPU, de entre todos los que están en memoria. Periódicamente, otro scheduler *de* **largo plazo** decide qué procesos que han estado demasiado tiempo en memoria deben ser pasados a disco para dar oportunidad a procesos que han estado mucho rato en el disco. Para tomar esa decisión se pueden usar factores como el tiempo que un proceso lleva en memoria o disco, cantidad de CPU usada hasta el momento, tamaño del proceso, prioridad, etc.

Entre los criterios que el planificador superior puede usar para la toma de decisiones podrían estar:

- ¿hace cuánto el proceso se intercambió a disco?
- ¿Cuánto tiempo a recibido el proceso recientemente?
- ¿Qué tan grande es el proceso?
- ¿Qué tan alta es la prioridad del proceso?

(11) PLANIFICACIÓN EN MULTIPROCESADORES

Cuando hay varias CPUs (y una memoria común), la planificación también se hace más compleja. Podríamos asignar una cola READY a cada procesador, pero se corre el riesgo de que la carga quede desbalanceada: algunos procesadores pueden llegar a tener una cola muy larga de procesos para ejecutar, mientras otros están desocupados (con la cola vacía). Para prevenir esta situación se usa una cola común de procesos listos. Este enfoque tiene dos alternativas:

- Cada procesador es responsable de su planificación, y saca procesos de la cola READY para ejecutar. ¿El problema? Hay ineficiencias por la necesaria sincronización entre los procesadores para accesar la cola.
- Dejar que sólo uno de los procesadores planifique y decida qué procesos deben correr los demás: ***multiprocesamiento asimétrico***.

UNIDAD 4: Concurrencia

Muchos problemas se pueden resolver más fácilmente o más eficientemente si usamos procesos (o hebras) cooperativos, que ejecutan concurrentemente, técnica que se conoce como *programación concurrente*. La programación concurrente es una herramienta poderosa, pero introduce algunos problemas que no existen en la programación secuencial (no concurrente).

Concurrencia = operaciones paralelas o pseudo paralelas (intercaladas).

COMUNICACIÓN ENTRE PROCESOS

Se da cuando los procesos trabajan juntos para realizar tareas.

Formas de comunicación

Existen 2 formas de comunicación entre procesos:

- MEMORIA COMPARTIDA: Procesos comparten variables y se espera que los procesos intercambien información a través de estas variables. En este esquema el S.O. solo provee memoria, la responsabilidad de la información es de las aplicaciones.

- SISTEMA DE MENSAJES: Los procesos intercambian mensajes y la responsabilidad descansa totalmente en el S.O. pudiéndose tener 2 operaciones:
- SEND (Destino, &Mensaje)
- RECEIVE (Origen, &Mensaje)

Se pueden dar ambas formas simultáneamente. En seguida veremos la forma de memoria compartida.

MEMORIA COMPARTIDA

A veces los procesos necesitan trabajar juntos con información en común. Bajo este esquema, los procesos pueden necesitar trabajar con datos en una **memoria compartida**.

EXCLUSIÓN MUTUA

Es evitar que los procesos colisionen al querer leer o escribir en sus datos compartidos al mismo tiempo.

CONDICIONES DE COMPETENCIA (RACE−CONDITION)

Es cuando 2 o más procesos manipulan datos compartidos y el resultado final depende de estas condiciones de competencia. *Por ejemplo: El último en escribir deja sus datos.*

SECCIONES CRITICAS

Para lograr la exclusión mutua y entrar a concursar se debe evitar que dos o más procesos utilicen sus datos compartidos al mismo tiempo. La parte del programa donde accede a la información compartida se denomina **región crítica o sección crítica.** Si se puede garantiza que 2 procesos no estén en su sección crítica al mismo tiempo, entran a condiciones de concurso.

ALGORITMOS PARA LOGRAR EXLUSION MUTUA (MEMORIA COMPARTIDA)

4 CONDICIONES PARA SOLUCION ADECUADA DE DATOS COMPARTIDOS:

- Dos procesos nunca pueden estar simultáneamente dentro de sus regiones críticas.
- No puede suponerse nada acerca de las velocidades o el número de las CPU.
- Ningún proceso que se ejecute fuera de su región crítica puede bloquear a otros procesos.
- Ningún proceso deberá tener que esperar indefinidamente para entrar en su región crítica.

(1) Sistema en Lotes (Inhabilitación de Interrupciones)

La solución mas sencilla es hacer que cada proceso inhabilite las interrupciones justo después de ingresar en su región critica y vuelva a habilitarlas antes de salir de ella. De esta forma, no pueden ocurrir interrupciones de reloj. Este enfoque casi nunca resulta atractivo porque no es prudente conferir a los procesos de usuario la facultad de desactivar las interrupciones por el riesgo de no volverlas a habilitar. El Kernel si utiliza esto para algunas actualizaciones, pero en conclusión la inhabilitación de interrupciones suele ser una técnica útil dentro del sistema operativo, pero no es apropiada para los procesos del usuario.

(2) Variable de Candado (Sin variables iniciales)

La primera solución por software es tener una variable compartida, con valor inicial es 0:

PROCESO 1	PROCESO 2
WHILE TRUE {	WHILE TRUE {

WHILE Cerrado = 0; Cerrado = 1; [Sección Crítica] Cerrado = 0; }	WHILE Cerrado = 0; Cerrado = 1; [Sección Crítica] Cerrado = 0; }

- **Los 2 pueden estar en sección crítica al mismo tiempo** si llegan a evaluar el WHILE Cerrado = 0 en el mismo instante.

Una versión más formal del algoritmo anterior es:

PROCESO_1	PROCESO_2

WHILE TRUE {
 	
[Tareas Previas 1]
 	
WHILE Proceso_2_Adentro;
 	
Proceso_1_Adentro = True;
 	
[Sección Crítica]
 	
Proceso_1_Adentro = False;
 	
[Tareas no Críticas 1]
 	
}
	WHILE TRUE {
	
	[Tareas Previas 2]
	
	WHILE Proceso_1_Adentro;
	
	Proceso_2_Adentro = True;
	
	[Sección Crítica]
	
	Proceso_2_Adentro = False;
	
	[Tareas no Críticas 2]
	
	}

- **La exclusión mutua no se garantiza** (si los 2 llegan están al mismo tiempo en el WHILE de espera, los 2 detectan pasan porque no hay ninguno adentro y los 2 ejecutan sus secciones críticas).
- Utiliza 2 variables.

(3) Variable de Candado (Con Variables iniciales)

La primera solución por software es tener una variable compartida, con valor inicial es 0:

PROCESO 1	PROCESO 2
WHILE TRUE { WHILE Cerrado <> 0; [Sección Crítica] Cerrado = 1; [Sección no Crítica]	WHILE TRUE { WHILE Cerrado <> 1; [Sección Crítica] Cerrado = 0; [Sección no Crítica]

}	}

- **Se evita que ambos entren a sección crítica**, pero
- **Uno bloquea a otro** en su sección no crítica cuando hay un proceso mucho más lento que otro.

El algoritmo anterior también se puede escribir de la siguiente manera:

BEGIN

Proceso_1

ParBegin

Proceso_1

Proceso_2

ParEnd

END

PROCESO_1	PROCESO_2

WHILE TRUE {
 	
[Tareas Previas 1]
 	
WHILE Proceso_2;
 	
[Sección Crítica]
 	
Proceso_2;
 	
[Tareas no Críticas 1]
 	
}
	WHILE TRUE {
	
	[Tareas Previas 2]
	
	WHILE Proceso_1;
	
	[Sección Crítica]
	
	Proceso_1;
	
	[Tareas no Críticas 2]
	
	}

- Utiliza la espera ocupada.
- **Orden constantemente alterno.**
- Si uno de los 2 procesos termina, el otro no podrá continuar.
- Problema en sincronizar alternancia (si **uno es más rápido** puede seguir entrando y dejar al otro esperando).

(4) Variable de Candado (Con Variables iniciales de desear entrar)

BEGIN

P1_DeseaEntrar = False

P2_DeseaEntrar = False

ParBegin

Proceso_1

Proceso_2

ParEnd

END

PROCESO_1	PROCESO_2

WHILE TRUE {
 	
[Tareas Previas 1]
 	
P1_DeseaEntrar = True
 	
WHILE P2_DeseaEntrar;
 	
[Sección Crítica]
 	
P1_DeseaEntrar = False;
 	
[Tareas no Críticas 1]
 	
}
	WHILE TRUE {
	
	[Tareas Previas 2]
	
	P2_DeseaEntrar = True
	
	WHILE P1_DeseaEntrar;
	
	[Sección Crítica]
	
	P2_DeseaEntrar = False;
	
	[Tareas no Críticas 2]
	
	}

- **Ínter bloqueo**, aplazamiento indefinido. Cuando los 2 procesos cambian sus variables de querer entrar al mismo tiempo y se pasa a espera ocupada.
- Garantiza exclusión mutua.

Una tentativa para resolver el ínter bloqueo es tratar de desbloquear a alguno de los 2 procesos con retardos aleatorios:

BEGIN

P1_DeseaEntrar = False

P2_DeseaEntrar = False

ParBegin

Proceso_1

Proceso_2

ParEnd

END

PROCESO_1	PROCESO_2
WHILE TRUE { [Tareas Previas 1]	WHILE TRUE { [Tareas Previas 2]

P1_DeseaEntrar = True	P2_DeseaEntrar = True
WHILE P2_DeseaEntrar	WHILE P1_DeseaEntrar
P1_DeseaEntrar = False;	P2_DeseaEntrar = False;
Retraso (Aleatorio, Algunos ciclos)	Retraso (Aleatorio, Algunos ciclos)
P1_DeseaEntrar = True;	P2_DeseaEntrar = True;
ENDWHILE;	ENDWHILE;
[Sección Crítica 1]	[Sección Crítica 2]
P1_DeseaEntrar = False;	P2_DeseaEntrar = False;
[Tareas no Críticas 1]	[Tareas no Críticas 2]
}	}

- **Garantiza exclusión mutua y no hay ínter bloqueos** por los diferentes retardos.
- **Aplazamiento indefinido** en la ejecución de 1 o más procesos.

(5) ALGORITMO DE DECKER

Program Exclusión_mutua;

Var Proceso_favorecido :(Primero, Segundo);

P1_DeseaEntrar, P2_DeseaEntrar : Boolean;

Procedure Proceso_Uno;

While TRUE do

Begin

[Tareas Previas 1]

P1_DeseaEntrar = True;

While P2_DeseaEntrar do

If Proceso_Favorecido = Segundo Then

Begin

P1_DeseaEntrar = False;

While Proceso_Favorecido = Segundo do;

P1_DeseaEntrar = True;

End;

[Sección Crítica 1]

Proceso_Favorecido = Segundo;

P1_DeseaEntrar = False;

[Tareas no críticas 1]

End;

Procedure Proceso_Dos;

While TRUE do

Begin

[Tareas Previas 2]

P2_DeseaEntrar = True;

While P1_DeseaEntrar do

If Proceso_Favorecido = Primero Then

Begin

P2_DeseaEntrar = False;

While Proceso_Favorecido = Primero do;

P2_DeseaEntrar = True;

End;

[Sección Crítica 2]

Proceso_Favorecido = Primero;

P2_DeseaEntrar = False;

[Tareas no críticas 2]

End;

BEGIN

P1_DeseaEntrar = False;

P2_DeseaEntrar = False;

Proceso_favorecido = Primero;

ParBegin

Proceso_Uno;

Proceso_Dos;

ParEnd

END

- Resuelve exclusión mutua.
- Utiliza variable de turno con variables de intención de entrada para resolver casos de conflicto.
- No se produce aplazamiento indefinido.

(6) ALGORITMO DE PETERSON

Program Exclusión_mutua;

Var Proceso_favorecido :(Primero, Segundo);

P1_DeseaEntrar, P2_DeseaEntrar : Boolean;

Procedure Proceso_Uno;

While TRUE do

Begin

[Tareas Previas 1]

P1_DeseaEntrar = True;

Proceso_Favorecido = Primero;

While P2_DeseaEntrar AND Proceso_Favorecido = Segundo do;

[Sección Crítica 1]

P1_DeseaEntrar = False;

[Tareas no críticas 1]

End;

Procedure Proceso_Dos;

While TRUE do

Begin

[Tareas Previas 2]

P2_DeseaEntrar = True;

Proceso_Favorecido = Segundo;

While P1_DeseaEntrar AND Proceso_Favorecido = Primero do;

[Sección Crítica 2]

P2_DeseaEntrar = False;

[Tareas no críticas 2]

End;

BEGIN

P1_DeseaEntrar = False;

P2_DeseaEntrar = False;

Proceso_favorecido = Primero;

ParBegin

Proceso_Uno;

Proceso_Dos;

ParEnd

END

- Resuelve exclusión mutua y los ínter bloqueos.
- No se produce aplazamiento indefinido.

(7) Algoritmo Productor Consumidor

Dos procesos comparten un mismo BUFFER de espacio limitado. El consumidor coloca información y el otro la saca, pudiéndose generalizar a N productores y consumidores.

Cuando el productor quiere agregar un elemento, pero el BUFFER esta lleno, debe dormir hasta que el consumidor saque un elemento y entonces puede seguir agregando elementos. Igualmente si el consumidor no tiene elementos que extraer del BUFFER, entonces duerme hasta que el productor coloque elementos.

#define N 100

int count = 0;

void producer(void)

```
{
while (true) {
produce_intem();
if (count == N) sleep();
enter_item();
count = count + 1;
if (count == 1) wakeup(consumer);
}
}
void consumer(void)
{
while (true) {
if (count == 0) sleep();
remove_item();
count = count − 1;
if (count == N − 1) wakeup(producer);
consume_item();
}
}
```

Se da un problema porque por ejemplo, al estar vacío el BUFFER, el consumidor esta evaluando su contador y ve que no hay datos, pero antes de dormirse el CPU pasa a ejecutar al productor, este pone un dato y despierta al consumidor, sin embargo como el Consumidor no llego a dormirse al reanudarse se duerme y el productor continua produciendo. Tarde o temprano el productor habrá llenado el BUFFER. Un arreglo es agregar un bit de espera donde si un proceso quiere despertar a uno que nunca se durmió, este bit evita que caiga de nuevo al Sleep(), pero esta solución se vuelve difícil de implementar con varios procesos y muchos bits de espera.

Semáforos

Los semáforos vinieron a solucionar el problema de los productores y consumidores de varios procesos con un bit de espera para evitar problemas con el BUFFER compartido. La solución la propuso E.W. Dijkstra (1965) introduciendo la variable de un semáforo con dos funciones manejadas atómicamente.

Las operaciones atómicas son:

P(S): WHILE S <= 0;

S = S − 1;

V(S): S = S + 1;

Cuando el valor del semáforo al dormir P(S) es menor o igual que 0, se queda en P(S) sin completar el dormir. Cuando V(S) se ejecuta, uno de los que estaban en P(S) elegido al azar puede despertar para terminar P(S), dejando a los otros dormidos. Esto funciona siempre que se garantice la atomicidad de operaciones. La variable del semáforo, aquí S, y normalmente conocida como MUTEX se inicia siempre con valor 1 para que el primer proceso pare a los demás. De forma general, la implementación para procesos es:

Aplicados a Productor Consumidor

Aplicando semáforos a los productores y consumidores tenemos el siguiente código (P(S) = down(S), V(S) = up(S)):

#define N 100 /* Tamaño BUFFER.

TYPEDEF int semaphore; /* Los semáforos son un tipo especial de int.

semaphore mutex = 1; /* Controla acceso a región crítica.

semaphore empty = N; /* Cuenta las ranuras vacías

semaphore full = 0; /* Cuenta las ranuras llenas.

Void producer(void)

{

int item;

while (TRUE) {

produce_item(&item); /* Generar algo.

down(&empty); /* Decrementa ranuras vacías.

down(&mutex); /* Entrar en sección crítica

enter_item(item); /* Inserta nuevo elemento.

up(&mutex); /* Sale sección crítica.

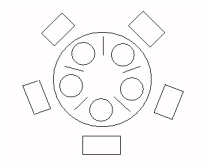
up(&full); /* Incrementa ranuras llenas.

}
}

```
Void consumer(void)

{

int item;

while (TRUE) {

down(&full); /* Decrementa ranuras llenas.

down(&mutex); /* Entra sección crítica.

remove_item(&item); /* Saca elemento.

up(&mutex); /* Sale de sección crítica.

up(&empty); /* Incrementa ranuras vacías.

consume_item(item); /* Hacer algo con elemento consumido.

}
}
```

Problema de Filósofos con semáforos

Cinco filósofos pasan la vida alternando entre comer y pensar, alrededor de una mesa redonda. Como son medio torpes, requieren dos tenedores para comer, pero como son pobres, sólo tienen 5 tenedores. Han acordado que cada uno usará sólo los tenedores a su izq. y a su derecha. Entonces, de los 5 filósofos, sólo 2 pueden comer al mismo tiempo, y no pueden comer dos que son vecinos.



Una posibilidad: representar cada tenedor con un semáforo inicializado en 1 (Semaforo tenedor[5];), y cada filósofo *i* hace:

```
while (TRUE) {

  P(Tenedor[i]);

  P(Tenedor[i+1 % 5]);

  comer();

  P(Tenedor[i]);

  P(Tenedor[i+1 % 5]);

  pensar();
```

}

Problema: bloqueo mutuo. Soluciones posibles:

Usar solución asimétrica: uno de los filósofos toma primero el tenedor i+1%5 y después el i.

Permitir que un filósofo tome sus tenedores sólo si ambos están disponibles (y hacerlo dentro de una sección crítica).

Permitir que sólo cuatro filósofos se sienten en la mesa.

Estas soluciones evitan bloqueo mutuo, pero también es conveniente asegurar que ningún filósofo muera de hambre. Si un proceso es postergado indefinidamente se dice que sufre de inanición o *starvation* (aún cuando no haya comida de por medio).

Sistemas de Mensajes

Básicamente se puede dar de 2 formas:

- COMUNICACION DIRECTA: Cada proceso que envía o recibe, debe nombra explícitamente al otro proceso en SEND y RECIEVE, teniendo las propiedades:
- 1 enlace se establece automáticamente entre cada par de procesos que quieren comunicarse.
- Un enlace es asociado exactamente a 2 procesos.
- El enlace es bidireccional.

La desventaja de este esquema es que al cambiar el nombre de un proceso, requiere examinar la definición de cualquier mensaje, pues las referencias a nombres antiguos deben ser cambiadas al nuevo nombre.

- COMUNICACION INDIRECTA: Los mensajes son enviados y recibidos desde apartados postales (MAILBOXES) también conocidos como puertos. Cada apartado tiene un número de identificación único, siendo las comunicaciones de la forma: SEND (A, &mensaje) y RECIEVE (A, &mensaje). Se tienen las siguientes propiedades:
- Un enlace entre procesos se establece si tienen un apartado compartido.
- Un enlace puede asociarse con más de 2 procesos.
- El enlace puede ser unidireccional o bidireccional.

Concurrencia en apartados

Si es un sistema de apartados, puede darse concurrencia al tener 3 procesos asociados a 1 mismo apartado. Por ejemplo, al escribir un proceso y los otros 2 leen, ¿a quien da el mensaje?. Todo dependerá de:

- Permitir que un enlace sea para 2 procesos como máximo.
- Permitir que solo un proceso ejecute RECIEVE en un momento dado. (con semáforos).
- Permitir que el sistema decida arbitrariamente.

La sincronización de procesos es muy importante para evitar pérdidas. Con varias computadoras el riesgo es mayor de pérdidas y cuando un proceso termina o es cancelado, el sistema debe notificar la terminación y tomar en cuenta los mensajes pendientes de enviar o recibir.

Tipos de apartados

Un apartados puede pertenecer a un (1) proceso o al (2) sistema. Si es del proceso, el propietario es quien

recibe mensajes y el usuario (quien envía mensajes a través de este apartado). Cuando el proceso propietario termina, desaparece el apartado.

Los apartados del sistema operativo tienen independencia de cualquier proceso y el sistema puede crearlos, enviar y recibir mensajes, y destruirlos cuando lo desee.

BUFFERING para mensajes

Las colas de mensajes en un enlace dado pueden ser:

- CAPACIDAD 0: El enlace no puede tener mensajes en espera, el enviador tiene que esperar hasta que el receptor reciba el mensaje. Ambos procesos deben sincronizarse para que se haga la transferencia del mensaje correctamente.
- CAPACIDAD LIMITADA: La cola tiene capacidad de al menos N mensajes en el enlace. Si la cola no esta llena el enviador puede colocar su mensaje en cola y esperar. Si la cola esta llena, el enviador debe esperar hasta que haya espacio.
- CAPACIDAD ILIMITADA: La cola tiene capacidad ilimitada y el enviador nunca tiene que esperar.

Tipos de Mensajes

- Tamaño fijo
- Tamaño Variable
- Typed Message (Mensaje con un tipo asociado)

UNIDAD 5: Ínter bloqueo (DeadLocks)

CASOS DE INTERBLOQUEOS

Cuando tenemos muchos procesos que compiten por recursos finitos, puede darse una situación en la que un proceso está bloqueado esperando por un recurso que nunca se liberará, porque lo posee otro proceso también bloqueado.

Ley de principios de siglo, en Kansas: "cuando dos trenes se aproximan a un cruce, ambos deben detenerse completamente, y ninguno podrá continuar hasta que el otro se haya ido."

CONDICIONES

Para que haya deadlock, deben darse simultáneamente las siguientes condiciones:

- **Exclusión mutua.** Los recursos no son compartidos. Si un proceso está usando un recurso, otros no pueden hacerlo.
- **Retención y espera.** Hay procesos que tienen recursos asignados y al mismo tiempo están esperando poder adquirir otros recursos.
- **No expropiación.** Los recursos no pueden ser expropiados a los procesos. (Ejemplo: impresora).
- **Espera circular.** Existe un conjunto *{P0, P1, ..., Pn}* de procesos tales que el proceso *Pi* está esperando por un recurso retenido por *Pi+1* para *0 <= i < n*, y *Pn* esté esperando recurso retenido por *P0*.

Una forma de modelar estas condiciones es usando un *grafo de recursos*: círculos representan procesos, los cuadrados recursos. Una arista desde un recurso a un proceso indica que el recurso ha sido asignado al proceso. Una arista desde un proceso a un recurso indica que el proceso ha solicitado el recurso, y está bloqueado esperándolo. Entonces, si hacemos el grafo con todos lo procesos y todos los recursos del sistema y encontramos un ciclo, los procesos en el ciclo están bajo bloqueo mutuo.

ESTRATEGIAS PARA RESOLVERLOS

(1) DEL AVESTRUZ (No hacer nada)

Los deadlocks son evidentemente indeseables. Los recursos retenidos por los procesos bajo bloqueo mutuo no están disponibles para otros procesos, y los procesos en deadlock no pueden avanzar. Pero si un deadlock se produce, en promedio, cada diez años y en cambio el sistema se cae todos los días por fallas en el hardware o en el sistema operativo, entonces el problema de los bloqueos mutuos es irrelevante. Muchos sistema operativo modernos (UNIX entre ellos) no se preocupan de evitar deadlocks (porque es caro y los procesos deben someterse a ciertas restricciones), pero algunas tendencias hacen que el tema vaya adquiriendo importancia: a medida que progresa la tecnología, tiende a haber más recursos y más procesos en un sistema.

(2) Detección y recuperación

Una posibilidad es monitorear cada cierto tiempo el estado del grafo de recursos. (¿Cada cuánto?) Si se detecta un ciclo, se matan todos los procesos del ciclo (¡se usa!), o se van matando procesos del ciclo hasta que no queden ciclos (¿cuál matar primero?). Es un método drástico, pero mejor que nada.

(3) Prevención

Una tercera estrategia es imponer restricciones a los procesos de manera de hacer estructuralmente imposible la ocurrencia de un bloqueo mutuo. La idea es asegurar que no se dé al menos una de las cuatro condiciones necesarias para que haya deadlock.

1. Exclusión mutua.

Hay recursos que son intrínsecamente no compartibles, de modo que no se puede descartar la exclusión mutua.

2. No expropiación.

Esta condición se puede eliminar imponiendo expropiación. Si un proceso *P* tiene recursos y solicita otro que está ocupado, se le pueden expropiar a *P* los que ya tiene, o bien expropiarle al otro proceso el recurso que *P* necesita. Es aplicable a cierta clase de recursos (cuyo estado se puede almacenar y luego recuperar), pero no a otros como registros de base de datos o impresoras.

Retención y espera.

Podemos impedir esta condición exigiendo que los procesos soliciten de una vez, al comienzo, todos los recursos que van a necesitar. Si uno o más recursos no están disponibles, el proceso se bloquea hasta que pueda obtenerlos todos. Inconvenientes: muchos procesos no pueden saber de antemano qué y cuántos recursos necesitarán. Subutilización de recursos, ya que quedan retenidos de principio a fin de la ejecución.

Alternativa: Hacer que los procesos liberen todos los recursos que tienen antes de solicitar un nuevo conjunto de recursos. También tiene inconvenientes: por ejemplo, programa que usa archivo temporal durante toda su ejecución. Si lo libera entremedio, otro proceso se lo puede borrar.

(4) Espera circular

Podemos evitar la espera circular si imponemos un orden total a los recursos (o sea, asignamos a cada recurso *R* un número único *F(R)*), y obligamos a los procesos a que soliciten recursos en orden: un proceso no puede solicitar *Q* y después *R* si *F(Q)>F(R)*. Por ejemplo:

F(CD−ROM)=1

F(impresora)=2

F(plotter)=3

F(Cinta)=4

De esta manera se garantiza que no se generarán ciclos en el grafo de recursos. Una mejora inmediata es exigir solamente que ningún proceso solicite un recurso cuyo número es inferior a los recursos que ya tiene. Pero tampoco es la panacea. En general, el número potencial de recursos es tan alto que es difícil dar con tal función *F* para ordenarlos.

(5) Evitación

En vez de restringir la forma u orden en que los procesos deben solicitar recursos, antes de conceder un recurso, chequeamos que sea seguro.

Hay diversos algoritmos, que varían en el tipo de información que requieren a priori de cada proceso. En el que vamos a ver, necesitamos que cada proceso declare la cantidad máxima de recursos de cada clase que va a necesitar: Por ejemplo: 2 unidades de cinta, una impresora láser y 200 bloques de disco, como *máximo*. (El sistema puede tener varias unidades de cinta y varias impresoras láser idénticas).

(6) Estado seguro

Un *estado de asignación de recursos* es el número de recursos disponibles y asignados, y el máximo declarado por cada proceso. Ejemplo: Sistema con 12 unidades de cinta y 3 procesos.

	Máximo	Actual	Diferencia	Disponible
P	10	5	5	3
Q	4	2	2	
R	9	2	7	

Un *estado seguro* es un estado en el cual el sistema puede asignar recursos a los procesos (hasta su máximo) en alguna secuencia, y evitar deadlock. Más formalmente, un estado es seguro sólo si existe una *secuencia segura*, es decir, una secuencia de procesos $<P1, P2,...,Pn>$ donde, para cada *Pi*, los recursos que *Pi* aún puede solicitar pueden satisfacerse con los recursos disponibles y los retenidos por *Pj* con $j<i$. Si los recursos que *Pi* necesita no están disponibles, *Pi* puede esperar hasta que los *Pj* terminen. Si no existe tal secuencia, se dice que el sistema está en un estado *inseguro*.

El estado del ejemplo es seguro, ya que la secuencia $<Q,P,R>$ satisface la condición de seguridad. Si en ese estado se le concede una unidad más a *R*, entonces pasamos a un estado inseguro. Un estado inseguro no necesariamente implica que se va a producir deadlock, pero se corre el riesgo. En cambio, mientras el sistema esté en un estado seguro, no puede haber deadlock. La clave entonces es no pasar a un estado inseguro; en el ejemplo, no se le puede conceder otra unidad a *R*; si la solicita, hay que suspenderlo, aunque el recurso esté disponible. Esto significa que habrá una subutilización de los recursos que no existiría si no se hiciera nada para manejar los bloqueos mutuos.

(7) Algoritmo del banquero

Un ejemplo sencillo:

Tengo 12 impresoras.

Préstamo actual Necesidad Máxima

P1 5 10

P2 3 5

P3 2 6

−−−−−−−−−−−−−−

10 utilizadas

Tengo 2 libres.

A P2 le faltan 2 −> Finalizo P2.

Tengo 5 libres.

A P1 le faltan 5 −> Finalizo P1.

Tengo 10 libres.

A P3 le faltan 4 −> Finalizo P3.

Sistematizando y generalizando para múltiples recursos, obtenemos el *algoritmo del banquero* (se supone que los procesos son clientes que piden crédito, y los recursos corresponden al dinero del que dispone el banco). La idea es básicamente la misma, sólo que debemos manejar vectores en vez de escalares. Por ejemplo, para 5 procesos y 3 recursos, **Máximo**, **Actual** y **Diferencia**, que antes eran columnas, ahora son matrices de 5x3; y **Disponible**, que antes era un escalar, ahora es un vector de 3 elementos.

Máximo	Actual	Diferencia