Índice General

Capítulo 1

Introducción a los Sistemas Operativos

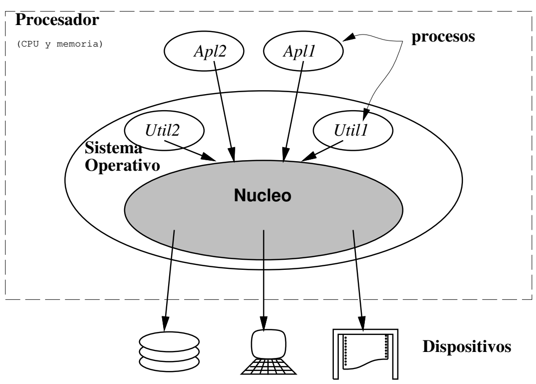

Figura 1.1: Componentes de un sistema computacional.

Un *Sistema Operativo* es un *programa* que actúa como intermediario entre el usuario y la máquina. El propósito de un sistema operativo es proveer un ambiente en que el usuario puede ejecutar sus *aplicaciones*. Las aplicaciones son todos aquellos programas que el usuario ejecuta para mejorar su productividad o para divertirse.

El primer objetivo de un sistema operativo es que el computador sea *cómodo de usar*. El segundo objetivo es que la máquina sea usada *eficientemente*.

Las componentes de un sistema operativo se observan en la figura 1.1. La principal componente del sistema operativo es el *núcleo* (*kernel*). El núcleo se encarga de ejecutar y dar servicios a los *procesos*.

Un proceso puede ser una aplicación o un *utilitario*. Las aplicaciones son los programas del usuario que le ayudan a mejorar su productividad. Las aplicaciones no son parte del sistema operativo. Los utilitarios son programas que pertenecen al sistema operativo y que ayudan al usuario a : administrar sus archivos, imprimir programas y resultados, interactuar con la máquina a través de una interfaz gráfica o un intérprete de comandos, etc.

Lo procesos interactúan necesariamente con el núcleo para crear otros procesos, comunicarse entre sí y obtener memoria para sus datos. Usualmente (aunque no necesariamente) los procesos también interactuán con el núcleo para manejar archivos. El usuario nunca interactúa directamente con el núcleo. El usuario interactúa con los procesos.

Por razones de eficiencia y simplicidad de implementación el núcleo es la componente del sistema operativo que está siempre residente en memoria. En cambio las aplicaciones y los utilitarios se cargan cuando se necesitan, y por lo tanto no siempre están residentes en la memoria

En este curso se estudiará en profundidad el diseño del núcleo de los sistemas operativos. Además el curso contempla actividades prácticas en donde se modificarán componentes del pseudo sistema operativo nSystem.

Programa

I. Introducción

   Historia del desarrollo de los sistemas operativos : sistemas batch, job, dump, monitor residente, operación off-line, buffering, spooling, multi-programación, job scheduling, máquinas virtuales, procesos, computadores personales, redes, sistemas distribuidos.

II. Procesos

   – Procesos pesados vs. procesos livianos o threads.
   – Preemption vs. non-preemption.
   – Un sistema de procesos livianos : nSystem.

- El problema de la sincronización de procesos: productor/consumidor, filósofos comiendo, lectores/escritores.
- Sincronización entre procesos mediante mensajes: comunicación síncrona y asíncrona.
- Otros mecanismos de sincronización entre procesos: semáforos, regiones críticas, monitores.

III. Estructura del Computador

- Arquitectura lógica del Hardware: espacio de direcciones reales, E/S mapeada en el espacio de direcciones reales, interrupciones, polling, vector de interrupciones, canales de E/S, modo dual, seguridad y protección, espacio de direcciones virtuales, segmentación, timer.
- Estructura del Software: núcleo, drivers, API, servicios del núcleo/sistema operativo.

IV. Administración de Procesos.

Scheduling de procesos, indentificador de proceso, descriptor de proceso, colas de scheduling, estados de un proceso, cambio de contexto, ráfagas de CPU, estrategias de scheduling, First Come First Served, Shortest Job First, Colas de Prioridad, Round Robin, scheduling en varios niveles, scheduling en nSystem.

V. Administración de Memoria Primaria.

Segmentación, Paginamiento, Memoria Virtual, Swapping, Demand Paging, page fault, localidad de los accesos a memoria, estrategias de reemplazo de páginas, First Come First Served, Least Recently Used, la estrategia del reloj, la estrategia del working set, copy-on-write, demand loading, paginamiento en x86, Translation Lookaside Buffer.

VI. Administración de Memoria Secundaria.

El sistema de archivos de unix, scheduling de disco.

VII. Estructura de Redes

Topología de redes, localización de hosts, estrategias de conexión, estrategias de ruteo, protocolos de comunicación de datos.

Evaluación:

- 2 controles + examen (70%).
- 2 o 3 tareas computacionales (no habrá alumnos pendientes).

Bibliografía:

A. Silberschatz, “Operating System Concepts”, 1991 Addison-Wesley

1.1 Evolución de los S.O.

Al comienzo sólo había Hardware y aplicaciones. Un programa en ROM lee el binario de una aplicación a partir de algún dispositivo.

La aplicación debe incluir todo, incluso debe saber como manejar cada dispositivo. Es decir no hay `read` ni `write`, sino que la aplicación debe comandar directamente el disco a través de puertas de E/S en el espacio de direcciones. El disco se ve como una secuencia finita de bloques de bytes sin ninguna estructura en archivos y directorios. La aplicación es responsable de dar alguna estructura al disco.

Si se desea lanzar otra aplicación hay que presionar el botón *reset* en la consola del computador.

Durante el desarrollo de una aplicación la depuración se lleva a cabo observando las luces de la consola del computador. Dado que sólo un programador puede usar el computador en un instante, los programadores deben reservar hora.

Problema: Bajo provecho de un computador costoso

- El computador pasa la mayor parte del tiempo ocioso, esperando instrucciones del programador.
- El tiempo de computador es mucho más caro que los programadores.

¿Cómo mejorar el rendimiento del computador?

1.1.1 Sistemas Batch

En un sistema batch el procesamiento se hace en lotes de tarjetas perforadas o jobs. El programador no interactúa directamente con el computador. Un job es un lote de tarjetas perforadas por el programador mediante máquinas perforadoras. Estas tarjetas contienen el programa y los datos delimitados por tarjetas de control.

Un *operador* del computador se encarga de recibir los jobs de varios programadores e introducirlos en una lectora de tarjetas para que sean procesados.

Un programa denominado *monitor residente* se encarga de leer las tarjetas, interpretar las tarjetas de control y lanzar los programas. Si el programa termina en un error, el monitor residente imprime un *dump*, que consiste en una imagen hexadecimal de la memoria del computador al momento de ocurrir el error. El operador entrega el dump al programador para que éste determine la causa del error y solucione el problema rehaciendo algunas de las tarjetas.

Ventajas

- Mientras el programador piensa, el computador puede procesar otros jobs.
- El computador pasa casi el 100% del tiempo ocupado, puesto que el operador es mucho mas diestro en el manejo de las tarjetas que los programadores.

Problema : Baja utilización de un procesador costoso

Aunque el computador está ocupado, su componente más costosa, el procesador o CPU pasa gran parte del tiempo ocioso porque la lectura de las tarjetas y la impresión de los resultados son muy lentas.

1.1.2 Operación Off-line

El término off-line significa fuera de la línea de producción. Aplicado a los sistemas batch, operación off-line significa que la lectura de las tarjetas y la impresión de los resultados no las realiza el computador. Estas actividades se realizan a cabo en procesadores satélites de bajo costo y especializados en una sola actividad (ver figura 1.2).

Las tarjetas de varios jobs son leídas por un procesador satélite, el que genera una cinta (parte A en la figura) que el operador transporta hacia la unidad de cintas del computador principal. El computador lee estas cintas mucho más rápidamente que las tarjetas (parte B en la figura).

De igual forma, el resultado de los jobs se graba en una o más cintas que son desmontadas por el operador y llevadas a un procesador satélite dedicado a la impresión (parte C en la figura).

Ventajas

Como la lectura de cintas es mucho más rapida que la de tarjetas, se puede lograr un mejor rendimiento

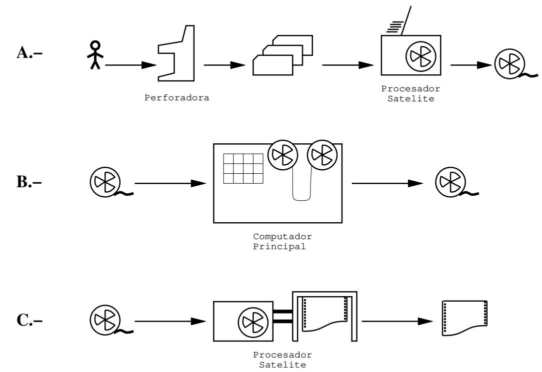

Figura 1.2: Operación Off-Line de periféricos

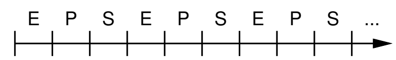

Figura 1.3: Alternancia entre lectura, proceso y escritura de un job típico.

del procesador principal asociando varios satélites de lectura y varios satélites de impresión.

En efecto, el costo de los procesadores satélites es marginal frente al costo del procesador principal. Cada vez que se agrega un procesador satélite se aumenta la capacidad de proceso sin comprar un nuevo procesador principal, sólo se aumenta el porcentaje de tiempo de utilización del procesador principal. Por supuesto esto tiene un límite, puesto que llega un momento en que el computador principal no es capaz de leer más cintas. Por lo tanto se agregan procesadores satélites mientras exista capacidad ociosa en el computador principal.

Las técnicas que veremos a continuación apuntan hacia ese mismo objetivo : aumentar el porcentaje de tiempo de utilización del procesador principal agregando otros procesadores dedicados de bajo costo.

Problema : Busy-waiting

Mientras se lee o escribe una línea (equivalente a una tarjeta), la CPU debe hacer *busy-waiting*, es decir no realiza ningún trabajo útil. En la figura 1.3 se observa el típico job que alterna entre lectura de una línea de datos, procesamiento de esa línea y escritura del resultado. En ella se observa que durante las fases E y S el procesador está ocioso.

1.1.3 Buffering

Normalmente se entiende por buffering una técnica que consiste en leer/escribir bloques de varias líneas

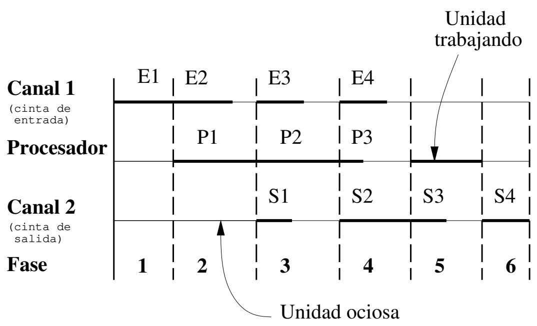

Figura 1.4: Uso de canales para leer, grabar y procesar en paralelo

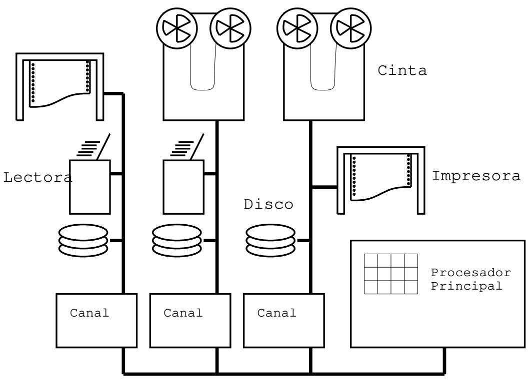

Figura 1.5: Utilización de canales para la conexión de dispositivos

en una sola operación de entrada/salida. Esto disminuye el tiempo total requerido para la entrada/salida, ya que leer/escribir 10 líneas toma en la práctica casi el mismo tiempo que leer/escribir una línea.

Por ejemplo si en el esquema original el tiempo de proceso estaba dominado por 10000 lecturas de una línea cada una, con buffering factor 10 el tiempo se reduciría drásticamente ya que se realizarían sólo 1000 operaciones de lectura de a 10 líneas que tomarían poco más de un décimo del tiempo sin buffering.

La técnica de buffering es un poco más compleja que lo descrito anteriormente, puesto que también se usan canales para realizar las operaciones de lectura/escritura de bloques en paralelo con el proceso del job. Los canales son procesadores especializados en el movimiento de datos entre memoria y dispositivos. Son ellos los que interactúan con las cintas con mínima intervención de la CPU. La CPU sólo necesita intervenir al final de cada operación de E/S.

La figura 1.4 muestra la técnica de buffering. En ella se observa que en la fase i se superponen las siguientes actividades:

- Lectura del bloque E_{i+1}.
- Proceso del bloque que tiene como entrada E_i y salida S_i.
- Escritura del bloque de salida S_{i-1}

Observe que la fase $i+1$ sólo puede comenzar cuando todas las partes de la fase i han concluido.

El rendimiento de esta técnica depende de la naturaleza del job que se procesa.

- Jobs que usan intensivamente la CPU: En este tipo de jobs el tiempo de proceso de un bloque de datos es muy superior al tiempo de lectura y escritura de un bloque. En este tipo de jobs se alcanza un 100% de utilización de la CPU.
- Jobs intensivos en E/S: En este tipo de jobs el tiempo de proceso de un bloque es ínfimo y por lo tanto muy inferior a su tiempo de lectura/escritura. En este tipo de jobs el tiempo de utilización de la CPU puede ser muy bajo.

Ventantas

Con la técnica de buffering se logra un mejor rendimiento del procesador, ya que ahora aumenta su porcentaje de utilización. El aumento del costo es marginal, ya que los canales son mucho mas económicos que el procesador.

Problema: Tiempo de despacho prolongado

El inconveniente con los sistemas off-line es que aumenta el tiempo mínimo transcurrido desde la entrega del job en ventanilla hasta la obtención de los resultados (tiempo de despacho). Este aumento se debe a la demora que significa todo el transporte de cintas entre procesadores satélites y procesador principal. Esta demora se paga aun cuando la cola de jobs esperando proceso es pequeña.

1.1.4 Simultaneous Peripheral Operation On-Line: Spooling

El término On-Line significa que la lectura de tarjetas e impresión de resultados ya no se realizan en procesadores satélites. Las lectoras e impresoras se conectan directamente al computador aprovechando los canales de E/S que éste posee (ver figura 1.5). La aparición de los sistemas On-Line ocurre gracias a la aparición de discos de costo relativamente bajo.

Al igual que los sistemas Off-Line, los sistemas On-Line también permiten lograr un mejor grado de utilización del procesador principal. La idea es que un

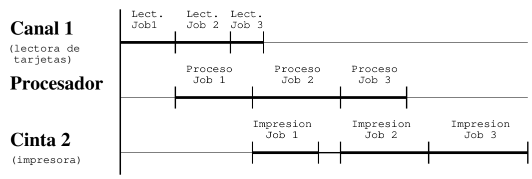

Figura 1.6: Spooling

job nunca lee sus datos directamente de la lectora de tarjetas o imprime directamente sus resultados. Un job obtiene su entrada de un archivo en disco y graba su salida en un archivo también en disco. Es el monitor residente el que se encarga de leer las tarjetas (dejándolas en el disco) e imprimir los resultados. Para mejorar el grado de utilización del procesador principal, el monitor residente superpone las siguientes actividades :

- Procesamiento del job que corresponde procesar,
- Lectura de las tarjetas de uno o más jobs que esperan ser procesados.
- Impresión de los resultados de uno o más jobs que ya fueron procesados.

Este esquema se muestra en figura 1.6. En ella se observa que la única restricción para procesar el i-ésimo job es que éste haya sido leído por algún canal. Del mismo modo, la única restricción para imprimir el resultado de un job es que ya se haya concluido su proceso.

Ventantas

Al desaparecer los procesadores satélites (off-line) se disminuye la labor del operador en lo que respecta a transportar cintas. Esto disminuye el tiempo de despacho de un job, especialmente cuando la cola de jobs esperando proceso se mantiene pequeña.

Problema : Aún queda tiempo de procesador disponible

Los discos –aunque más rápidos que las cintas– siguen siendo más lentos que la CPU. En procesos intensivos en E/S, la utilización de la CPU sigue siendo baja. Por lo tanto el próximo adelanto sigue apuntando a ocupar esa capacidad ociosa evitando la compra de una nueva CPU.

1.1.5 Sistemas de Multiprogramación

En un sistema On-Line en un instante dado pueden haber varios procesos listos para ejecutarse (en el Spool). En un sistema de multiprogramación (fines de los '60) el monitor residente aprovecha los intervalos de espera de E/S para hacer avanzar otros jobs. Es en esta época en que el monitor residente puede comenzar a llamarse un Sistema Operativo. El sistema operativo debe decidir que jobs hace avanzar y cuales no, tratando de maximizar el rendimiento del computador. Esto se llama *Job scheduling*.

Problema : Ausencia de protección entre jobs

Si se lanza un programa defectuoso, su caída provoca la caída de todo el computador. Surge la necesidad de tener un mecanismo de protección entre jobs.

1.1.6 Máquinas o procesadores virtuales

Aparecen las arquitecturas de computadores que son capaces de emular varias máquinas o procesadores virtuales a partir de un solo procesador real. Cada procesador virtual posee :

- Un espacio de direcciones independiente.
- Dispositivos de E/S independientes.
- Espacios de almacenamiento de información compartidos.
- Interrupciones.
- Tiempo de CPU obtenido en forma de tajadas de tiempo del procesador real.

En los sistemas de multiprogramación se alcanza el mayor rendimiento de un computador.

Problema : Baja productividad de los programadores

Desde la aparición de los sistemas batch los programadores tienen muchas dificultades para depurar sus programas. El tiempo de despacho de cada prueba que realizan es bastante prolongado, aún en los sistemas de multiprogramación. A medida que los costos de los computadores bajan, el costo de los programadores y el tiempo de desarrollo se convierten en una componente importante para el usuario. Es decir, por primera vez el problema se centra en mejorar la productividad de los programadores y no la del computador.

1.1.7 Sistemas de Tiempo Compartido

A principio de los '70 aparecen los primeros sistemas de tiempo compartido. En estos sistemas varios programadores o usuarios pueden trabajar *interactivamente* con el computador a través de terminales. Es decir recuperan aquella capacidad de trabajar directamente con el computador, capacidad que habían perdido con la aparición de los sistemas batch.

Ahora cada programador tiene su propia consola en donde puede realizar una depuración más cómoda. Además los usuarios disponen de un sistema de archivos en línea que pueden compartir. Ya no se habla de tiempo de despacho de un job, sino que mas bien tiempo de respuesta de un comando, el que se reduce drásticamente.

En los sistemas de tiempo compartido los usuarios comparten recursos e información –abaratando el costo del sistema– pero manteniendo una relativa independencia entre programadores o usuarios –sin que se molesten entre ellos.

Problema : El procesador es un cuello de botella

A medida que crece el número de usuarios, el desempeño del sistema se degrada, ya sea por escasez de CPU o escasez de memoria.

1.1.8 Computadores Personales

A fines de los '70, gracias al progreso en la miniaturización es posible incluir todas las funciones de una CPU en un solo chip, el que se llamó un microprocesador. Con ellos se puede construir computadores no muy rápidos (comparados con los sistemas de tiempo compartido de esa época), pero que por su bajo costo es posible destinar a un solo programador o usuario, sin necesidad de compartirlo. De ahí su nombre de computador personal.

Los computadores personales no tienen necesidad de incorporar las características de los sistemas batch o de tiempo compartido. Por ello, en un comienzo sólo incorporan un monitor residente (como CP/M-80 o su sucesor MS-DOS) y no un verdadero sistema operativo que soporte multiprogramación o máquinas virtuales. Tampoco necesita canales de E/S (sin embargo los computadores personales de hoy en día sí incorporan todas estas características).

La atracción que ejercen en esa época los computadores personales se debe a que son sistemas interactivos con un tiempo de respuesta predecible y son de bajo costo. Es decir 10 computadores personales son mucho más baratos que un sistema de tiempo compartido para 10 usuarios.

Problema : Dificultad para compartir información

Cada computador personal tiene que tener su propia impresora y disco. Estos recursos continúan siendo caros, pero no pueden ser compartidos. Por otra parte los programadores y usuarios no pueden compartir información, algo que sí pueden hacer en los sistemas de tiempo compartido para desarrollar un proyecto en grupo.

1.1.9 Redes de computadores personales

Es así como a mediados de los '80 surgen las redes de computadores personales (como Novell). La idea es mantener la visión que tiene un usuario de un computador personal, pero la red le permite compartir el espacio en disco y la impresora con el fin de economizar recursos.

El esquema funciona dedicando uno de los computadores personales a la función de servidor de disco e impresora. Los demás computadores se conectan vía una red al servidor. Una capa en el monitor residente hace ver el disco del servidor como si fuese un disco local en cada computador personal de la red. Lo mismo ocurre con la impresora.

Aunque potencialmente un usuario también podría usar el servidor para sus propios fines, esto no es conveniente, ya que el monitor residente no tiene mecanismo de protección. Si el usuario lanza una aplicación defectuosa, la caída de esta aplicación significará la caída de todos los computadores personales, puesto que el servidor dejará de responder las peticiones de disco o impresión.

La desventaja de este tipo de redes es que no resuelve el problema de compartir información. En efecto, las primeras redes permiten compartir directorios de programas en modo lectura, pero no permiten compartir directorios en modo escritura lo que impide el desarrollo en grupo.

1.1.10 Sistemas distribuidos o redes de estaciones de trabajo

La otra solución surge como respuesta al problema de compartir información. En las estaciones de trabajo se trata de emular un sistema de tiempo compartido, ya que este tipo de sistemas sí permite compartir fácilmente la información. La idea es que el usuario se conecte a un terminal inteligente (estación o puesto de trabajo), es decir que contiene un procesador en donde corren sus programas. Pero la visión que el usuario tiene es la de un sistema de tiempo compartido. *El computador es la red*, los discos que se ven en este computador pueden estar físicamente en cualquiera de las estaciones de trabajo, es decir el sistema computacional de tiempo compartido está físicamente distribuido en varias estaciones de trabajo. De ahí la apelación de sistemas distribuidos.

Este esquema se implementa interconectando computadores de costo mediano a través de una red. Estos computadores son de costo medio porque incorporan todas las características de un sistema de tiempo compartido (sistema operativo, canales, procesadores virtuales), aunque no son tan rápidos como los computadores de tiempo compartido avanzados de la época.

La desventaja de los sistemas distribuidos y de las redes de PCs está en su alto costo de administración. En efecto, estos sistemas requieren atención continua de operadores que intervienen cuando hay momentos de desconexión de la red. Esto se debe a que en la práctica los sistemas distribuidos se implementan en base a *parches* al sistema operativo de tiempo compartido que corre en cada estación. Por lo tanto la mayor parte del software no está diseñado para tolerar desconexiones de la red aunque sean momentáneas.

Del mismo modo, las redes de PCs se implementan como parches al monitor residente de los PCs y por lo tanto adolecen del mismo problema. Hoy en día se realiza una amplia investigación para diseñar sistemas operativos distribuidos tolerantes a fallas.

1.1.11 Sistemas multiprocesadores

La tercera forma de resolver el problema de los sistemas de tiempo compartido es agregar más procesadores al computador central. Un sistema multiprocesador posee de 2 a 20 procesadores de alto desempeño que comparten una misma memoria real (al menos 128 MB).

El sistema multiprocesador se comporta como un sistema de tiempo compartido pero que es capaz de ejecutar efectivamente varios procesos en paralelo, uno en cada procesador. Sin embargo, un proceso nunca se ejecuta al mismo tiempo en más de un procesador. Cuando la cantidad de procesos en ejecución supera al número de procesadores reales, se recurre a la repartición del tiempo de CPU en tajadas, al igual que en los sistemas de tiempo compartido.

Esto significa que el modelo conceptual del proceso no cambia. Un proceso sigue siendo *un solo procesador virtual* con su propia memoria no compartida. Sin embargo existen sistemas operativos que permiten que dos o más procesos compartan parte de sus espacios de direcciones, escapándose de la definición original de proceso.

El problema de los sistemas multiprocesadores es doble. Por el lado del software es difícil implementar sistemas operativos que aprovechen eficientemente todos los procesadores disponibles. Y por lado del hardware, el aumento del costo de un multiprocesador es exponencial con respecto al número de procesadores que comparten la misma memoria. Hoy en día existe una barrera de 20 procesadores.

1.1.12 El presente

Los computadores personales están en el mismo rango de velocidades de las estaciones de trabajo y ambos términos se confunden. Las nuevas implementaciones de los sistemas de tiempo compartido de los '80 han sido ampliamente superadas en rapidez por las estaciones y computadores personales. Sin embargo, estos sistemas de tiempo compartido siguen existiendo debido a la amplia base de aplicaciones que aún necesitan estos computadores para poder correr.

Los computadores personales más veloces se convierten en servidores de disco o bases de datos. Las estaciones de trabajo más veloces se convieten en servidores de disco o aplicación en donde los usuarios pueden correr aquellos programas que necesitan mucha memoria. En algunas instalaciones se usan para ambos fines haciendo que los programas voraces en CPU y memoria degraden el servicio de disco degradando el tiempo de respuesta de todas las estaciones de trabajo.

Capítulo 2

Procesos

Un *proceso* es un *programa en ejecución*. Este programa se ejecuta en un procesador, tiene su código, datos, pila, un contador de programa y un puntero a la pila.

Cuando el sistema operativo ofrece múltiples procesos, los procesadores son virtuales y el núcleo multiplexa el procesador real disponible en tajadas de tiempo que otorga por turnos a los distintos procesos.

Cuando el computador es multiprocesador y el sistema operativo está diseñado para explotar todos los procesadores disponibles, entonces los procesos se pueden ejecutar efectivamente en paralelo.

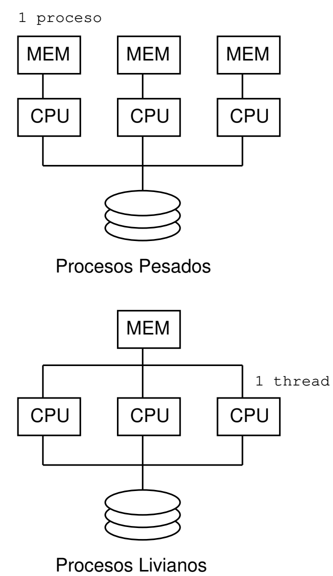

Figura 2.1: Procesos pesados y livianos

2.1 Clasificación de procesos

2.1.1 Procesos pesados versus procesos livianos

Los procesos que implementa un sistema operativo se clasifican según el grado en que comparten la memoria (ver figura 2.1) :

- Procesos Pesados (proceso Unix) : Los procesos no comparten ninguna porción de la memoria. Cada proceso se ejecuta en su propio procesador virtual con CPU y memoria. Todos los procesos sí comparten el mismo espacio de almacenamiento permanente (el disco).
- Procesos Livianos o threads : Los threads comparten toda la memoria y el espacio de almacenamiento permanente.

El primer tipo de procesos se dice *pesado* porque el costo de implementación en tiempo de CPU y memoria es mucho más elevado que el de los procesos *livianos*. Además la implementación de procesos pesados requiere de una MMU o Unidad de Manejo de la Memoria. Esta componente de hardware del procesador se encarga de la traducción de direcciones virtuales a reales. La implementación en software de esta traducción sería demasiado costosa en tiempo de CPU, puesto que para garantizar una verdadera protección habría que recurrir a un intérprete del lenguaje de máquina.

Unix estándar sólo ofrece procesos pesados, pero como veremos existen extensiones que implementan procesos livianos para Unix. Un ejemplo de sistema de procesos livianos es el que implementaba el sistema operativo de los computadores Commodore Amiga, que no tenía la MMU necesaria para implementar procesos pesados.

La ventaja de los procesos pesados es que garantizan protección. Si un proceso falla, los demás procesos continúan sin problemas. En cambio si un thread falla, esto causa la falla de todos los demás threads que comparten el mismo procesador.

La ventaja de los threads es que pueden comunicarse eficientemente a través de la memoria que comparten. Si se necesita que un thread comunique información a otro thread basta que le envíe un puntero a esa información. En cambio los procesos pesados necesitan enviar toda la información a otro procesos pesado usando pipes, mensajes o archivos en disco, lo que resulta ser más costoso que enviar tan solo un puntero.

2.1.2 Preemption versus non-preemption

Los procesos también se pueden clasificar según quién tiene el control para transferir el procesador multiplexado de un proceso a otro :

- Procesos *preemptive* : Es el núcleo el que decide en que instante se transfiere el procesador real de un proceso al otro. Es decir un proceso puede perder el control del procesador en cualquier instante.
- Procesos *non-preemptive* : Es el proceso el que decide en que instante transfiere el procesador real a otro proceso.

Los procesos de Unix son preemptive mientras que los procesos de Windows 3.X o Macintosh Sistema 7.X son non-preemptive. En Windows si una aplicación entra en un ciclo infinito, no hay forma de quitarle el procesador real, la única salida es el relanzamiento del sistema. Lo mismo ocurre en un Macintosh. Por esta misma razón, la tendencia a futuro es que todos los sistemas ofrecerán procesos preemptive.

2.2 El nano System

Dado que en Unix un proceso pesado corre en un procesador virtual es posible implementar un sistema de threads que comparten el mismo espacio de direcciones virtuales en un proceso Unix. Un ejemplo de estos sistemas es el nano System (nSystem de ahora en adelante).

En vez de usar la palabra thread –que es difícil de pronunciar en castellano– usaremos como sinónimo la palabra tarea. Las tareas de nSystem se implementan multiplexando el tiempo de CPU del proceso Unix en tajadas de tiempo que se otorgan por turnos a cada tarea, de la misma forma en que Unix multiplexa el tiempo del procesador real para otorgarlo a cada proceso pesado.

De esta forma varios procesos Unix puede tener cada uno un enjambre de tareas que comparten el mismo espacio de direcciones, pero tareas pertenecientes a procesos distintos *no comparten la memoria.*

2.2.1 Gestión de tareas en nSystem

Los siguientes procedimientos de nSystem permiten crear, terminar y esperar tareas :

- `nTask nEmitTask( int (*proc)(), parametro1 ... parametro6)` : Emite o lanza una nueva tarea que ejecuta el procedimiento `proc`. Acepta un m'aximo de 6 par'ametros (enteros o punteros) que son traspasados directamente a `proc`. Retorna un descriptor de la tarea lanzada.
- `void nExitTask(int rc)` : Termina la ejecución de la tarea que lo invoca, `rc` es el c'odigo de retorno de la tarea.
- `int nWaitTask(nTask task)` : Espera a que una tarea termine, entrega el c'odigo de retorno dado a `nExitTask`.
- `void nExitSystem(int rc)` : Termina la ejecución de todas las tareas. Es decir, termina el proceso Unix con código de retorno `rc`.

En la figura 2.2 se observa que si una tarea A espera otra tarea B antes de que B haya terminado, A se bloquea hasta que B invoque `nExitTask`. De la misma forma, si B termina antes de que otra tarea invoque `nWaitTask`, B no termina –permanece bloqueada– hasta que alguna tarea lo haga.

Como ejemplo veamos como se calcula un número de Fibonacci usando las tareas de nSystem :

```
int pfib(int n)
{
  if (n<=1) return 1;
  else
  {
    nTask task1= nEmitTask(pfib, n-1);
    nTask task2= nEmitTask(pfib, n-2);
    return nWaitTask(task1)+nWaitTask(task2);
} }
```

La ejecución de `pfib(6)` crea dos tareas que calculan *concurrentemente* `pfib(5)` y `pfib(4)`. Como a su vez estos dos procedimientos crean nuevas tareas, al final se crea un árbol de tareas.

Observe que esta solución se introduce sólo con motivos pedagógicos. Ella es sumamente ineficiente como mecanismo para calcular un número de Fibonacci, puesto que el sobrecosto natural introducido por la

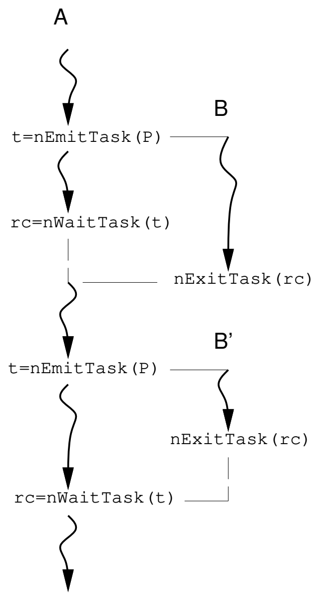

Figura 2.2: Creación, espera y término de tareas en nSystem

gestión de un número exponencial de tareas es colosal. Para que esta solución sea eficiente se necesitaría una implementación de nSystem que usara procesadores reales para cada tarea creada, pero en la actualidad esto no es posible en ningún multi-procesador de memoria compartida.

2.2.2 El problema de la sección crítica

Supongamos que en el problema anterior se desea contar el número de tareas que se crean. La solución natural que se viene en mente es introducir una variable global `count` que se incrementa en cada llamada de `pfib`:

```
int count=0;
int pfib(int n)
{
  count++;
  if (...) ...
  else ...
}
```

Esta solución es errada. En efecto, la instrucción en C `count++` se traduce por una secuencia de instrucciones de máquina del tipo:

```
Load count, Reg   ;  Reg= count
Add  Reg, 1, Reg  ;  Reg= Reg+1
Store Reg, count  ;  count= Reg
```

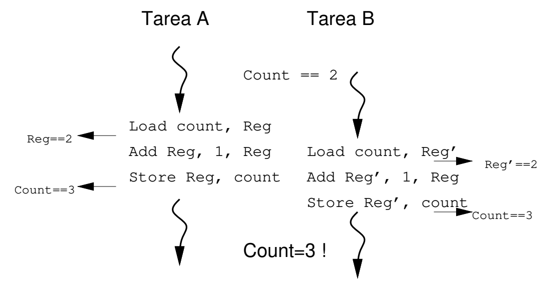

Figura 2.3: Ejecución concurrente de una misma sección crítica

En la figura 2.3 se aprecia cómo la ejecución en paralelo de dos de estas secuencias puede terminar incrementando `count` sólo en 1. Esta situación es un error porque no se contabiliza una de las invocaciones de `pfib`.

En el ejemplo hemos supuesto que ambas tareas se ejecutan en dos procesadores reales. Sin embargo, hemos dicho que nSystem no implementa paralelismo real sino que emula varios procesadores a partir de un solo proceso multiplexando en tajadas el tiempo de CPU. Esta forma de ejecutar procesos también puede resultar en un número errado para `count`, cuando una tajada de tiempo de CPU se acaba justo cuando un proceso está en medio del incremento de `count` [1].

Este tipo de problemas es muy común en los procesos que comparten algún recurso, en este caso la variable `count`. El problema es que la utilización del recurso no es atómica, se necesita una secuencia de instrucciones para completar una operación sobre el recurso, en el ejemplo, la contabilización de una invocación de `pfib`. El problema es que la ejecución de esta operación no debe llevarse a cabo concurrentemente en más de una tarea en un instante dado, porque el resultado será impredescible.

Una *sección crítica* es una porción de código que sólo puede ser ejecutada por una tarea a la vez para que funcione correctamente. En el ejemplo la instrucción `count++` (es decir su traducción a instrucciones de máquina) es un a sección crítica.

2.2.3 Comunicación entre tareas en nSystem

En nSystem se pueden implementar secciones críticas mediante mensajes. El envío de mensajes en nSystem se logra con los siguientes procedimientos:

[1] Más adelante veremos que nSystem puede ejecutar las tareas en modo non-preemptive. En este caso, como no hay tajadas de tiempo, los incrementos de `count` no pueden entremezclarse.

- `int nSend(nTask task, void *msg)`: Envía el mensaje `msg` a la tarea `task`. Un mensaje consiste en un puntero a un área de datos de cualquier tamaño. El emisor se queda bloqueado hasta que se le haga `nReply`. `nSend` retorna un entero especificado en `nReply`.

- `void *nReceive(nTask *ptask, int max_delay)`: Recibe un mensaje proveniente de cualquier tarea. La identificación del emisor queda en `*ptask`. El mensaje retornado por `nReceive` es un puntero a un área que ha sido posiblemente creada en la pila del emisor. Dado que el emisor continúa bloqueado hasta que se haga `nReply`, el receptor puede accesar libremente esta área sin peligro de que sea destruida por el emisor.

  La tarea que lo invoca queda bloqueada por `max_delay` miliseg, esperando un mensaje. Si el período finaliza sin que llegue alguno, se retorna `NULL` y `*ptask` queda en `NULL`. Si `max_delay` es 0, la tarea no se bloquea (nReceive retorna de inmediato). Si `max_delay` es -1, la tarea espera indefinidamente la llegada de un mensaje.

- `void nReply(nTask task, int rc)`: Responde un mensaje enviado por `task` usando `nSend`. Desde ese instante, el receptor no puede accesar la información contenida en el mensaje que había sido enviado, ya que el emisor podría destruirlo. El valor `rc` es el código de retorno para el emisor. nReply no se bloquea.

Con estos procedimientos el problema de la sección crítica se puede resolver creando una tarea que realiza el conteo de tareas.

```
int count=0; /* Contador de tareas */

/* Crea la tarea para el conteo y la tarea
 * que calcula concurrentemente Fibonacci(n)
 */
int pfib(int n, int *pcount)
{
  nTask counttask= nEmitTask(CountProc);
  int ret= pfib2(n, counttask);
  int end= TRUE;
  nSend(counttask, (void*)&end);
  *pcount= nWaitTask(counttask);
  return ret;
}

/* Realiza el conteo de tareas */
int CountProc()
{
  int end= FALSE;
  int count=0;

  do
  {
    nTask sender;
    int *msg= (int*)nReceive(&sender, -1);
    end= *msg;
    count++;
    nReply(sender, 0);
  }
    while (end==FALSE);

  return count-1;
}

/* Calcula Concurrentemente Fibonacci(n) */
int pfib2(int n, nTask counttask)
{
  int end= FALSE;
  nSend(counttask, (void*)&end);

  if (n<=1) return 1;
  else
  {
    nTask task1=nEmitTask(pfib,n-1,counttask);
    nTask task2=nEmitTask(pfib,n-2,counttask);
    return nWaitTask(task1)+nWaitTask(task2);
} }
```

Esta solución resuelve efectivamente el problema de la sección crítica, puesto que la única tarea que incrementa el contador es `counttask`, y ésta se ejecuta estrictamente en secuencia. En el fondo, la solución funciona porque el contador ya no es una variable de acceso compartido. Aunque todas las tareas tienen acceso a este contador, sólo una tarea lo accesa verdaderamente.

Es importante destacar que el siguiente código podría no funcionar.

```
int *msg= (int*)nReceive(&sender, -1);
nReply(sender, 0);
end= *msg;
```

En efecto, al hacer `nReply` el emisor puede continuar y retornar de `pfib2` y por lo tanto destruir la variable `end` que contiene el mensaje, antes de que se ejecute `end=*msg;`. Esto entregaría resultados impredescibles.

El paradigma que hay detrás de esta solución es el de cliente/servidor. Se crea una tarea servidor (`counttask`) que da el servicio de conteo, mientras que `fibtask` y las demás tareas que se crean para calcular Fibonacci son los clientes.

Nótese que al introducir este servidor, ni siquiera con infinitos procesadores la solución sería eficiente puesto que se establece un cuello de botella en el servidor. En efecto el servidor de conteo es estrictamente secuencial. Las dos tareas que se crean concurrentemente tienen que ser atendidas secuencialmente por el servidor.

Bibliotecas de Unix y nSystem

Muchos procedimientos de biblioteca de Unix pueden continuar usándose en nSystem, sin embargo gran cantidad de procedimientos de biblioteca usan variables globales cuyo acceso se convierte en una sección crítica en nSystem. El código original de Unix no implementa ningún mecanismo de control que garantice que una sola tarea está invocando ese procedimiento, puesto que Unix estándar no ofrece threads. Por lo tanto los procedimientos de biblioteca de Unix que usan y modifican variables globales *no pueden ser invocados directamente desde nSystem.* Para llamar estos procedimientos en nSystem hay que implementar un mecanismo de control de acceso entre tareas.

2.2.4 Otros procedimientos de nSystem

Durante el lanzamiento de nSystem se realizan algunas inicializaciones y luego se invoca el procedimiento `nMain`, el que debe ser suministrado por el programador. Éste procedimiento contiene el código correspondiente a la primera tarea del programador. Desde ahí se pueden crear todas las tareas que sean necesarias para realizar el trabajo. El encabezado para este procedimiento debe ser :

```
int nMain(int argc, char *argv[])
{
   ...
}
```

El retorno de `nMain` hace que todas las tareas pendientes terminen y por lo tanto es equivalente a llamar `nExitSystem`. Es importante por lo tanto evitar que `nMain` se termine antes de tiempo.

Los siguientes procedimientos son parte de nSystem y pueden ser invocados desde cualquier tarea de nSystem.

Par'ametros para las tareas :

- `void nSetStackSize(int new_size)` : Define el tamaño de la pila para las tareas que se emitan a continuación.
- `void nSetTimeSlice(int slice)` : Tamaño de la tajada de tiempo para la administración *Round–Robin*[2] (preemptive) (`slice` está en miliseg). Degenera en *FCFS* (non–preemptive) si `slice` es cero, lo que es muy útil para depurar programas. El valor por omisión de la tajada de tiempo es cero.
- `void nSetTaskName(char *format, <args> ... )` : Asigna el nombre de la tarea que la invoca. El formato y los parámetros que recibe son análogos a los de `printf`.

Entrada y Salida :

Los siguientes procedimientos son equivalentes a `open`, `close`, `read` y `write` en UNIX. Sus parámetros son los mismos. Los "nano" procedimientos sólo bloquean la tarea que los invoca, el resto de las tareas puede continuar.

- `int nOpen(char *path, int flags, int mode)` : Abre un archivo.
- `int nClose(int fd)` : Cierra un archivo.
- `int nRead(int fd, char *buf, int nbyte)` : Lee de un archivo.
- `int nWrite(int fd, char *buf, int nbyte)` : Escribe en un archivo.

Servicios Misceláneos :

- `nTask nCurrentTask(void)` : Entrega el identificador de la tarea que la invoca.
- `int nGetTime(void)` : Entrega la "hora" en miliseg.
- `void *nMalloc(int size)` : Es un `malloc` non-preemptive.
- `void nFree(void *ptr)` : Es un `free` non-preemptive.
- `void nFatalError(char *procname, char *format, ... )` : Escribe salida formateada en la salida estándar de errores y termina la ejecución (del proceso Unix). El formato y los parámetros que recibe son análogos a los de `printf`.

[2] *Round–Robin* y *FCFS* serán vistos en el capítulo de administración de procesos.

- `void nPrintf(char *format, ... )` : Es un `printf` sólo bloqueante para la tarea que lo invoca.
- `void nFprintf(int fd, char *format, ... )` : Es "como" un `fprintf`, sólo bloqueante para la tarea que lo invoca, pero recibe un fd, no un `FILE *`.

2.2.5 Buffering usando tareas

Veamos como se resuelve el problema del buffering usando tareas. Este mismo problema se resolverá en el próximo capítulo usando interrupciones y canales.

```
nTask bufftask= nEmitTask(BuffServ);

void ReadBuff(int *pbuff)
{
  nSend(bufftask, (void*)pbuff);
}

void BuffServ()
{
  char pbuff2[512];

  for(;;)
  {
    nTask clienttask;
    char *pbuff;

    nRead(0, pbuff2, 512);
    pbuff= (char*)nReceive(&clienttask,-1);
    bcopy(pbuff, pbuff2, 512);
    nReply(clienttask, 0);
} }
```

Esta solución permite efectivamente superponer la lectura de un dispositivo (por el descriptor Unix número 0). En efecto, una vez que el servidor `BuffServ` entrega un bloque de datos, el cliente (la aplicación) procesa este bloque al mismo tiempo que el servidor ya está leyendo el próximo bloque. Con suerte cuando el servidor vuelva a recibir un mensaje pidiendo un nuevo bloque, ese bloque ya estará disponible.

Observe que esta solución también necesita interrupciones para poder implementarse, sólo que esta vez la gestión de las interrupciones la realiza nSystem y no el programador de la aplicación.

2.3 Semáforos

Los mensajes de nSystem son un mecanismo para sincronizar procesos. Existen otros mecanismos que veremos a partir de esta sección, algunos radicalmente distintos, otros también basados en mensaje pero con una *semántica* distinta, es decir el funcionamiento de los mensajes es distinto.

Un *semáforo* es un distribuidor de tickets y sirve para acotar el número de tareas que pasan por una determinada instrucción. Para ello un semáforo S dispone de una cierta cantidad de tickets a repartir.

Las operaciones que acepta un semáforo son :

- Wait(S) : Pide un ticket al semáforo. Si el semáforo no tiene tickets disponibles, el proceso se bloquea hasta que otro proceso aporte tickets a ese mismo semáforo.
- Signal(S) : Aporta un ticket al semáforo. Si había algún proceso esperando un ticket, se desbloquea. Si había más procesos esperando tickets, se desbloquea el primero que llegó pidiendo tickets.

En nSystem los semáforos se crean con :

```
nSem sem= nMakeSem(inittickets);
```

En donde `inittickets` es la cantidad inicial de tickets. Las operaciones Wait y Signal se llaman `nWaitSem` y `nSignalSem` respectivamente. Además un semáforo se destruye con `nDestroySem`.

Uno de los usos de los semáforos es la implementación de secciones críticas. Por ejemplo el problema del conteo de `pfib` se puede resolver con :

```
int count= 0;
nSem sem= nMakeSem(1);

int pfib(int n)
{
  nWaitSem(sem);
  count++;
  nSignalSem(sem);
  ...
}
```

El semáforo contiene inicialmente un solo ticket. La primera tarea que llega a la sección crítica pide el único ticket, de modo que ninguna otra tarea puede entrar a la sección crítica. Al salir la tarea devuelve el ticket para que otra tarea pueda entrar a la sección crítica.

2.3.1 El problema del productor/consumidor

Un proceso productor produce ítemes depositándolos en un *buffer*. Los ítemes son extraídos del buffer por

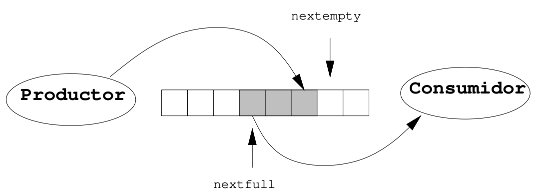

Figura 2.4: El problema del productor/consumidor

un proceso consumidor. El buffer puede contener hasta un máximo de `N` ítemes. El productor y consumidor trabajan concurrentemente.

El buffer se mantiene en un arreglo de `N` ítemes de tipo `Item`. La variable `nexempty` indica el índice en el arreglo en donde hay que depositar el próximo ítem producido. La variable `nextfull` indica el índice en el arreglo de donde hay que extraer el próximo ítem por consumir. Esta estructura se aprecia en la figura 2.4.

El problema está en lograr que cuando el consumidor trata de extraer un ítem cuando el buffer está vacío, el consumidor se bloquee hasta que el productor deposite un ítem. Del mismo modo, cuando el productor intenta depositar un ítem cuando el buffer ya tiene el máximo de `N` ítemes, el productor se debe bloquear hasta que el consumidor extraiga un ítem.

Veamos una solución de este problema usando dos semáforos. Un semáforo mantendrá el número de posiciones disponibles para depositar ítemes en el buffer. Cuando el productor deposita un ítem resta un ticket con Wait, dado que hay un posición menos en el buffer. Si el buffer está lleno, el productor se bloqueará ya que no hay más tickets en el semáforo. Cuando el consumidor extrae un elemento agrega un ticket con Signal, porque ahora hay una nueva posición disponible.

El segundo semáforo sirve para mantener el número de ítemes dispobibles en el buffer. El productor agrega tickets con Signal y el consumidor quita tickets –si hay disponibles– con Wait.

La siguiente es una implementación del productor y consumidor.

Inicialmente se tiene :

`empty` un semáforo con N tickets
`full` un semáforo con 0 tickets
`buffer` un arreglo de N Item
`nextempty=0`
`nextfull=0`

```
void Productor()
{
  for(;;) /* Ciclo infinito */
  {
    Item x= Produce();
    /* Listo, ya tenemos un item */

    Wait(empty);
    buff[nextempty]= x;
    nextempty= (nextempty+1)%N;
    Signal(full);
} }

Consumidor()
{
  for(;;) /* Ciclo infinito */
  {
    Item x;
    Wait(full);
    x= buff[nextfull];
    nextfull=(nextfull+1)%N;
    Signal(empty);
    Consume(x);
    /* Listo, ya consumimos el item */
} }
```

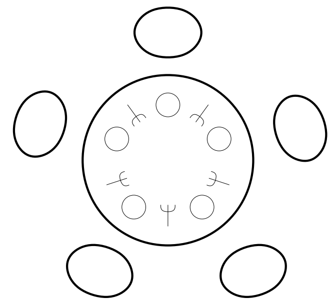

Figura 2.5: El problema de los filósofos comiendo.

Observe que las modificaciones de las variables `nextempty` y `nextfull` no son secciones críticas, puesto que cada variable es modificada por un solo proceso. Sin embargo si generalizamos el problema a varios productores y consumidores, las modificaciones sí serán secciones críticas.

Ejercicio : Modifique esta solución para que funcione en el caso de varios productores y consumidores.

2.3.2 El problema de los filósofos pensantes

5 filósofos pasan sus días pensando y comiendo. Cada filósofo tiene su puesto asignado en una mesa con 5 sillas. En la mesa hay 5 tenedores como se muestra en la figura 2.5.

Cuando un filósofo piensa no interactúa con el resto de sus colegas. Pero de vez en cuando necesita comer y por lo tanto se sienta en su silla. Para poder comer necesita los dos tenedores que están a ambos lados de su plato. Entonces debe intentar tomar un tenedor primero y *después* el otro. Si alguno de estos tenedores está en poder de su vecino, el filosófo no puede comer. Debe esperar a que los dos tenedores estén libres. Si logró tomar uno de los tenedores, el filósofo puede mantenerlo mientras espera el otro, como también puede soltarlo para que su otro vecino pueda comer.

Este es el más clásico de los problemas de sincronización debido a que es muy usual encontrar problemas equivalentes en los sistemas concurrentes y distribuidos. Lamentablemente no siempre es sencillo detectar su presencia, por lo que también se le atribuye muchas de las caídas de este tipo de sistemas.

Solución errónea

Cada filósofo es un proceso. El uso de cada tenedor es una sección crítica, ya que dos filósofos no pueden estar usando el mismo tenedor en un instante dado. El acceso a los tenedores se controla en un arreglo de 5 semáforos con un ticket cada uno. Para poder comer, el filósofo `i` debe pedir el ticket del semáforo `i` y el del semáforo `(i+1)%5`.

Inicialmente se tiene :
`tenedor` es un arreglo de 5 semáforos,
cada uno con un solo ticket.

```
Folosofo(int i)
{
  for(;;)
  {
    Wait(tenedor[i]);
    Wait(tenedor[(i+1)%5]);
    Comer();
    Signal(tenedor[i]);
    Signal(tenedor[(i+1)%5]);
    Pensar();
} }
```

Observe que en esta solución, cuando un filósofo tomó un tenedor no lo suelta a pesar de que tiene que esperar por el próximo tenedor.

Esta solución es errónea porque los filósofos pueden llegar a una situación de bloqueo eterno, o en inglés *dead-lock*. Esta situación se produce cuando cada filósofo está en poder de un tenedor y espera a que el otro se desocupe, lo que nunca ocurrirá. En estas condiciones los filósofos morirán de hambre.

Primera solución

Se tiene especial cuidado al tomar los tenedores, de modo que siempre se tome primero el tenedor con menor índice en el arreglo de semáforos. El código para tomar los tenedores queda :

```
j=min(i,(i+1)%5);
k=max(i,(i+1)%5);
Wait(tenedor[j]);
Wait(tenedor[k]);
```

Con este orden es imposible la situación de *dead-lock* en que cada proceso tenga un tenedor y espere el otro, puesto que el último tenedor (índice 4) siempre se pide en segundo lugar. Por lo tanto se garantiza que al menos uno de los filósofos podrá tomar este tenedor y comer. En el peor de los casos tendrán que comer secuencialmente.

Segunda solución

Permitir un máximo de 4 filósofos en la mesa. Para lograr esto se necesita un nuevo semáforo (`sala`) que parte inicialmente con 4 tickets.

```
Folosofo(int i)
{
  for(;;)
  {
    Wait(sala);
    Wait(tenedor[i]);
    Wait(tenedor[(i+1)%5]);
    Comer();
    Signal(tenedor[i]);
    Signal(tenedor[(i+1)%5]);
    Signal(sala);
    Pensar();
} }
```

Con esta restricción también se garantiza que al menos un filósofo podrá comer. Al resto tarde o temprano le llegará su turno.

2.4 Monitores

Un monitor es la versión concurrente de una estructura de datos : posee un estado interno más un conjunto de operaciones. A pesar de que las operaciones se invocan concurrentemente, el monitor las ejecuta secuencialmente.

Los monitores fueron inventados por Hoare para un dialecto de Pascal, de ahí su sintaxis a la Pascal.

Sintaxis :

```
type <nombre> = monitor
  <variables>
  procedure entry <proc1>( <argumentos1> );
```

```
    <cuerpo>
  procedure entry <proc2>( <argumentos2> );
    <cuerpo>
    ...
  begin
    <inicializaciones>
  end
```

Las operaciones son <proc1>, <proc2>, ... El código asociado a todas estas operaciones es una sección crítica y por lo tanto nunca hay dos instancias en ejecución. Sin embargo el monitor puede suspender la ejecución de una operación para eventualmente ejecutar otra y más tarde retomar la operación suspendida. Esto se logra mediante variables de tipo `condition`.

Declaración : `var x: contition`

Las siguientes acciones sobre variables condicionales deben ser invocadas dentro de una de las operaciones del monitor.

- `x.wait` : El proceso que llega a ejecutar esta acción se suspende hasta que otro proceso invoque `x.signal` dentro del mismo monitor. El proceso queda en una cola FIFO que forma parte de `x`.
- `x.signal` : Si no hay procesos encolados en esta variable, la acción se ignora (no confundir con semáforos). Si `x` tiene procesos encolados, el proceso que invoca `x.signal` cede el monitor al primer proceso que quedó en espera. El proceso que ha cedido el monitor podrá recuperarlo cuando el proceso que lo recibió salga del monitor, salvo que este último lo ceda con otro `signal` a un tercer proceso.

Resolvamos el problema del productor/consumidor con monitores. Inicialmente :

```
Var buffer: BUFFER;
```

Para depositar un ítem `x`, un productor ejecuta `buffer.Put(x);` mientras que para extraer un ítem, un consumidor ejecuta `buffer.Get(x);` donde `x` se pasa por referencia.

El tipo BUFFER se define como un monitor en el siguiente código :

```
type BUFFER= monitor

  var pool: array[0..N-1] of Item;
      in, out, count: Integer;
      noempty, nofull: condition;

  Procedure entry Put(x: Item);
  begin
    if (count=n) then nofull.wait;
    pool[in]:= x;
    in:= (in+1) mod N;
    count:= count+1;
    noempty.signal;
  end

  Procedure entry Get(var x: Item);
  begin
    if (count=0) then noempty.wait;
    x:= pool[out];
    out:= (out+1) mod N;
    count:= count+1;
    nofull.signal;
  end;
```

2.4.1 El problema de los escritores y lectores

En este problema varios procesos concurrentes comparten una misma estructura de datos y necesitan consultarla o actualizarla (modificarla). Un proceso lector es aquel que está consultando la estructura y un proceso escritor es aquel que la está modificando. Las características de este problema son las siguientes :

- Se permite varios procesos lectores al mismo tiempo.
- Sólo se permite un escritor en un instante dado. Mientras el escritor modifica la estructura no puede haber procesos lectores ni otros procesos escritores.

Veamos una solución usando monitores. El monitor `M` controlará el acceso a la estructura de datos. Se definen entonces las siguientes operaciones en el monitor :

- `M.EnterRead`: Un proceso anuncia que entra en un trozo de código en que va a consultar la estructura de datos. Es decir el proceso se convierte en lector.
- `M.ExitRead`: El lector anuncia que sale del código de consulta.
- `M.EnterWrite`: Un proceso anuncia que entra a un segmento de código en que va a actualizar la estructura de datos. Es decir el proceso se convierte en un escritor.
- `M.ExitWrite`: El escritor anuncia que sale del código de actualización.

Solución

```
Var M: Monitor
  var readers: integer;
      writing: boolean;
      canread: condition;
      canwrite: condition;
  Procedure Entry EnterRead
  begin
    if (writing) then
       canread.wait;
    Incr(readers);
    canread.signal
  end;
  Procedure Entry ExitRead
  begin
    Decr(readers);
    if (readers=0) then
      canwrite.signal
  end;
  Procedure Entry EnterWrite
  begin
    if ((readers>0)
        or writing) then
      canwrite.wait; { (1) }
    writing:= true
  end;
  Procedure Entry ExitWrite
  begin
    writing:= false;
    canwrite.signal; { (2) }
    if (not writing) then
      canread.signal { (3) }
  end
  begin
    writing:= false;
    readers:= 0
  end
```

Observe que en `(2)` se presentan dos posibilidades. En la primera hay escritores esperando por lo que el proceso cede el monitor al escritor bloqueado en `(1)` y sólo lo recuperará cuando el escritor salga del monitor. Al recuperar el monitor `writing` será verdadera. La segunda posibilidad es que no hayan escritores en espera. En este caso el proceso no cede el control y `writing` será falso, por lo tanto si hay lectores esperando, éstos serán desbloqueados en `(3)`.

Esta solución tiene inconvenientes, puesto que los escritores pueden sufrir *hambruna* (*starvation*). Un proceso sufre hambruna cuando otros procesos pueden concertarse para hacer que nunca logre obtener un recurso. En este caso, si siempre hay lectores (`readers>0`) consultando la estructura, los escritores serán bloqueados para siempre.

El problema de la hambruna de los escritores se puede resolver haciendo que no entren más lectores cuando hay escritores esperando. Esto se puede lograr con pequeñas modificaciones a la solución presentada. Sin embargo ahora el problema es que los escritores pueden causar hambruna a los lectores.

Cuando una solución no provoca hambruna a ninguna tarea se dice que esta solución es *equitativa* (en inglés *fair*). Una solución equitativa del problema de los lectores y escritores es que éstos sean atendidos en orden FIFO, pero atendiendo concurrentemente todos los lectores que lleguen durante un lapso en que no llegan escritores. Lamentablemente, no es sencillo implementar esta solución con monitores. Luego implementaremos esta solución usando mensajes y tareas en nSystem.

2.5 Sistemas de mensajes

Como mencionamos antes, los mensajes se inventaron para hacer que procesos que no comparten memoria puedan comunicarse, pero también pueden usarse para sincronizar procesos que comparten la memoria.

En todo sistema de mensajes existe una operación para *enviar* un mensaje y otra para *recibir* un mensaje. En algunos sistemas también existe una operación para *responder* un mensaje. Sin embargo la *semántica* de estos sistemas varía ampliamente, y por lo tanto la programación también varía. A continuación clasificaremos los sistemas de mensajes según sus aspectos semánticos.

Comunicación directa

En este tipo de sistemas los mensajes se envían directamente al proceso que debe recibirlo. Para ello, al enviar el mensaje se especifica la identificación del proceso receptor (*process identication* o *pid*).

```
send(dest_pid, msg)
```

Ejemplos de este tipo de mensajes son los que implementan nSystem y el lenguaje ADA.

Los sistemas de comunicación directa se presentan en varios sabores:

i. Recepción de un solo proceso: El sistema permite seleccionar el proceso emisor del mensaje.

```
msg= Receive(sender_pid)
```

Si es otro el proceso emisor, el mensaje queda encolado hasta que se especifique la identificación de ese proceso.

ii. Recepción de cualquier proceso : El sistema obliga a recibir los mensajes provenientes de cualquier proceso.

```
msg= Receive(&sender_pid)
```

Esto significa que el receptor debe estar siempre preparado para recibir mensajes de cualquier proceso. Este es el caso de nSystem y ADA.

iii. Recepción de algunos procesos : Se puede especificar un conjunto de los identificadores de procesos de donde se acepta la recepción de mensajes de inmediato. Los mensajes de otros procesos quedan encolados.

iv. *Broadcast* o envío a todos los procesos : Se envía un mensaje que es recibido por todos los procesos.

```
Broadcast(msg)
```

El mensaje se recibe en cada proceso mediante `Receive`.

v. *Multicast* o envío a varios procesos : Se envía un mismo mensaje a varios procesos. Esto es equivalente a usar varias veces `Send` con el mismo mensaje pero con distintos destinatarios.

Comunicación indirecta

Los mensajes se envía a través de canales de comunicación. Al enviar un mensaje se especifica la identificación del canal.

```
send(chan_id, msg)
```

Por ejemplo los *streams* o *pipes* de Unix corresponden a este tipo de mensajes.

En algunos sistemas de comunicación indirecta también se puede especificar un conjunto de canales de donde recibir mensajes. Por ejemplo en Unix se usa `select` para seleccionar un conjunto de *streams* por lo cuales se desea recibir datos. Esto es muy útil cuando se desea leer comandos ingresados a través de varios terminales. En Unix System V se puede hacer lo mismo con `poll`.

Los canales de comunicación pueden ser del siguiente tipo :

i. Punto a punto unidireccional : Sólo un proceso puede enviar mensajes por un canal y sólo un proceso puede recibir mensajes de ese canal. En el lenguaje OCCAM los canales son de este tipo.

ii. Punto a punto bidireccional : Un par de procesos usa el canal para comunicarse en ambos sentidos. Los *streams* de Unix pertenecen a esta clase de canales.

iii. Puerta o *port* : Cualquier proceso puede enviar mensajes por ese canal. El emisor especifica otro canal por donde recibir la respuesta.

Los canales pueden estar orientados hacia el envío de mensajes como es el caso del lenguaje OCCAM. En otros sistemas los canales están orientados hacia el envío de secuencias de caracteres como es el caso de los *streams* de Unix.

Buffering

Los sistemas de mensajes se agrupan según la cantidad de mensajes que se pueden enviar sin que el emisor se quede bloqueado.

i. *0-buffering* : Al enviar un mensaje, el emisor se queda bloqueado hasta que :

   - El receptor reciba el mensaje (con `Receive`).
   - El receptor responda el mensaje (con `Reply`).

   Al segundo tipo de mensajes corresponde el nSystem y los sistemas de RPC o *Remote Procedure Call*.

   La comunicación del tipo *0-buffering* también se denomina *síncrona*.

ii. buffer acotado : Los mensajes se copian en un buffer de tamaño finito de modo que el emisor puede continuar y enviar nuevos mensajes. Si el buffer está lleno, el emisor se bloquea hasta que algún proceso receptor desocupe parte del buffer.

   Este tipo de comunicación se denomina *asíncrona*. Al enviar un mensaje, el solo hecho de que `Send` retorne no implica que el mensaje fue recibido. Si el emisor necesita conocer cuando un mensaje fue recibido, debe esperar un mensaje de vuelta (un reconocimiento) del receptor.

   Emisor :

```
Send(dest_pid, msg);
ack= Receive(dest_pid);
```

Receptor :

```
msg= Receive(&dest_pid);
Send(dest_pid, ack);
```

iii. buffer ilimitado : Como el caso anterior, sólo que el buffer es de tamaño infinito. Esta clasificación sólo sirve para fines teóricos, pues en la práctica no existe.

Capítulo 3

Estructura del Computador

En este capítulo veremos la organización de un computador tanto del punto de vista arquitectónico del hardware como del software.

3.1 Arquitectura Lógica del Computador

La arquitectura lógica del computador es la visión que tiene un programador de la máquina sobre la cual se pretende programar el núcleo de un sistema operativo. En esta visión no interesa si el computador tiene memoria caché o si el procesador usa *pipeline* para ejecutar las instrucciones de máquina, puesto que éstas son características que no influyen en el cómo se programa la máquina (salvo si se considera la eficiencia).

En cambio en la arquitectura lógica de un computador sí interesa cuál es su lenguaje de máquina, cómo se comanda/examinan los dispositivos, cómo se reciben las interrupciones, etc. Un programador del sistema operativo sí necesita conocer esta información para saber cómo programarlo.

3.1.1 Espacio de Direcciones Reales

El programador debe ser capaz de direccionar cada palabra en memoria real, cada puerta de dispositivo que posea el computador, cada pixel de la memoria de video y cada palabra de la memoria ROM para el *bootstrap* del sistema. El conjunto de direcciones que se pueden usar para estos fines se denomina el espacio de direcciones reales del computador (ver figura 3.1).

Un programador comanda o examina un dispositivo escribiendo o leyendo en la dirección asignada a este dispositivo en el espacio de direcciones. Esto es lo que se denomina Entrada/Salida mapeada en la memoria (lo que se debe interpretar como el espacio de direcciones reales).

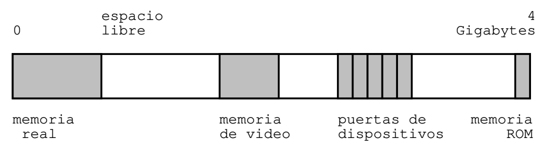

Figura 3.1: Espacio de direcciones de un computador.

3.1.2 Estrategias de E/S

Busy-waiting

Cuando el programador necesita ser informado de la ocurrencia de un evento, entra en un ciclo de consulta constante del dispositivo, leyendo la puerta de control del dispositivo que le informa si tal evento ocurrió o no. En esta estrategia, este ciclo de consulta ocupa el 100% de tiempo de CPU y de ahí el nombre de busy-waiting.

A modo de ejemplo supongamos que tenemos un dispositivo de entrada que se lee en bloques de 512 bytes. El dispositivo tiene una puerta de control en la dirección `0xe0000` que indica si hay un byte para leer y una puerta de datos en la dirección `0xe0001` por donde se reciben los datos. Un programa lee un bloque llamando a la rutina `ReadBlock` suministrando la dirección de un área de 512 bytes en donde se debe dejar el resultado. La siguiente es una implementación de `ReadBlock`.

```
void ReadBlock(char *pbuff)
{
  char* puerta_control= (char*)0xe0000;
  char* puerta_datos=puerta_control+1;
  int nbytes=512;

  while (nbytes--)
  {
    /* Ciclo de busy-waiting */
    while (*puerta_control==0)
      /* todavia no llega un byte */ ;
```

```
    /* Ahora si llego un byte */
    /* Leemos el byte y lo colocamos
       en el area suministrada */
    *pbuff++= *puerta_datos;
} }
```

Interrupciones

En esta estrategia, se programa el dispositivo y el controlador de interrupciones para que produzcan una interrupción en el momento de ocurrir el evento esperado en este dispositivo. Luego, el sistema operativo puede desligarse completamente del dispositivo y dedicarse a otras actividades.

Cuando se produce la interrupción, el programa en ejecución se suspende y se ejecuta una rutina de atención encargada de realizar las acciones que requiere el dispositivo que interrumpe. Cuando esta rutina de atención retorna, se retoma el programa interrumpido.

A pesar que una interrupción es completamente transparente para el programa interrumpido (si la rutina de atención está bien programada) desde el punto de vista funcional, un usuario sí puede percibir una degradación en la rapidez de proceso si el número de interrupciones es muy elevado.

En este mecanismo cada byte o palabra transferidos entre el procesador y un dispositivo provoca una interrupción. Dado que un procesador de hoy en día no puede atender más de 100.000 interrupciones por segundo la velocidad de transferencia entre procesador y dispositivos usando interrupciones está limitada a no más de 100.000 bytes/palabras por segundo. Esta velocidad límite es inferior a la del mecanismo de busy-waiting.

El sistema operativo también puede *inhibir* las interrupciones por cortos períodos cuando necesita ejecutar trozos de código que no puede ser interrupidos. Para ello un bit en la palabra de estado del procesador indica si las interrupciones están inhibidas o no.

Localización de un dispositivo

Para ubicar el dispositivo que interrumpió existen las técnicas de *polling* y vector de interrupciones.

En la técnica de polling el sistema operativo no sabe cuál dispositivo interrumpió y por lo tanto tiene que consultar secuencialmente todos los dispositivos hasta dar con aquel o aquellos que interrumpen. Este mecanismo tiene asociado un tiempo prolongado de reacción a las interrupciones.

El segundo mecanismo se basa en que cada fuente de interrupción tiene asociado un índice. Este índice

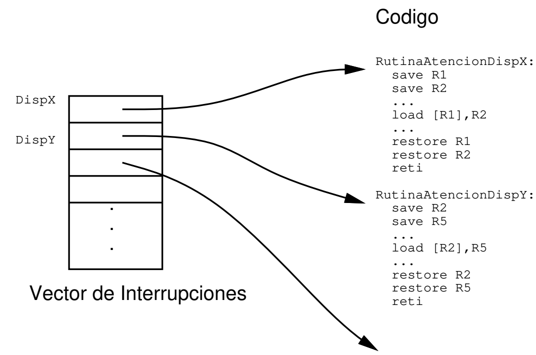

Figura 3.2: Vector de interrupciones.

se usa para subindicar una tabla de rutinas de atención (ver figura 3.2). Por lo tanto cada fuente de interrupción tiene asociada una única rutina de atención. Esta tabla se denomina vector de interrupciones y se ubica en algún lugar de la memoria real del computador.

También existen esquemas mixtos en donde varios dispositivos tienen asociado una misma posición en el vector de interrupciones (debido a que usan la misma línea de interrupción). En ese caso el sistema operativo debe recurrir a la técnica de polling para ubicar entre estos dispositivos cuál fue el que interrumpió.

Adaptemos el ejemplo anterior modificando `ReadBlock` de modo que la lectura del próximo bloque se realice en paralelo con el bloque que ya se leyó. Para esto después de terminar la lectura de un bloque habilitamos las interrupciones del dispositivo. Los bytes que se vayan recibiendo se acumulan en un área escondida (`pbuff2`). Con suerte se reciben los 512 bytes de un bloque antes del próximo llamado a `ReadBlock`. De esta forma en la próxima llamada el ciclo de busy-waiting no se ejecutará y se copiará de inmediato el área escondida hacia el área del llamador.

```
/* Puertas del dispositivos */
char* puerta_control=(char*)0xe0000;
char* puerta_datos= puerta_control+1;

/* El buffer de bytes */
int nbytes=0;
char pbuff2[512];

void RutinaAtencionDispositivoX()
{
  pbuff2[nbytes++]= *puerta_datos;
  if (nbytes==512)
    InhibirIntDispositivoX();
}
```

```
void ReadBlock(char *pbuff)
{
  if (nbytes==0)
    HabilitarIntDispositivoX();
  while (nbytes<512)
    /* busy-waiting */;
  bcopy(pbuff,pbuff2,nbytes);
  nbytes=0;
  HabilitarIntDispositivoX();
}
```

Canales de E/S

Cuando se requiere transferir grandes bloques de datos entre memoria y dispositivos existen coprocesadores especializados para estos fines llamados canales de E/S o canales DMA (de Direct Memory Access).

Para iniciar este tipo de transferencias el sistema operativo indica a algún canal –a través de puertas en el espacio de direcciones– la dirección de memoria en donde comienza el bloque de datos, el tamaño del bloque, el sentido de la transferencia (memoria a dispositivo o viceversa) y la línea por la cual el dispositivo anunciará que hay un byte/palabra para transferir. Luego se comanda al canal para que inicie la transferencia.

De esta forma, el sistema operativo puede desligarse completamente de la transferencia, la que se realiza con mínima degradación del procesador al ejecutar los programas. Al terminar la transferencia, el canal interrumpe al sistema operativo para que determine las acciones a seguir.

La velocidad de transferencia entre memoria y dispositivos está limitada únicamente por la velocidad del bus que los separa (típicamente de 3 a 100 MBytes por segundo).

El siguiente código es una implementación de `ReadBlock` que usa la técnica de buffering (vista en la introducción).

```
/* Puertas del dispositivo */
char* puerta_control=(char*)0xe0000;
char* puerta_datos= puerta_control+1;

/* Puertas del canal */
char**puerta_canalX_inicio= ...;
int*  puerta_canalX_nbytes= ...;
int*  puerta_canalX_disp= ...;
char* puerta_canalX_control= ...;

/* El buffer de bytes */
int  leido= FALSE;
int  programado= FALSE;
char pbuff2[512];

int RutinaAtencionCanalX()
{
  leido= TRUE;
}

void ReadBlock(char *pbuff)
{
  if (!programado)
    ProgramarCanalX();

  while (!leido)
    /* busy-waiting */ ;
  bcopy(pbuff, pbuff2, 512);
  leido= FALSE;
  ProgramarCanalX();
}

void ProgramarCanalX()
{
  *puerta_CanalX_inicio= pbuff2;
  *puerta_CanalX_nbytes= 512;
  *puerta_CanalX_disp=
     <nro. linea DRQ del dispositivo>;
  *puerta_CanalX_control= LEER;
  HabilitarIntCanalX();
  programado= TRUE;
}
```

Observe que la rutina de atención atiende las interrupciones del canal y no las del dispositivo. Del mismo modo, son las interrupciones del canal las que se habilitan, no las del dispositivo. Esto se hace para que efectivamente se reciba una sola interrupción al final del bloque y no una interrupción por cada byte transferido.

3.1.3 Modo Dual

Un programa que corre en un sistema mono-usuario como un PC/DOS puede leer o escribir cualquier dirección en la memoria. Por lo tando puede alterar el código de DOS, modificar el vector de interrupciones o la memoria de video y comandar cualquiera de los dispositivos.

En cambio en un sistema multiprogramado un programa de un usuario no puede influir en el programa de otro usuario. Para implementar esto todos los micro-procesadores modernos (Motorola 68000 e Intel 286 en adelante) incorporan un *modo dual*:

- *Modo sistema*: Todas las instrucciones son válidas. Este modo también se llama modo privilegiado, monitor o supervisor.

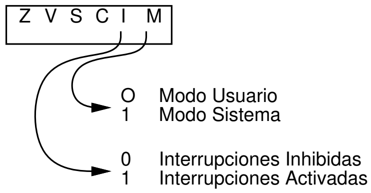

Figura 3.3: Palabra de estado del procesador

- *Modo usuario*: Se inhiben ciertas instrucciones consideradas *peligrosas* porque pueden influir en los programas de otros usuarios. Se dice que estas instrucciones son privilegiadas, porque sólo pueden ejecutarse en el modo privilegiado (modo sistema).

El núcleo de un sistema operativo siempre corre en el modo sistema mientras que los programas del usuario corren en modo usuario. Si un programa del usuario intenta ejecutar una de estas instrucciones se produce una interrupción (*trap*) que causa la invocación de una rutina de atención provista por el núcleo. Esta rutina de atención se ejecuta en modo sistema y tiene el poder como para abortar el programa que intentó ejecutar la instrucción inhibida.

El registro de estado del procesador indica el modo en que se trabaja (ver figura 3.3).

Ejemplos de instrucciones privilegiadas son la instrucción que inhibe las interrupciones y la instrucción que cambia el registro de estado.

Observe que este mecanismo no es suficiente para prohibir el acceso a los dispositivos puesto que estos se comandan con las mismas instrucciones de lectura y escritura en memoria.

3.1.4 Espacio de Direcciones Virtuales

Cada programa del usuario corre en su propio espacio de direcciones virtuales independiente. Es el núcleo –con ayuda del Hardware– el que se encarga de emular varios espacios de direcciones en un solo espacio de direcciones real del procesador.

Esto significa que cada vez que un programa especifica una dirección en donde leer o escribir una palabra, esa dirección está referida al espacio de direcciones virtuales de ese programa. Esa dirección no corresponde a la dirección real que ocupa la palabra en la memoria real del computador. Además la palabra de dirección virtual 100 de un programa en ejecución tiene una dirección real distinta de la palabra de dirección virtual 100 en otro programa.

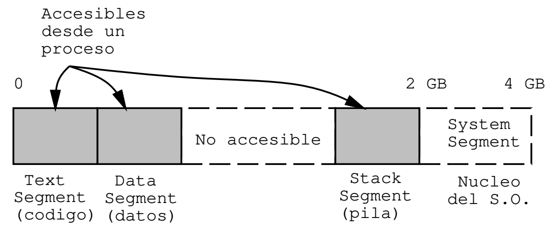

Figura 3.4: Espacio de direcciones virtuales de un programa en Unix

El Hardware del computador se encarga de traducir eficientemente las direcciones virtuales a direcciones reales en todo acceso a la memoria o dispositivos. Para ello, el núcleo le suministra los parámetros sobre cómo realizar esa traducción.

El espacio de direcciones típico de un programa en Unix es el que se muestra en la figura 3.4. Las características de este espacio son:

- No hay vector de interrupciones.
- No existen puertas de dispositivos en este espacio.
- Se distinguen 3 zonas accesibles por un programa en ejecución: estas son el segmento de código, el segmento de datos y el segmento de pila.
- El segmento de código sólo puede ser leído (no escrito).
- Estos segmentos son inaccesibles a partir de otros programas que corren en el mismo procesador.
- Los segmentos de datos y pila están separados por una zona inaccesible. Esta zona está reservada para la extensión de estos segmentos.
- Un programa puede hacer crecer el área de datos utilizando la primitiva `sbrk`.
- La pila crece hacia las direcciones inferiores y se extiende automáticamente si se desborda.
- Existe una área invisible para el programador llamada segmento del sistema que sólo el núcleo puede accesar en modo sistema.

De esta forma los programas del usuario no pueden ver los dispositivos, el vector de interrupciones, o las estructuras de datos que maneje internamente el núcleo. Este es el mecanismo de protección más importante que debe poseer un sistema operativo.

3.1.5 Cronómetro regresivo

El hardware del computador debe incluir un cronómetro regresivo o *timer*. Éste es un dispositivo que el núcleo puede comandar para que invoque una interrupción después de transcurrido un cierto tiempo. De esta forma el núcleo cuenta con un mecanismo para entregar el procesador a una aplicación por un tiempo acotado.

3.2 Arquitectura del Sistema Operativo

La organización de los sistemas operativos ha evolucionado desde los monitores residentes como DOS hasta los modernos sistemas multiproceso como Solaris. A continuación revisamos algunas de las componentes que debe incluir todo sistema operativo moderno.

3.2.1 El núcleo

El núcleo es la componente del sistema operativo que siempre está residente en la memoria real del computador. La función primordial del núcleo es transformar los recursos reales del computador en recursos estándares y cómodos de usar.

Es así como el núcleo transforma un procesador real con su memoria finita en un número prácticamente ilimitado de procesadores virtuales o *procesos*. Cada proceso dispone de su propio tiempo de CPU, una memoria extensible y mecanismos estándares para interactuar con los dispositivos, sin importar los detalles físicos de su implementación.

La API del núcleo

Una *API* (Interfaz de Programación de Aplicaciones) es el conjunto de servicios que ofrece un sistema a las aplicaciones usuarias de ese sistema. Las aplicaciones invocan estos servicios a través de llamadas a procedimientos. La API queda definida por lo tanto por los nombres de estos procedimientos, sus argumentos y el significado de cada uno de ellos.

El conjunto de servicios que ofrece el núcleo a los procesos se denomina la API del núcleo. Está formada por procedimientos pertenecientes al núcleo, pero que se invocan desde un proceso cualquiera. La invocación de uno de estos procedimientos es una *llamada al sistema*.

Ejemplos de llamadas al sistema en Unix son:

- Manejo de Procesos: creación (`fork`), destrucción (`kill`), término (`exit`), sincronización (`wait`), carga de un binario (`exec`).
- Manejo de memoria: extensión de la memoria de datos (`sbrk`).
- Manejo de archivos y dispositivos: `open`, `read`, `write` y `close`.

Estas llamadas se implementan usualmente con una instrucción de máquina que provoca una interrupción. Esta interrupción hace que el procesador real pase a modo sistema e invoque una rutina de atención perteneciente al núcleo y que ejecuta la llamada al sistema. Los argumentos de la llamada se pasan a través de los registros del procesador.

3.2.2 Los drivers para dispositivos

La operación de los dispositivos es altamente dependiente de su implementación. Es así como un disco SCSI se opera de una forma distinta de un disco IDE. Para independizar el código del núcleo de los variados mecanismos de interacción con los dispositivos, el núcleo define clases de dispositivos. Ejemplos de clases son disco, cinta, puerta de comunicación, interfaz de red, etc. Para cada clase se define una interfaz estándar para interactuar con cualquier dispositivo que pertenezca a la clase. Esta interfaz corresponde a las declaraciones de un conjunto de procedimientos no implementados.

Un driver es el código que implementa una interfaz estándar para interactuar con un dispositivo específico, como por ejemplo un disco SCSI. Este código es por lo tanto altamente dependiente de los discos SCSI y no funcionará con discos IDE. Sin embargo, el núcleo interactúa con este driver para discos SCSI de la misma forma que lo hace con el driver para discos IDE, es decir a través de la misma interfaz.

La visión que tiene el núcleo de un disco a través de un driver es la de un *arreglo* de bloques de 512 o 1024 bytes de tamaño fijo. El núcleo puede leer o escribir directamente cualquiera de estos bloques haciendo uso de la interfaz estándar de la clase disco.

Por otra parte, la visión que tiene el núcleo de una cinta es la de un conjunto de bloques de tamaño variable que sólo pueden leerse o grabarse en *secuencia*. También puede rebobinar esta cinta para volver a leerla o grabarla. Todo esto a través de la interfaz estándar de la clase cinta.

En Unix una aplicación puede accesar una partición de un disco en su formato nativo abriendo por ejemplo `/dev/sd0a`.

Es usual que los drivers estén siempre residentes en la memoria real y por lo tanto son parte del núcleo. Sin embargo la tendencia es que los drivers son módulos que se cargan dinámicamente si es necesario. También existen drivers que corren como un proceso –como cualquier aplicación– y por lo tanto corren en modo usuario (por ejemplo el servidor X de X-windows).

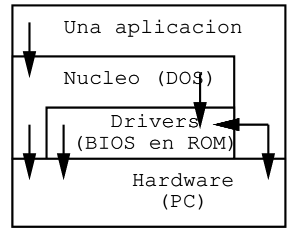

Figura 3.5: Organización del Software en un PC

3.2.3 El sistema de archivos

El sistema de archivos es la componente del sistema operativo que estructura un disco en una jerarquía de directorios y archivos. Conceptualmente multiplexa un disco de tamaño fijo en una jerarquía de discos de tamaño variable o archivos.

Dada esta equivalencia conceptual entre discos y archivos no es raro que ambos se manipulen con las mismas llamadas al sistema: `open`, `read`, `write`, `close` y `lseek` (esta última mueve la cabeza del disco hacia un bloque determinado).

Es usual que el sistema de archivos sea parte del núcleo. Por lo demás la motivación inicial de muchos sistemas operativos como Unix era el de ofrecer un sistema de archivos a un único proceso. Por algo DOS significa *Disk Operating System* y por ello es natural que forme parte del núcleo. Sin embargo hay sistemas operativos que ofrecen el sistema de archivos como parte de un proceso que no es parte del núcleo.

3.2.4 El intérprete de comandos

El intérprete de comando (o *shell*) se encarga de leer las órdenes interactivas del usuario y ejecutar los programas que el usuario indique.

Usualmente el intérprete de comandos es un proceso más del sistema operativo y no forma parte del núcleo. Por ejemplo Unix ofrece varios intérpretes de comandos (`sh`, `csh` y sus variantes). El intérprete de comandos de DOS se encuentra en `COMMAND.COM`.

3.2.5 Ejemplos de Sistemas Operativos

A continuación revisamos la estructura de algunos sistemas operativos.

DOS

En sus primeras 3 versiones, DOS era realmente un monitor residente que se situaba entre una aplicación y el Hardware/ROM de un PC (ver figura 3.5). Los objetivos del sistema no eran ambiciosos puesto que debía correr en computadores con poca memoria.

El núcleo de DOS implementaba un sistema de archivos jerárquico y da acceso a puertas seriales y paralelas. El sistema es estrictamente mono-proceso y la memoria se limita a los primeros 640 KBytes de la memoria descontando lo que ocupa el mismo DOS. Un programa puede llamar a la API de DOS a través de la instrucción de máquina `INT` que provoca una interrupción.

Dado que la Intel 8088 no ofrecía modo dual ni mecanismos para implementar espacios de direcciones virtuales, tanto DOS como el proceso en ejecución corren en el mismo modo y espacio de direcciones. Esto no es un problema en un sistema mono-usuario y mono-proceso y que por lo tanto no necesita un mecanismo de protección.

Los drivers no son parte de DOS, sino que vienen en la memoria ROM de un PC. Esta componente se llama BIOS (Basic Input Output System) y puede ser invocada directamente por la aplicación sin pasar por DOS. Lamentablemente la BIOS no incluye los drivers suficientemente generales para manejar apropiadamente los distintos dispositivos. Esto obliga a que las aplicaciones deban accesar directamente el hardware para operaciones tan simples como dibujar una línea en la pantalla.

Hoy en día DOS sí implementa en una forma rudimentaria múltiples procesos, cada uno en su propio espacio de direcciones. Sin embargo DOS debe permitir que estos manipulen directamente los dispositivos (sin hacerlo a través de DOS) para garantizar la compatibilidad con las antiguas aplicaciones. Por ello DOS no puede ofrecer verdadera protección entre procesos.

Linux, Berkeley Unix y Unix System V release 3.X

Todas estas variantes de Unix tienen estructura similar (ver figura 3.6). El núcleo incluye los drivers y el sistema de archivos.

Además de los servicios de la API del núcleo, Unix ofrece muchísimos otros servicios a través de procesos *demonios*, que corresponden a procesos que siempre

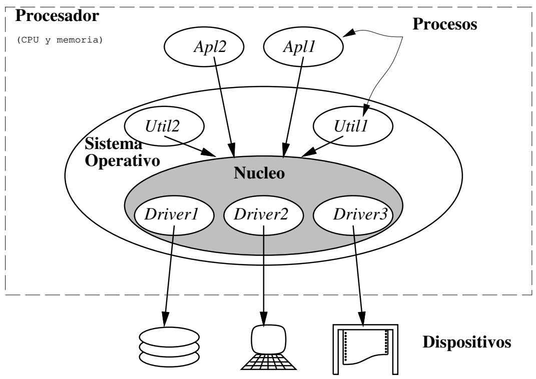

Figura 3.6: Estructura de un Sistema Unix tradicional.

están corriendo. Entre estos se cuentan los servicios de spooling, mail, news, www, etc.

Chorus y Mach

El núcleo de estos sistemas operativos está diseñado con el enfoque minimalista: sólo ofrece el servicio de manejo de procesos e intercomunicación entre procesos a través de una API no estándar. El sistema de archivos, los drivers y cualquier otro servicio queda fuera del núcleo y son ofrecidos por medio de procesos.

Ambos sistemas son compatibles con Unix System V gracias a un proceso que implementa la API estándar de Unix. Las llamadas de sistema de un proceso Unix se implementan enviando mensajes al proceso de la API que ejecuta la llamada y devuelve el resultado en otro mensaje.

El interés de este tipo de sistemas es que pueden implementar varias APIs no sólo la de Unix, sino que también la de Windows o la de Macintosh, etc., cada una de estas APIs se implementan en procesos independientes.

Capítulo 4

Administración de Procesos

Una de las actividades fundamentales del núcleo del sistema operativo es implementar procesos. Cada proceso es un procesador virtual en donde se ejecuta una aplicación o una herramienta del sistema operativo. El núcleo debe encargarse entonces de administrar los recursos del hardware del computador para que sean asignados convenientemente a los procesos.

En este capítulo examinaremos cómo el núcleo asigna el o los procesadores reales a los procesos. En una primera etapa supondremos que el hardware sólo ofrece un procesador real. Más tarde generalizaremos a varios procesadores.

4.1 Scheduling de Procesos

En la mayoría de los sistemas computacionales existe un solo procesador real. Por lo tanto el núcleo debe asignar el procesador por turnos a los numerosos procesos que pueden estar activos. Hay distintas estrategias para asignar estos turnos, dependiendo del objetivo que se persiga.

Por ejemplo en un sistema de multiprogramación se busca maximizar el tiempo de uso del procesador, mientras que en un sistema de tiempo compartido se busca atender en forma expedita a muchos usuarios que trabajan interactivamente.

La asignación *estratégica* del procesador a los procesos es lo que se denomina *scheduling de procesos*. Es estratégica porque se intenta lograr algún objetivo particular como alguno de los que se mencionó más arriba y para ello se usan estrategias que pueden funcionar muy bien en determinados sistemas, pero muy mal en otros. La componente del núcleo que se encarga de esta labor se denomina *scheduler* del procesador.

Además de scheduling de procesos, también se realiza scheduling de los accesos a disco y para ello existe un scheduler asociado a cada disco. Este scheduler ordena estratégicamente los mútiples accesos a disco de varios procesos con el fin de minimizar el tiempo de acceso.

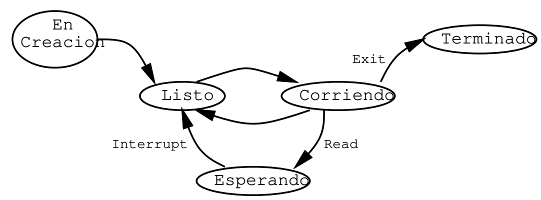

Figura 4.1: Estados de un procesos

Para abreviar la notación, cuando se hable de "el scheduler" se subentenderá que se trata del scheduler del procesador.

4.1.1 Estados de un proceso

Mientras un proceso se ejecuta puede pasar por distintos estados. Estos estados se aprecian en la figura 4.1.

- **En creación** : el núcleo está obteniendo los recursos que necesita el proceso para poder correr, como por ejemplo memoria o disco.
- **Corriendo** : El proceso está en poseción del procesador, el que ejecuta sus instrucciones.
- **Esperando** : El proceso espera que se lea un sector del disco, que llegue un mensaje de otro proceso, que transcurra un intervalo de tiempo, que termine otro proceso, etc.
- **Listo** : El proceso está activo pero no está en poseción del procesador.
- **Terminado** : El proceso terminó su ejecución, pero sigue existiendo para que otros procesos puedan determinar que terminó.

Un proceso pasa de un estado a otro constantemente y varias veces por segundo. Por ejemplo cuando un proceso está **corriendo** el scheduler puede quitarle el procesador para entregárselo a otro proceso. En este caso el primer proceso queda **listo** para ejecutarse. Es decir en cualquier momento, el procesador puede entregarle nuevamente el procesador y quedar **corriendo**. De este estado el proceso puede leer un comando del terminal y por lo tando quedar en **espera** de que el usuario invoque alguna acción.

Desde luego el número exacto de estados depende del sistema. Por ejemplo en nSystem un proceso puede pasar por los siguientes estados :

- READY : Elegible por el scheduler o corriendo.
- ZOMBIE : La tarea llamó `nExitTask` y espera `nWaitTask`.
- WAIT_TASK : La tarea espera el final de otra tarea.
- WAIT_REPLY : La tarea hizo `nSend` y espera `nReply`.
- WAIT_SEND : La tarea hizo `nReceive` y espera `nSend`.
- WAIT_READ : La tarea está bloqueada en un `read`.
- WAIT_WRITE : La tarea está bloqueada en un `write`.
- otros.

4.1.2 El descriptor de proceso

El descriptor de proceso es la estructura de datos que utiliza el núcleo para mantener toda la información asociada a un proceso. Ella contiene :

- El estado del proceso : En creación, listo, corriendo, etc.
- Registros del procesador : Aquí se guardan los valores del contador de programa, del puntero a la pila y de los registros del procesador real cuando el proceso abandonó el estado **corriendo**.
- Información para el scheduler : Por ejemplo la prioridad del proceso, punteros de enlace cuando el descriptor se encuentra en alguna cola, etc.
- Asignación de recursos : Memoria asignada, archivos abiertos, espacio de *swapping*, etc.
- Contabilización de uso de recurso : Tiempo de procesador consumido y otros.

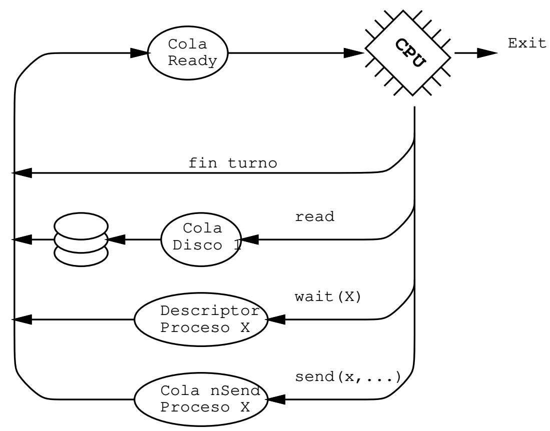

Figura 4.3: Tránsito de un proceso por distintas colas

El núcleo posee algún mecanismo para obtener el descriptor de proceso a partir del *identificador del proceso*, que es el que conocen los procesos.

La figura 4.2 muestra el descriptor que usa nSystem. En nSystem el identificador de tarea es un puntero a este descriptor. Sin embargo, por razones de encapsulación este puntero se declara del tipo `void *` en `nSystem.h`.

4.1.3 Colas de Scheduling

Durante la ejecución, un proceso pasa por numerosas colas a la espera de algún evento. Esto se puede observar en la figura 4.3.

Mientras un proceso espera la obtención del procesador, este proceso permanece en una cola del scheduler del procesador. Esta cola puede estar organizada en una lista enlazada simple (cada descriptor de proceso tiene un puntero al próximo descriptor en la cola).

Además de la cola del scheduler del procesador existen la colas de scheduling de disco. El proceso permanece en estas colas cuando realiza E/S.

Cuando un proceso envía un mensaje síncrono a otro proceso que no está preparado para recibirlo, el primer proceso queda en una cola de recepción de mensajes en el descriptor del proceso receptor.

Por último el proceso puede quedar en una cola del reloj regresivo, a la espera de que transcurra un instante de tiempo.

4.1.4 Cambio de contexto

Cambio de contexto (context switch, task switch o process switch) es la acción que efectúa el scheduler cuando transfiere el procesador de un proceso a otro.

```
typedef struct Task /* Descriptor de una tarea */
{
  int status;       /* Estado de la tarea (READY, ZOMBIE ...) */
  char *taskname;   /* Util para hacer debugging */

  SP sp;            /* El stack pointer cuando esta suspendida */
  SP stack;         /* El stack */

  struct Task *next_task;   /* Se usa cuando esta en una cola */
  void *queue;              /* Debugging */

  /* Para el nExitTask y nWaitTask */
  int  rc;                  /* codigo de retorno de la tarea  */
  struct Task *wait_task;   /* La tarea que espera un nExitTask */

  /* Para nSend, nReceive y nReply */
  struct Queue *send_queue; /* cola de emisores en espera de esta tarea */
  union { void *msg; int rc; } send; /* sirve para intercambio de info */
  int wake_time;            /* Tiempo maximo de espera de un nReceive */
  int in_wait_squeue;       /* 1 si esta tarea se encuentra en wait_squeue */
}
  *nTask;
```

Figura 4.2: El descriptor de proceso en nSystem

Para realizar el cambio de contexto, el scheduler debe realizar diversas labores que detallamos a continuación :

i. Resguardar/restaurar Registros.

   Cuando un proceso está **corriendo**, se utiliza el contador de programa, el puntero a la pila y los registros del procesador real. Al hacer el cambio de contexto, el nuevo proceso que toma el control del procesador usará esos mismos registros. Por ello es necesario resguardar los registros del proceso saliente en su descriptor de proceso. De igual forma, el descriptor del proceso entrante mantiene los valores que contenían los registros en el momento en que perdió el procesador. Estos valores deben ser restaurados en los registros reales para que el proceso funcione correctamente.

ii. Cambiar espacio de direcciones virtuales.

   El espacio de direcciones del proceso saliente no es el mismo del espacio de direcciones del proceso entrante. En el capítulo sobre administración de memoria veremos como se logra el cambio de espacio de direcciones virtuales.

iii. Contabilización de uso de procesador.

   Usualmente, el núcleo se encarga de contabilizar el tiempo de procesador que utiliza cada proceso. Una variable en el descriptor de proceso indica el tiempo acumulado. En cada cambio de contexto el núcleo consulta un cronómetro que ofrece el hardware de cada computador. El scheduler suma el tiempo transcurrido desde el último cambio contexto al tiempo acumulado del proceso saliente.

Estas labores del scheduler consumen algo de tiempo de procesador y por lo tanto son sobrecosto puro. Cada cambio de contexto puede significar de 1 microsegundo a un milisegundo según el sistema operativo y la ayuda que ofrezca el hardware para realizar eficientemente el cambio de contexto.

Normalmente la componente más cara en tiempo de procesador es la del cambio de espacio de direcciones virtuales. Ahí radica el origen del nombre de procesos livianos. Como los procesos livianos comparten el mismo espacio de direcciones no es necesario cambiarlo durante un cambio de contexto. Por lo tanto pueden ser implementados más eficientemente. En este caso el peso de un proceso corresponde al tiempo de procesador que se consume al realizar un cambio de contexto. A mayor tiempo, mayor peso.

Interrupciones vs. cambio de contexto

Es importante hacer notar que cuando se produce una interrupción, el hecho de invocar la rutina de atención

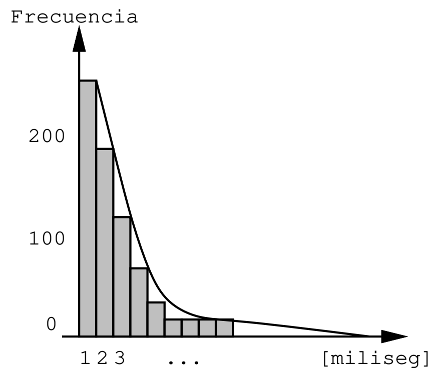

Figura 4.4: Histograma de la duración de la ráfagas de CPU. El primer tramo indica el número de ráfagas que duran entre 0 y 1 milisegundo, el segundo tramo indica las ráfagas de 1 a 2 milisegundos, etc.

de la interrupción *no es un cambio de contexto*. Esto se debe a que el código de la rutina de atención es parte del núcleo y por lo tanto no pertenece a ningún proceso en particular. Para que haya un cambio de contexto *es necesario* que se pase de la ejecución del código de un proceso a la ejecución del código de otro proceso.

Sin embargo muchos cambios de contexto ocurren durante una interrupción, por ejemplo cuando interrumpe el reloj regresivo. En otros casos, la interrupción no se traduce en un cambio de contexto. Por ejemplo si un disco interrumpe, esto puede significar que el proceso que esperaba esta interrupción se coloca en la cola del scheduler pero no se le transfiere de inmediato el procesador, se continúa ejecutando el proceso interrumpido.

4.1.5 Ráfagas de CPU

Una ráfaga de CPU es una secuencia de instrucciones que puede ejecutar un proceso sin pasar a un estado de espera. Es decir el proceso *no espera* un acceso al disco, la recepción de un mensaje, el término de un proceso, etc.

La ejecución de un proceso consta de innumerables ráfagas de CPU. Empíricamente se ha determinado que la mayoría de las ráfagas toman pocas instrucciones. Esto se observa gráficamente en la figura 4.4.

El scheduler puede aprovecharse de este hecho empírico para disminuir el sobrecosto de los cambios de contexto. En efecto, al final de una ráfaga el cambio de contexto es inevitable porque el proceso en curso pasa a un modo de espera. Al contrario, los cambios de contexto en medio de una ráfaga son evitables y por lo tanto son sobrecosto puro. Un buen scheduler tratará de evitar los cambios de contexto en medio de una ráfaga corta y solo introducirá cambios de contexto en ráfagas prolongadas, que son las menos.

4.2 Estrategias de scheduling de procesos

Como dijimos anteriormente el scheduling de procesos intenta lograr algún objetivo particular como maximizar la utilización del procesador o atender en forma expedita a usuarios interactivos. Para lograr estos objetivos se usan diversas estrategias que tienen sus ventajas y desventajas. A continuación se explican las estrategias más utilizadas.

4.2.1 First Come First Served o FCFS

Ésta es una estrategia para procesos non-preemptive. En ella se atiende las ráfagas en forma ininterrumpida en el mismo orden en que llegan a la cola de procesos, es decir en orden FIFO.

Esta estrategia tiene la ventaja de ser muy simple de implementar. Sin embargo tiene varias desventajas :

i. Pésimo comportamiento en sistemas de tiempo compartido. Por lo tanto sólo se usa en sistemas batch.

ii. El tiempo de despacho depende del orden de llegada.

   Por ejemplo supongamos que llegan 3 ráfagas de duración 24, 3 y 3.

   Si el orden de llegada es 24, 3 y 3, el tiempo promedio de despacho es :
   $$\frac{24+27+30}{3} = 27$$

   Si el orden de llegada es 3, 3 y 24, el promedio es :
   $$\frac{3+6+30}{3} = 13$$

iii. Los procesos intensivos en tiempo de CPU tienden a procesarse antes que los intensivos en E/S. Por lo tanto, al comienzo el cuello de botella es la CPU mientras que al final son los canales de E/S y la CPU se utiliza poco. Es decir esta estrategia no maximiza el tiempo de utilización de la CPU.

Observe que aun cuando esta estrategia es non-preemptive esto no impide que en medio de una ráfaga de CPU pueda ocurrir una interrupción de E/S que implique que un proceso en espera pase al estado listo para correr.

4.2.2 Shortest Job First o SJF

En principio si se atiende primero las ráfagas más breves se minimiza el tiempo promedio de despacho. Por ejemplo si en la cola de procesos hay ráfagas que van a durar 5, 2, 20 y 1, estas ráfagas deberían ser atendidas en el orden 1, 2, 5 y 20. El tiempo promedio de despacho sería:

$$\frac{1+3+8+29}{4} = 10$$

Se demuestra que este promedio es inferior a cualquier otra permutación en el orden de atención de las ráfagas. Pero desde luego el problema es que el scheduler no puede predecir el tiempo que tomará una ráfaga de CPU.

Sin embargo es posible calcular un predictor del tiempo de duración de una ráfaga a partir de la duración de las ráfagas anteriores. En efecto, se ha comprobado empíricamente que la duración de las ráfagas tiende a repetirse. El predictor que se utiliza usualmente es:

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

En donde:

τ_{n+1}^P :	El predictor de la próxima ráfaga
t_n^P :	Duración de la ráfaga anterior
τ_n^P :	Predictor de la ráfaga anterior
α :	Ponderador para la última ráfaga

Típicamente $\alpha = 0.5$. Con esta fórmula se pondera especialmente la duración de las últimas ráfagas, ya que al expandir esta fórmula se obtiene:

$$\tau_{n+1}^P = \alpha t_n^P + (1-\alpha)\alpha t_{n-1}^P + (1-\alpha)\alpha^2 t_{n-2}^P + ...$$

De este modo cuando hay varios procesos en la cola de procesos listos para correr, se escoge aquel que tenga el menor predictor.

SJF puede ser non-preemptive y preemptive. En el caso non-preemptive una ráfaga se ejecuta completamente, aún cuando dure mucho más que su predictor. Desde luego, la variante non-preemptive se usa sólo en sistemas batch.

En el caso preemptive se establece una cota al tiempo de asignación del procesador. Por ejemplo la cota podría ser el segundo predictor más bajo entre los procesos listos para correr. También hay que considerar el caso en que mientras se ejecuta una ráfaga, llega una nueva ráfaga con un predictor menor. En este caso, la ráfaga que llega debería tomar el procesador.

4.2.3 Colas con prioridad

En esta estrategia se asocia a cada proceso una prioridad. El scheduler debe velar porque en todo momento el proceso que está corriendo es aquel que tiene la mayor prioridad entre los procesos que están listos para correr. La prioridad puede ser fija o puede ser calculada. Por ejemplo SJF es un caso particular de scheduling con prioridad en donde la prioridad se calcula como $p = -\tau_{n+1}$.

El problema de esta estrategia es que puede provocar hambruna (starvation), ya que lo procesos con baja prioridad podrían nunca llegar a ejecutarse porque siempre hay un proceso listo con mayor prioridad.

Aging

Aging es una variante de la estrategia de prioridades que resuelve el problema de la hambruna. Esta estrategia consiste en aumentar cada cierto tiempo la prioridad de todos los procesos que se encuentran listos para correr. Por ejemplo, cada un segundo se puede aumentar en un punto la prioridad de los procesos que se encuentran en la cola *ready*. Cuando el scheduler realiza un cambio de contexto el proceso saliente recupera su prioridad original (sin las bonificaciones).

Desde este modo, los procesos que han permanecido mucho tiempo en espera del procesador, tarde o temprano ganarán la prioridad suficiente para recibirlo.

4.2.4 Round-Robin

Es una estrategia muy usada en sistemas de tiempo compartido. En Round-Robin los procesos *listos para correr* se organizan en una lista circular. El scheduler se pasea por la cola dando *tajadas* pequeñas de tiempo de CPU (*time slicing*): de 10 a 100 milisegundos. Esta estrategia se aprecia en la figura 4.5.

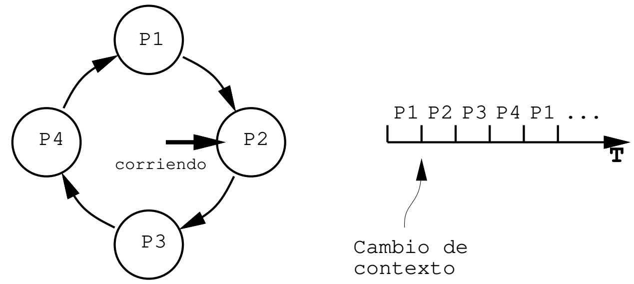

Figura 4.5: Round-Robin

Esta estrategia minimiza el *tiempo de respuesta* en los sistemas interactivos. El tiempo de respuesta es el tiempo que transcurre entre que un usuario ingresa un comando interactivo hasta el momento en que el comando despliega su primer resultado. No se considera el tiempo de despliegue del resultado completo.

Esto se debe a que para un usuario interactivo es mucho más molesto que un comando no dé ninguna señal de avance que un comando que es lento para desplegar sus resultados. Por esta razón el tiempo de respuesta sólo considera el tiempo que demora un comando en dar su primer señal de avance.

Implementación

Esta estrategia se implementa usando una cola FIFO con todos los procesos que están listos para correr. Al final de cada tajada el scheduler coloca el proceso que se ejecutaba al final de la cola y extrae el que se encuentra en primer lugar. El proceso extraído recibe una nueva tajada de CPU. Para implementar la tajada, el scheduler programa una interrupción del reloj regresivo de modo que se invoque una rutina de atención (que pertenece al scheduler) dentro del tiempo que dura la tajada.

Si un proceso termina su ráfaga sin agotar su tajada (porque pasa a un modo de espera), el scheduler extrae el próximo proceso de la cola y le da una nueva tajada completa. El proceso que terminó su ráfaga *no se coloca* al final de la cola del scheduler. Este proceso queda en alguna estructura de datos esperando algún evento.

Tamaño de tajada

Una primera decisión que hay que tomar es qué tamaño de tajada asignar. Si la tajada es muy grande la estrategia degenera en FCFS con todos sus problemas. Al contrario si la tajada es muy pequeña el sobrecosto en cambios de contexto es demasiado alto.

Además si la tajada es muy pequeña aumenta el tiempo promedio de despacho. En efecto, si se tienen dos ráfagas de duración t. El tiempo medio de despacho de ambas ráfagas en FCFS será:

$$\overline{T} = \frac{t+2t}{2} = \frac{3}{2}t$$

En cambio en RR con tajada mucho más pequeña que t, ambas ráfagas serán despachadas prácticamente al mismo tiempo en $2t$, por lo que el tiempo medio de despacho será:

$$\overline{T} \rightarrow 2t$$

Una regla empírica que da buenos resultados en Round-Robin es que se fija el tamaño de la tajada de modo que el 80% de las ráfagas deben durar menos que la tajada y por lo tanto se ejecutan sin cambios de contexto de por medio.

Llegada de procesos

El segundo problema que hay que resolver es donde colocar los procesos que pasan de un estado de espera al estado listo para correr. Es decir en que lugar agregar estos procesos en la lista circular de Round-Robin (ver figura 4.6).

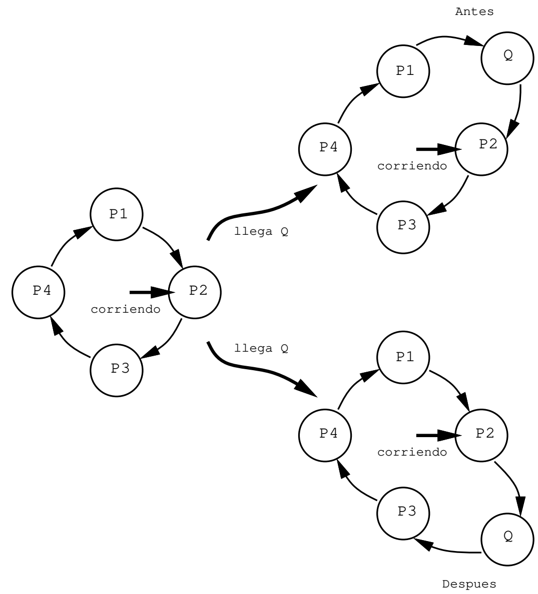

Figura 4.6: Inserción de procesos en Round-Robin

Cuando un proceso llega, las dos únicas posibilidades son agregarlo antes del proceso en ejecución o agregarlo después.

La primera posibilidad se implementa agregando el proceso que llega al final de la cola del scheduler. Este proceso deberá esperar una vuelta completa antes de que lo toque su tajada. Esto tenderá a desfavorecer a los procesos intensivos en E/S debido a que sus ráfagas terminan mucho antes que la tajada, teniendo que esperar la vuelta completa después de cada operación de E/S. En cambio, los procesos intensivos en CPU agotarán completamente su tajada antes de tener que esperar la vuelta completa.

En la segunda posibilidad el proceso que llega se agrega en primer lugar en la cola del scheduler (es decir la cola ya no es siempre FIFO). Esta variante tiene un comportamiento más errático. Veamos qué sucede en los siguientes casos:

- Hay dos procesos, uno intensivo en CPU que siempre está listo para correr y el otro es un proceso intensivo en E/S con ráfagas cortas separadas por pequeños intervalos de espera. En este caso el proceso de E/S también será desfavorecido, ya que cuando este proceso llega, la cola siempre está vacía y da lo mismo agregarlo al principio o al final. La situación es idéntica a la de la estrategia anterior.

- Hay muchos procesos intensivos en E/S y muchos procesos intensivos en CPU. En este caso los procesos de E/S pueden causar hambruna a los procesos de CPU, pues los procesos de E/S terminan su ráfaga antes que su tajada y después de un eventualmente corto período de espera quedan en primer lugar en la cola. Si hay suficientes procesos de E/S como para ocupar el 100% de la CPU, los procesos intensivos en CPU nunca recibirán su tajada.

4.3 Jerarquías de Scheduling

Normalmente los sistemas operativos implementan una jerarquía de scheduling de procesos. Esto se debe a que, primero, la CPU no es el único recurso que hay que administrar, la memoria también es escasa y hay que administrarla; y segundo, no todos los procesos tienen la misma urgencia, por lo que es conveniente que los procesos no urgentes se pospongan hasta que el computador se encuentre más desocupado.

Es así como el scheduling se podría organizar en 3 niveles :

- Scheduling de corto plazo.

  En este nivel se administra sólo la CPU usando las estrategias vistas en la sección anterior. Esta administración la realiza el núcleo.

- Scheduling de mediano plazo.

  En este nivel se administra la memoria. Cuando la memoria disponible no es suficiente se llevan procesos a disco. Esta transferencia a disco toma mucho más tiempo que un traspaso de la CPU (0.1 a 1 segundo vs. menos de un milisegundo). Por lo tanto las transferencias a disco deben ser mucho menos frecuentes que los cambios de contexto. Por esta razón se habla de mediano plazo. Este nivel de scheduling también se realiza en el núcleo.

- Scheduling de largo plazo.

  Este nivel está orientado al procesamiento batch. Los procesos batch o jobs no son urgentes, por lo que su lanzamiento puede ser postergado si el computador está sobrecargado. No todos los sistemas operativos implementan este nivel de scheduling y si lo hacen, lo realizan fuera del núcleo a través de procesos "demonios".

  En este nivel hay una cola de jobs esperando ser lanzados. El scheduler lanza jobs sólo en la medida que el procesador tiene capacidad disponible.

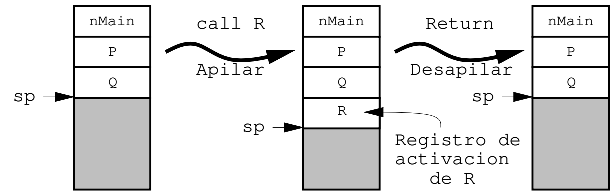

Figura 4.8: Llamado y retorno de procedimientos usando una pila

  Una vez que un job fue lanzado, éste se administra sólo en los niveles de corto y mediano plazo. Es decir que el scheduler de largo plazo no tiene ninguna incidencia sobre los jobs ya lanzados.

4.4 Administración de procesos en nSystem

El nSystem es un sistema de procesos livianos o nano tareas que corren en un solo proceso pesado de Unix. Las tareas de nSystem se implementan multiplexando el tiempo de CPU del proceso Unix en tajadas. Esto se puede hacer gracias a que un proceso Unix tiene asociado su propio reloj regresivo, un mecanismo de interrupciones (señales), E/S no bloqueante y todas las características del hardware que se necesitan para implementar multiprogramación en un solo procesador, con la salvedad de que este hardware es emulado en software por Unix.

La figura 4.7 muestra la organización de nSystem en módulos. nSystem es una biblioteca de procedimientos que se sitúa entre la aplicación (el código del programador) y la API de Unix.

Los procedimientos superiores corresponden a la API de nSystem y son por lo tanto invocados por la aplicación. Los procedimientos de más abajo son procedimientos internos que se requieren en la implementación de la API. El nombre que aparece verticalmente es el archivo en donde se implementa un módulo en particular. Las flechas indican que el procedimiento desde donde se origina la flecha necesita invocar el procedimiento de destino de la flecha.

4.4.1 La pila

Cada nano tarea tiene su propia pila. Los registros de activación de los procedimientos que se invocan en una tarea se apilan y desapilan en esta estructura de datos (ver figura 4.8). Un registro del procesador, denominado `sp`, apunta hacia el tope de la pila.

La pila es de tamaño fijo, por lo que una recursión demasiado profunda puede desbordarla. El nSystem

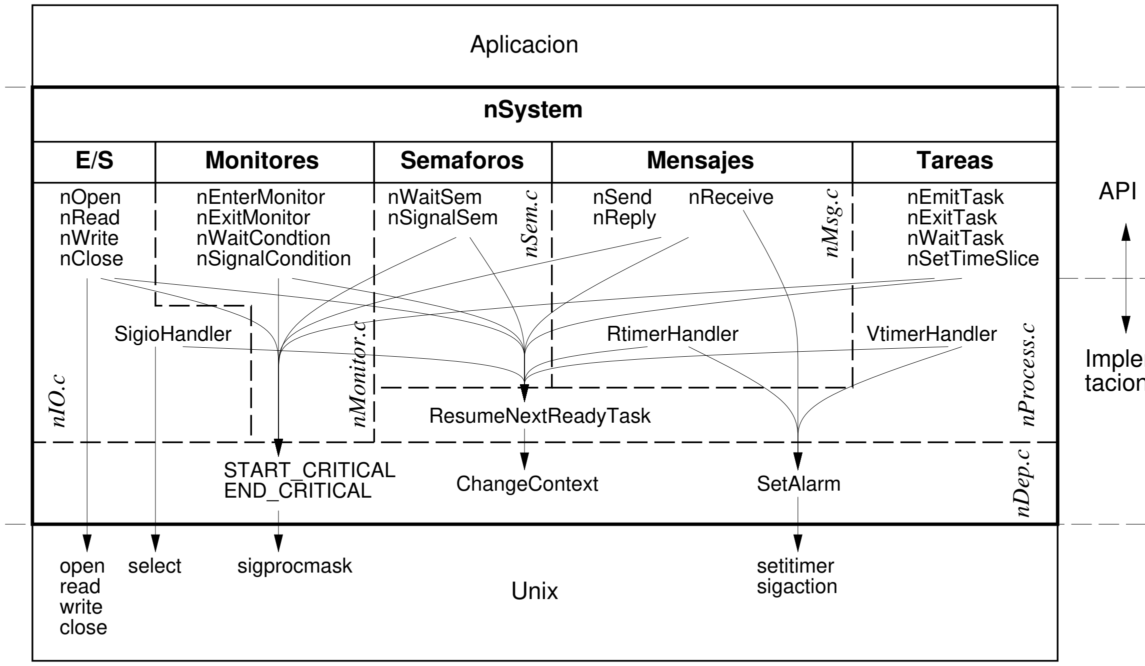

Figura 4.7: Organización de nSystem

chequea el posible desborde de la pila en cada cambio de contexto. De ocurrir un desborde el nSystem termina con un mensaje explicativo. Sin embargo si no hay cambios de contexto no puede detectar el desborde. En este caso, el nSystem podría terminar en un *segmentation fault* o *bus error*, sin mayor explicación de la causa de la caída.

4.4.2 El cambio de contexto

En nSystem nunca hay más de una nano tarea corriendo realmente. El registro `sp` del procesador apunta al tope de la pila de la tarea que está corriendo.

El cambio de contexto en nSystem lo realiza el scheduler llamando al procedimiento :

```
void ChangeContext(nTask out_task, nTask in_task);
```

Este procedimiento realiza las siguientes actividades (ver figura 4.9) :

i. Resguarda los registros del procesador en el tope de la pila de la tarea saliente (`out_task`).

ii. Guarda `sp` en el descriptor de esta tarea saliente (`out_task->sp`).

iii. Rescata el puntero al tope de la pila de la tarea entrante a partir de su descriptor de proceso (`in_task->sp`).

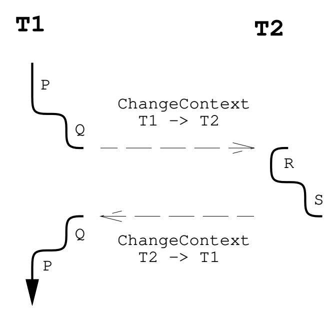

Figura 4.9: Cambio de pila durante un cambio de contexto.

iv. Restaura los registros de la tarea entrante que se encuentran en el tope de la pila.

v. Retoma la ejecución a partir del llamado a `ChangeContext` de la tarea entrante.

Es decir que el llamado a `ChangeContext` de la tarea saliente no retorna hasta que se haga un nuevo cambio hacia aquella tarea (ver figura 4.10).

Observe que las labores que realiza `ChangeContext` no pueden llevarse a cabo en C y por lo tanto parte de este procedimiento está escrito en *assembler* y es completamente dependiente de la arquitectura del procesador. Sin embargo, este cambio de contexto *no requiere una llamada al sistema*, el cambio de contexto se realiza sólo con instrucciones del modo usuario

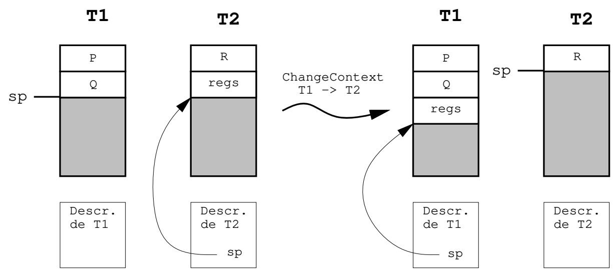

Figura 4.10: Cambio de contexto en nSystem.

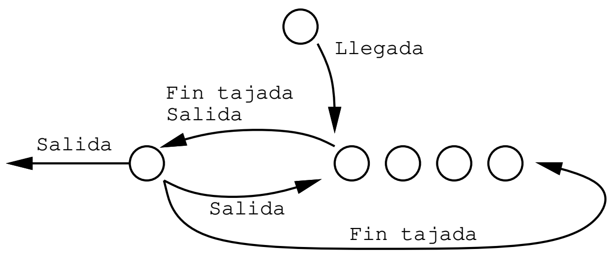

Figura 4.11: Scheduling en nSystem

en el proceso Unix en donde corre el nSystem.

4.4.3 Scheduling en nSystem

La estrategia de scheduling de nSystem podría denominarse como *preemptive last come first served* o PLCFS. Las ráfagas son atendidas en orden LIFO, es decir que las tareas que llegan al scheduler toman el control de procesador de inmediato y la tarea que se ejecutaba en ese instante se coloca en primer lugar en la cola del scheduler. Para evitar que una tarea acapare indefinidamente el procesador, cada t milisegundos el scheduler le quita el procesador a la tarea en ejecución, cualquiera sea esta tarea, y la coloca al final de una cola.

Esta estrategia se implementa usando una cola dual, es decir en donde se pueden agregar tareas ya sea al final como al comienzo. Cuando una tarea que estaba en estado de espera (por ejemplo a la espera de una operación de E/S) pasa al modo lista para ejecutarse (producto de una interrupción de E/S) el scheduler coloca la tarea en ejecución al comienzo de la cola dual y retoma de inmediato la tarea que llega (ver figura 4.11).

El cambio de tarea cada t milisegundos se implementa programando una señal que se invoca cada t milisegundos de tiempo virtual del proceso Unix. En esta interrupción el scheduler coloca la tarea en ejecución al final de la cola dual, extrae la tarea que se encuentra en primer lugar y la retoma.

Para esta señal se usa el tiempo virtual del proceso Unix, es decir el tiempo de uso del procesador real por parte del proceso Unix sin contar el tiempo en que el procesador estaba en poseción de otro proceso Unix. En efecto, cambiar de tarea cada t milisegundos de tiempo real no tendría sentido, pues durante ese intervalo podría haber sucedido que el proceso Unix en donde corre nSystem no hubiese tenido el procesador real y por lo tanto la tarea en ejecución no pudo avanzar tampoco.

Scheduling simplificado

La siguiente es una versión simplificada del código del scheduler. En ella no se consideran ciertas condiciones de borde que obscurecen innecesariamente el código. Estas condiciones son el caso en que no hay tareas listas para correr y el caso en que el scheduling es non-preemptive.

El scheduler mantiene las siguientes variables :

`nTask`	`current_task`	La tarea en ejecución
`Queue`	`ready_queue`	La cola dual
`int`	`current_slice`	El intervalo entre interrupciones

El scheduler retoma una tarea con el siguiente procedimiento :

```
ResumeNextReadyTask()
{
  nTask this_task= current_task;
  nTask next_task= GetTask(ready_queue);
  ChangeContext(this_task, next_task);
  current_task= this_task; /* (1) */
}
```

Observe que si T era la tarea que corría cuando se invocó `ResumeNextReadyTask`, el llamado a `ChangeContext` no retornará hasta que se retome T. Es decir cuando se invoque nuevamente `ResumeNextReadyTask` y T aparezca en primer lugar en la cola `ready_queue`. Mientras tanto, el nSystem ejecutó otras tareas y realizó eventualmente otros cambios de contexto. Por esta razón en (1) se asigna T a la tarea actual y no `next_task`.

En cada señal periódica se invoca el siguiente procedimiento :

```
VtimerHandler()
{
  /* Programa la siguiente interrupcion */
  SetAlarm(VIRTUALTIMER, current_slice,
           VtimerHandler);
  /* Coloca la tarea actual al final
     de la cola */
  PutTask(ready_queue, current_task);
  /* Retoma la primera tarea en la cola */
```

```
  ResumeNextReadyTask();
}
```

Para implementar `SetAlarm` se usaron las llamadas al sistema `setitimer` y `sigaction`.

Note que la tarea que recibe el procesador estaba bloqueada en el llamado a `ChangeContext` de `ResumeNextReadyTask` y por lo tanto es ella la que actualizará la variable `current_task` con su propio identificador. En nSystem todos los cambios de contexto se realizan a través de `ResumeNextReadyTask`. La única excepción es el cambio de contexto que se realiza al crear una tarea.

4.4.4 Entrada/Salida no bloqueante

En Unix se puede colocar un descriptor de archivo (`fd`) en un modo no bloqueante y hacer que se invoque la señal `SIGIO` cuando haya algo para leer en ese descriptor. Esto significa que `read(fd,...)` no se bloquea cuando no hay nada que leer en `fd`, sino que retorna -1. Esto se hace en Unix con la llamada al sistema `fcntl`.

Esto es importante porque de no programar los descriptores en modo bloqueante, cuando una tarea intente leer un descriptor se bloquearán todas las tareas hasta que haya algo para leer en ese descriptor. Esto es inadmisible en un sistema multiprogramado.

Una primera implementación de `nRead` es :

```
int nRead(int fd, char *buf, int nbyte)
{ /* Intentamos leer */
  int rc= read(fd, buf, nbyte);
  while (rc<0 && errno==EAGAIN)
  { /* No hay nada disponible */
    AddWaitingTask(fd, current_task);
    current_task->status= WAIT_READ;
    /* Pasamos a la proxima tarea ready */
    ResumeNextReadyTask();
    /* Ahora si que deberia funcionar */
    rc= read(fd, buf, nbyte);
  }
  return rc;
}
```

El valor `EAGAIN` en `errno` indica que no hay nada para leer y que `read` debería ser reinvocado más tarde. Unix informa al proceso de nSystem que hay algo para leer en algún descriptor invocando la señal `SIGIO`. El procedimiento que atiende esta señal es :

```
SigioHandler()
{
  PushTask(current_task, ready_queue);
  ... preparar argumentos para select ...
  select(...);
  ... por cada tarea task que puede
      avanzar:
      PushTask(task, ready_queue);
  ...
  ResumeNextReadyTask();
}
```

El llamado al sistema `select` determina qué descriptores de archivo tienen datos para leer o en qué descriptores se puede escribir sin llenar los buffers internos de Unix que obligarían a bloquear el proceso Unix. En el código que sigue al `select` se busca si hay tareas bloqueadas a la espera de alguno de los descriptores cuya lectura/escritura no causará un bloqueo temporal de nSystem. Estas tareas quedarán antes en la cola que la tarea en ejecución cuando se invocó `SigioHandler`, porque el scheduling es preemptive LCFS.

4.4.5 Implementación de secciones críticas en nSystem

El código de `nRead` puede funcionar mal esporádicamente cuando se reciba alguna señal en medio del llamado a `AddWaitingTask`. Esto se debe a que el código de `nRead` es una sección crítica y por lo tanto se debe evitar que otras tareas invoquen este procedimiento concurrentemente.

nSystem implementa las secciones críticas de la forma más simple posible en un mono-procesador : inhibe las interrupciones. Por lo tanto la versión final de `nRead` es :

```
int nRead(int fd, ...)
{
  START_CRITICAL();
    ... como antes ...
  END_CRITICAL();
}
```

El procedimiento `START_CRITICAL` inhibe las señales de Unix usando la llamada al sistema `sigprocmask`. La misma llamada se usa en `END_CRITICAL` para reactivar las señales.

No es necesario inhibir las señales en `VtimerHandler` o `SigioHandler` porque estos procedimientos son para atender señales de Unix, y en este caso Unix los invoca desde ya con las señales inhibidas. De igual forma cuando estos procedimientos retornan, se reactivan automáticamente las señales.

4.4.6 Implementación de semáforos en nSystem

Un semáforo se representa mediante una estructura de datos que tiene la siguiente información :

int	count	El número de tickets
Queue	queue	Los proceso que esperan

El procedimiento nWaitSem debe bloquear la tarea que lo invoca cuando count es 0 :

```
void nWaitSem(nSem sem)
{
  START_CRITICAL();
  if (sem->count>0)
    sem->count--;
  else
  {
    current_task->status= WAIT_SEM;
    PutTask(sem->queue, current_task);
    /* Retoma otra tarea */
    ResumeNextReadyTask();
  }
  END_CRITICAL();
}
```

El procedimiento nSignalSem debe desbloquear una tarea cuando count es 0 y hay tareas en espera. La tarea desbloqueada toma de inmediato el control del procesador (ya que el scheduling es LCFS).

```
void nSignalSem(nSem sem)
{
  START_CRITICAL();
  if (EmptyQueue(sem->queue))
    sem->count++;
  else
  {
     nTask wait_task= GetTask(sem->queue);
     wait_task->status= READY;
     PushTask(ready_queue, current_task);
     PushTask(ready_queue, wait_task);
     /* wait_task es la primera en la cola! */
     ResumeNextReadyTask();
  }
  END_CRITICAL();
}
```

Es importante destacar que cuando en nSignalSem se inhiben las señales de Unix y se retoma otra tarea llamando a ResumeNextReadyTask, es esta otra tarea la que reactivará las señales en el END_CRITICAL del final de nWaitSem.

El END_CRITICAL que se encuentra al final de nSignalSem sólo se invocará cuando se retome nuevamente la tarea que invocó este procedimiento. Y entonces se reactivarán las señales que fueron inhibidas por otra tarea, aquella que perdió el procesador.

4.5 Implementación de procesos en Unix

Las implementaciones clásicas de Unix administran los procesos en un esquema similar al de nSystem. La estrategia de scheduling de procesos es un poco más elaborada pues implementa procesos con prioridades y al mismo tiempo intenta dar un buen tiempo de respuesta a los usuarios interactivos, algo que no es fácil de lograr debido a las restricciones que impone la memoria del procesador.

Más interesante que la estrategia de scheduling es estudiar, primero, a partir de qué momento los procesos comienzan a compartir parte de su espacio de direcciones, y segundo, cómo se logra la exclusión en secciones críticas.

4.5.1 Unix en monoprocesadores

Cuando un proceso ejecuta una aplicación el procesador se encuentra en modo usuario, y por lo tanto algunas instrucciones están inhibidas. Si se ejecutan se produce una interrupción que atrapa el núcleo.

Los segmentos del espacio de direcciones que son visibles desde una aplicación son el de código, datos y pila.

Cuando ocurre una interrupción el procesador pasa al modo sistema e invoca una rutina de atención de esa interrupción. Esta rutina es provista por el núcleo y su dirección se rescata del vector de interrupciones, el que no es visible desde un proceso en modo usuario.

Llamadas al sistema

Las llamadas al sistema siempre las ejecuta el núcleo en el modo sistema. La única forma de pasar desde una aplicación al modo sistema es por medio de una interrupción. Por lo tanto las llamadas al sistema se implementan precisamente a través de una interrupción.

Cuando se hace una llamada al sistema como read, write, etc., se invoca un procedimiento de biblioteca cuya única labor es colocar los argumentos en registros del procesador y luego ejecutar una instrucción de máquina ilegal (instrucción trap). Esta instrucción causa la interrupción que el núcleo reconoce como una llamada al sistema.

El segmento sistema

Al pasar a modo sistema, comienza a ejecutarse código del núcleo. El espacio de direcciones que se ve desde el núcleo contiene los mismos segmentos que se ven desde una aplicación : código, datos y pila. Pero

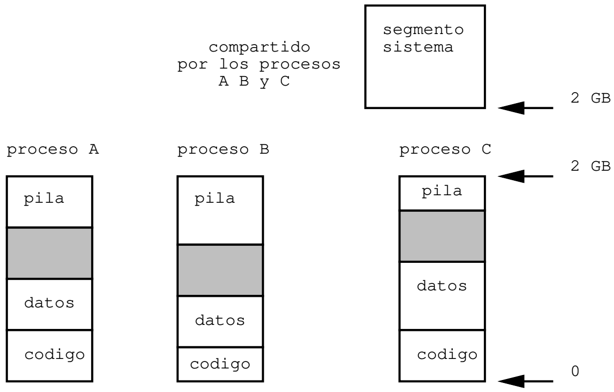

Figura 4.12: El espacio de direcciones en modo sistema.

además se agrega un nuevo segmento: el segmento sistema. Este segmento se aprecia en la figura 4.12.

El segmento sistema es *compartido* por todos los procesos. Es decir no existe un segmento sistema por proceso, sino que todos los procesos ven los mismos datos en este segmento. Si un proceso realiza una modificación en este segmento, los demás procesos verán la modificación.

El segmento sistema contiene el código del núcleo, el vector de interrupciones, los descriptores de proceso, la cola de procesos *ready*, las colas de E/S de cada dispositivo, los *buffers* del sistema de archivos, las puertas de acceso a los dispositivos, la memoria de video, etc. Enfín, todos los datos que los procesos necesitan compartir.

Por simplicidad, es usual que las implementaciones de Unix ubiquen los segmentos de la aplicación por debajo de los 2 gigabytes (la mitad del espacio direccionable) y el segmento sistema por arriba de los 2 gigabytes. Además en el segmento sistema se coloca toda la memoria real del computador. De esta forma el núcleo no solo puede ver la memoria del proceso interrumpido, sino que también pueden ver la memoria de los demás procesos.

Procesos ligeros en el núcleo

Cada proceso pesado tiene asociado en el núcleo un proceso ligero encargado de atender sus interrupciones (ver figura 4.13). Este proceso ligero tiene su pila en el segmento sistema y comparte sus datos y código con el resto de los procesos ligeros que corren en el núcleo.

Es en este proceso ligero en donde corre la rutina de atención de cualquier interrupción que se produzca mientras corre una aplicación en el modo usuario, sea esta interrupción una llamada al sistema, un evento de E/S relacionado con otro proceso, una división por cero, etc.

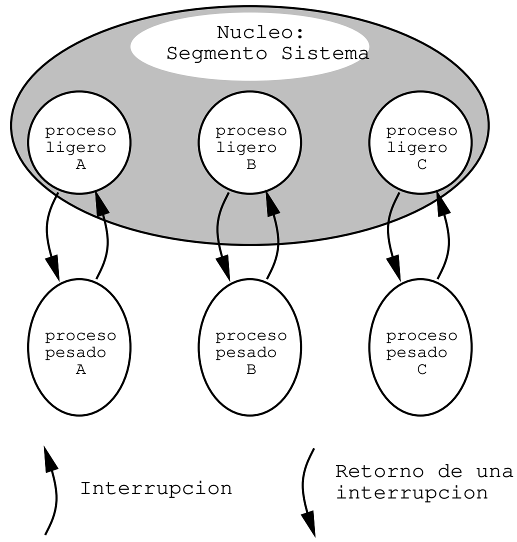

Figura 4.13: Procesos ligeros en el núcleo

Conceptualmente, una llamada al sistema es un mensaje síncrono que envía el proceso pesado al proceso ligero en el núcleo. De igual forma el retorno de la rutina de atención corresponde a la respuesta a ese mensaje que permite que el proceso pesado pueda continuar.

Los procesos del núcleo se ejecutan con las interrupciones inhibidas y por lo tanto no pueden haber cambios de contexto no deseados. De esta forma se logra la exclusión en las secciones críticas del núcleo. Los cambios de contexto sólo ocurren explícitamente.

Para implementar este esquema, el hardware de los microprocesadores modernos ofrece dos punteros a la pila, uno para el modo usuario y otro para el modo sistema y su intercambio se realiza automáticamente cuando ocurre una interrupción. Desde luego, en el modo usuario, el puntero a la pila del modo sistema no es visible, mientras que en el modo sistema sí se puede modificar el puntero a la pila del modo usuario.

Relación con nSystem

Cuando un proceso Unix pasa al modo sistema, le entrega el control a su respectivo proceso ligero en el núcleo. Este último corre con las interrupciones inhibidas, comparte el segmento sistema y tiene acceso a toda la máquina. Por lo tanto conceptualmente es similar a una nano tarea en nSystem que inhibe las interrupciones con `START_CRITICAL()`, comparte el espacio de direcciones con otras nano tareas y tiene acceso a todo el procesador virtual.

Es decir, *un proceso pesado en Unix que llama al sistema o es interrumpido se convierte en un proceso ligero ininterrumpible en el núcleo.* Luego, los algoritmos y estrategias que se vieron para nSystem pueden ser usados para implementar mensajes, semáforos, E/S, etc. en un monoprocesador bajo Unix.

El esquema presentado tiene la ventaja de ser simple de implementar. Sin embargo tiene la desventaja de que el sistema permanece largos períodos con las interrupciones inhibidas, lo que puede ser inaceptable para sistemas de control automático. Este tipo de sistemas requieren a veces respuestas rápidas a eventos, en el rango de unos pocos micro-segundos.

Ejemplos de sistemas que usan este esquema son Unix SCO, Berkeley Unix y Linux.

4.5.2 Unix en multiprocesadores

Un multiprocesador puede correr varios procesos Unix en paralelo. Todos los procesadores comparten la memoria real del computador, pero la visión conceptual de un proceso Unix sigue siendo la misma : *un solo procesador* en donde corre un programa. Es decir que los múltiples procesos Unix que corren en un multiprocesador no comparten sus espacios de direcciones.

Scheduling

En este esquema hay *un reloj regresivo por procesador*. De modo que el tiempo de cada procesador se puede multiplexar en tajadas de tiempo que se otorgan por turnos a los distintos procesos que estén listos para correr. Cada vez que ocurre una interrupción del reloj regresivo, la rutina de atención debe colocar el proceso en ejecución en la cola global de procesos *ready*, extraer un nuevo proceso y ejecutarlo.

El problema se presenta cuando la interrupción se produce simultáneamente en dos procesadores. Entonces los dos procesadores modificarán paralelamente la cola global de procesos, con el peligro de dejar la cola en un estado inconsistente.

Primera solución : un solo procesador en el núcleo

Cuando todos los procesadores ejecutan instrucciones en el modo usuario, no hay secciones críticas puesto que los procesos que corren en cada procesador no comparten sus espacios de direcciones. De igual forma si sólo uno de los procesadores se encuentra en el modo sistema, no hay peligro de modificación concurrente de una estructura de datos compartida.

El peligro de las secciones críticas se presenta cuando hay 2 o más procesadores reales en el modo sistema al mismo tiempo, es decir en el núcleo compartiendo el mismo segmento sistema.

Por lo tanto, la forma más simple de implementar Unix en un multiprocesador es impedir que dos procesadores puedan correr simultáneamente en el modo sistema.

El problema es cómo impedir que dos procesadores ejecuten en paralelo código del núcleo. La solución estándar en un monoprocesador que consiste en inhibir las interrupciones no funciona en un multiprocesador. En efecto, sólo se pueden inhibir las interrupciones externas a la CPU, es decir las del reloj regresivo y las de E/S. Un proceso en el modo usuario puede realizar llamadas al sistema con las interrupciones externas inhibidas.

Es decir que si dos procesos que se ejecutan en procesadores reales llaman al sistema al mismo tiempo, ambos entrarán al núcleo y habrá dos procesadores corriendo en el modo sistema. Esto originará problemas de sección crítica.

Exclusión a través de candados

Para impedir que dos procesadores pasen al modo sistema simultáneamente por medio de una llamada al sistema, se coloca un candado (*lock*) a la entrada del núcleo. Cuando un procesador entra al núcleo cierra el candado. Si otro procesador intenta entrar al núcleo tendrá que esperar hasta que el primer proceso abra el candado al salir del núcleo.

El candado es una estructura de datos que posee dos operaciones :

- `Lock(&lock)` : Si el candado está cerrado espera hasta que esté abierto. Cuando el candado esta abierto, cierra el candado y retorna de inmediato.
- `Unlock(&lock)`: Abre el candado y retorna de inmediato.

El candado se cierra al principio de la rutina de atención de las interrupciones por llamadas al sistema. Es decir cuando el procesador ya se encuentra en el modo sistema, pero todavía no ha realizado ninguna acción peligrosa. Si el procesador encuentra cerrado el candado, se queda bloqueado hasta que el candado sea abierto por el procesador que se encuentra en el modo sistema. Mientras tanto el procesador bloqueado no puede ejecutar otros procesos, pues dado que la cola *ready* es una estructura de datos compartida, no puede tener acceso a ella para extraer y retomar un proceso.

El candado se abre en la rutina de atención justo antes de retornar a la aplicación.

Observe que no es razonable implementar el candado antes de que la aplicación pase al modo sistema, puesto que nadie puede obligar a una aplicación a verificar que el candado está abierto antes de hacer una llamada al sistema.

Implementación de candados

La siguiente implementación es errónea. El candado se representa mediante un entero que puede estar `OPEN` o `CLOSED`.

```
void Lock(int *plock)
{
  while (*plock==CLOSED)
    ;
  *plock=CLOSED;
}

void Unlock(int *plock)
{
  *plock= OPEN;
}
```

El problema es que dos procesadores pueden cargar simultáneamente el valor de `*plock` y determinar que el candado esta abierto y pasar ambos al modo sistema.

Para implementar un candado, la mayoría de los microprocesadores ofrecen instrucciones de máquina atómicas (indivisibles) que permiten implementar un candado. En sparc esta instrucción es :

`swap` *dir. en memoria*, *reg*

Esta instrucción intercambia atómicamente el contenido de la dirección especificada con el contenido del registro. Esta instrucción es equivalente al procedimiento :

```
int Swap(int *ptr, int value)
{
  int ret= *ptr;
  *ptr= value;
  return ret;
}
```

Sólo que por el hecho de ejecutarse atómicamente, dos procesadores que tratan de ejecutar en paralelo esta instrucción, serán *secuencializados*.

Veamos como se puede usar este procedimiento para implementar un candado :

```
void Lock(int *plock)
{
  while (Swap(plock, CLOSED)==CLOSED)
    mini-loop(); /* busy-waiting */
}

void Unlock(int *plock)
{
  *plock= OPEN;
}
```

Observe que si un procesador encuentra el candado cerrado, la operación `swap` se realiza de todas formas, pero el resultado es que el candado permanece cerrado. En inglés, un candado implementado de esta forma se llama *spin-lock*.

Si el candado se encuentra cerrado, antes de volver a consultar por el candado se ejecuta un pequeño ciclo que dure algunos micro-segundos. Esto se hace con el objeto de disminuir el tráfico en el bus de datos que es compartido por todos los procesadores para llegar a la memoria. Para poder entender esto es necesario conocer la implementación del sistema de memoria de un multiprocesador que incluye un caché por procesador y un protocolo de coherencia de cachés.

Ventajas y desventajas

Se ha presentado una solución simple para implementar Unix en un multiprocesador. La solución consiste en impedir que dos procesadores puedan ejecutar simúltanemente código del núcleo, dado que comparten el mismo segmento sistema.

Este mecanismo se usa en SunOS 4.X ya que es muy simple de implementar sin realizar modificaciones mayores a un núcleo de Unix diseñado para monoprocesadores, como es el caso de SunOS 4.X. Este esquema funciona muy bien cuando la carga del multiprocesador está constituida por procesos que requieren mínima atención del sistema.

La desventaja es que en un multiprocesador que se utiliza como servidor de disco en una red, la carga está constituida sobretodo por muchos procesos "demonios" que atienden los pedidos provenientes de la red (servidores NFS o *network file system*). Estos procesos demonios corren en SunOS sólo en modo sistema. Por lo tanto sólo uno de estos procesos puede estar ejecutándose en un procesador en un instante dado. El resto de los procesadores está eventualmente bloqueado a la espera del candado para poder entrar al modo sistema.

Si un procesador logra extraer un proceso para hacerlo correr, basta que este proceso haga una llamada al sistema para que el procesador vuelva a bloquearse en el candado por culpa de que otro procesador esta corriendo un proceso NFS en el núcleo.

Núcleos Multi-threaded

Para resolver definitivamente el problema de las secciones críticas en el núcleo de un sistema operativo para multiprocesadores se debe permitir que varios procesadores puedan ejecutar procesos ligeros en el núcleo. Además estos procesos deben ser interrumpibles. Los procesos ligeros del núcleo deben sincronizarse a través de semáforos, mensajes o monitores.

Un semáforo del núcleo se implementa usando un *spin-lock*. Éste es un candado a la estructura de datos que representa el semáforo. Las operaciones sobre semáforos las llamaremos `kWait` y `kSignal` para indicar que se llaman en el *kernel* (núcleo).

```
void kWait(kSem sem)
{
  Lock(&sem->lock);
  if (sem->count>0)
  {
    sem->count--;
    Unlock(&sem->lock);
  }
  else
  {
    currentTask()->status= WAIT_SEM;
    PutTask(sem->queue, currentTask());
    Unlock(&sem->lock);
    /* Retoma otra tarea */
    ResumeNextReadyTask();
  }
}
```

El principio es que el candado permanece cerrado sólo cuando un procesador está manipulando el semáforo. Si el contador está en 0, el proceso se coloca en la cola del semáforo, se abre nuevamente el candado y el procesador llama al scheduler para retomar un nuevo proceso. En ningún caso, el procesador espera en un *spin-lock* hasta que el contador del semáforo se haga positivo. De esta forma, el tiempo que permanece cerrado el candado es del orden de un microsegundo, muy inferior al tiempo que puede permanecer cerrado el candado que permite el acceso al núcleo en SunOS 4.X.

Observe que cuando hay varios procesadores corriendo, no se puede tener una variable global que apunta al proceso en ejecución. Además, el scheduler debe tener otro candado asociado a la cola de procesos ready, para evitar que dos procesadores extraigan o encolen al mismo tiempo.

El código para kSignal es:

```
void kSignal(Sem sem)
{
  Lock(&sem->lock);
  if (EmptyQueue(sem->queue))
  {
    sem->count++;
    Unlock(&sem->lock);
  }
  else
  {
     nTask wait_task= GetTask(sem->queue);
     Unlock(&sem->lock);
     wait_task->status= READY;
     /* ready_lock es el candado para
        la ready_queue */
     Lock(&ready_lock);
     PushTask(ready_queue, current_task);
     PushTask(ready_queue, wait_task);
     Unlock(&ready_lock);
     /* wait_task no necesariamente es la
        primera en la cola! Por qu\'e? */
     ResumeNextReadyTask();
  }
}
```

Ejemplos de sistemas operativos que usan este esquema son Solaris 2, Unix System V.4, Window NT y Mach (Next y OSF/1). Se dice que el núcleo de estos sistemas es *multi-threaded*, porque pueden correr en paralelo varios threads en el núcleo, a diferencia de la implementación clásica en donde corre un solo thread en el núcleo, y además es ininterrumpible.

Capítulo 5

Administración de Memoria

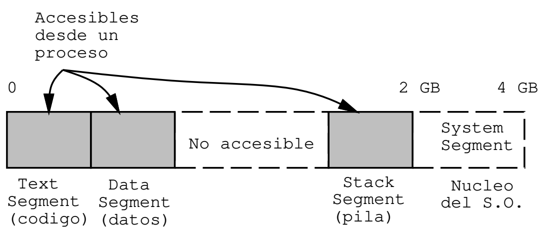

Figura 5.1: Segmentos del espacio de direcciones virtuales de un proceso.

5.1 Segmentación

La *segmentación* es un esquema para implementar espacios de direcciones virtuales que se usaba en los primeros computadores de tiempo compartido. Pese a que hoy en día se encuentra en desuso, es interesante estudiar la segmentación por su simplicidad.

Como se vió en capítulos anteriores, cada proceso tiene su propio espacio de direcciones virtuales, independiente del resto de los procesos. Este espacio de direcciones virtuales se descompone en cuatro áreas llamadas segmentos (ver figura 5.1) :

- Un segmento de código.
- Un segmento de datos.
- Un segmento de pila.
- Un segmento de sistema, invisible para la aplicación, en donde reside el núcleo.

En una organización *segmentada*, los segmentos *residen en un área contigua de la memoria real del computador*. La figura 5.2 muestra un posible estado de la memoria de un computador con los distintos segmentos de varios procesos.

En la figura se observa que el segmento sistema contiene una imagen de toda la memoria real del computador. Esta es una técnica muy usada en la implementación de Unix.

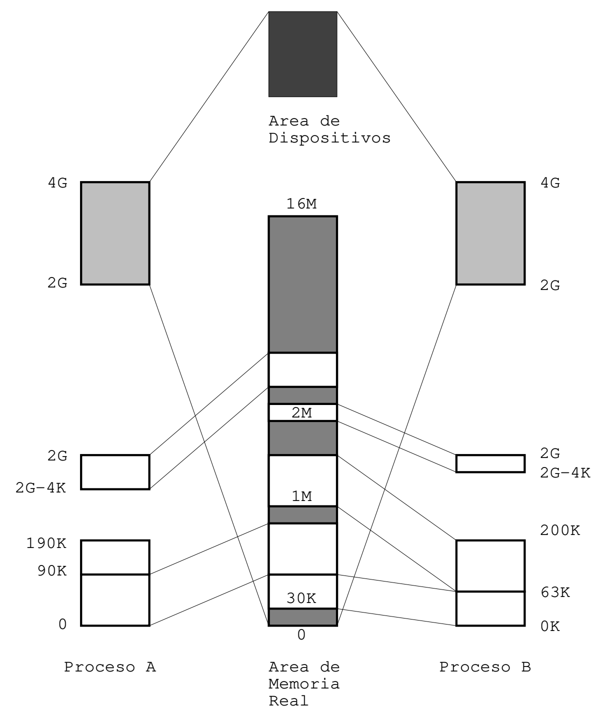

Figura 5.2: Ejemplo de ubicación de los segmentos de dos procesos en la memoria real.

Cuando un proceso accesa la memoria siempre suministra una dirección en su espacio de direcciones virtuales. El procesador debe traducir esa dirección a su posición efectiva en la memoria real del computador, es decir a su dirección real.

5.1.1 La tabla de segmentos del procesador

Para traducir las direcciones virtuales a direcciones reales, el procesador posee una tabla de segmentos con 4 filas. Cada una de estas filas describe uno de los 4 segmentos del programa en ejecución. Para cada segmento se indica :

- Base : Dirección virtual en donde comienza (incluyendo esta dirección).
- Límite : Dirección virtual en donde finaliza (excluyendo esta dirección).
- Desplazamiento : Desplazamiento que hay que sumar a una dirección virtual para obtener su dirección real. Se calcula como la dirección de comienzo del segmento en la memoria real menos la dirección virtual de inicio del segmento.
- Atributos del segmento : lectura/escritura, solo lectura e invisible (accesible solo por el núcleo).

La siguiente tabla muestra el contenido de la tabla de segmentos para el proceso B de la figura 5.1.

Inicio virtual	Fin virtual	Desplazamiento	Atributos	Corresponde a
2G	4G	0-2G	Invisible	sistema
2G-4K	2G	2M-2G+4K	Lect/Escr.	pila
63K	200K	1M-63K	Lect/Escr.	datos
0	63K	30K-0	Lectura	código

5.1.2 Traducción de direcciones virtuales

En cada acceso a la memoria un proceso especifica una dirección virtual. El hardware del procesador traduce esta dirección virtual a una dirección real utilizando un conjunto de registros del procesador que almacenan la tabla de segmentos del proceso en ejecución.

Esta traducción se lleva a cabo de la siguiente forma : el hardware del procesador compara la dirección virtual especificada con la base y el límite de cada uno de los segmentos :

$$base_{seg} \leq \textit{dirección virtual} < \textit{límite}_{seg}$$

Si la dirección (virtual) cae dentro de uno de los segmentos entonces la dirección real se obtiene como :

$$\textit{dirección real} = \textit{dirección virtual} + desp_{seg}$$

Si la dirección no cae dentro de ninguno de los segmentos entonces se produce una interrupción. En Unix usualmente el proceso se aborta con un mensaje no muy explicativo que dice *segmentation fault*. El Hardware es capaz de realizar eficientemente esta traducción en a lo más un ciclo del reloj del microprocesador.

El núcleo del sistema operativo se encarga de colocar valores apropiados en la tabla de segmentos, y lo hace de tal forma que una dirección virtual jamás pertenece a dos segmentos. Sin embargo, es importante hacer notar que una misma dirección real sí puede pertenecer a dos segmentos asociados a procesos distintos.

Cambios de contexto

Cada proceso tiene su propia tabla de segmentos almacenada en su descritor de proceso. El procesador mantiene solamente en registros internos la tabla de segmentos del proceso que está corriendo.

Cuando el núcleo decide efectuar un cambio de contexto, tiene que cambiar el espacio de direcciones virtuales del proceso saliente por el del proceso entrante. Para realizar este cambio, el núcleo carga la tabla de segmentos del procesador con la tabla de segmentos que se encuentra en el descriptor del proceso entrante.

5.1.3 Administración de la memoria de segmentos

El núcleo crea y destruye segmentos cuando se crea o termina un proceso o cuando se carga un nuevo binario (llamadas al sistema `fork`, `exit` y `exec` en Unix). El núcleo administra la memoria para estos segmentos en un área de la memoria denominada *heap* (montón). Esta estructura es similar al *heap* que utiliza `malloc` y `free` para administrar la memoria de un proceso.

Por conveniencia, el núcleo separa en dos heaps independientes la administración de la memoria para segmentos y la administración de la memoria para sus propias estructuras de datos como los descriptores de proceso, colas de scheduling, colas de E/S, etc.

En un heap hay trozos de memoria ocupada y trozos libres. Los trozos libres se enlazan en una lista que es recorrida, de acuerdo a alguna estrategia, cuando se solicita un nuevo trozo de memoria. Cuando se libera un trozo, se agrega a la lista. Esto hace que el heap se *fragmente* progresivamente, es decir se convierte en un sin número de trozos libres de pequeño tamaño.

Para disminuir la fragmentación, al liberar un trozo se revisan los trozos adyacentes en el heap para ver si están libres. Si alguno de ellos está libre (o los dos), entonces los trozos se concatenan.

Las estrategias más conocidas para realizar la búsqueda en la lista de trozos libres son :

- *First-fit*: Se recorre secuencialmente la lista de trozos libres hasta encontrar el primer trozo que sea de tamaño mayor o igual al tamaño del trozo solicitado. El trozo encontrado se parte en dos para entregar sólo el tamaño solicitado.

- *Best-fit*: Se recorre toda lista para encontrar el trozo que se acerque más en tamaño al trozo solicitado. El mejor trozo se parte en dos como en First-fit.

- *Worst-fit*: Se recorre toda la lista para encontrar el trozo más grande. Este trozo se parte en dos como First-fit. Solo existe por completitud.

La estrategia best-fit tiene mal desempeño debido a que al partir el mejor trozo, el trozo que queda libre es demasiado pequeño. Será muy difícil que se pueda entregar ese pequeño trozo en un pedido subsecuente. Esto hace que la lista de trozos libres se va poblando de trozos demasiado pequeños que aumentan el costo de recorrer la lista.

De estas estrategias, first-fit es la que mejor funciona, pero organizando los trozos libres en una lista circular. La idea es evitar que se busque siempre a partir del mismo trozo, puesto que esto tiende a poblar el comienzo de la lista con trozos pequeños, alargando el tiempo de búsqueda. En esta variante, al hacer una búsqueda se memoriza el último trozo visitado en la lista circular. Al hacer una nueva búsqueda se comienza por el trozo siguiente al trozo memorizado.

5.1.4 Compactación en el núcleo

A medida que se solicitan y liberan trozos de memoria en un heap, inevitablemente el heap se va fragmentando. La mejores estrategias como first-fit disminuyen la rapidez de fragmentación, pero no la evitan.

En un heap como el que utiliza `malloc` y `free` para administrar la memoria de un proceso, el problema de la fragmentación no tiene solución. Pero no se trata de un problema grave, puesto que un proceso

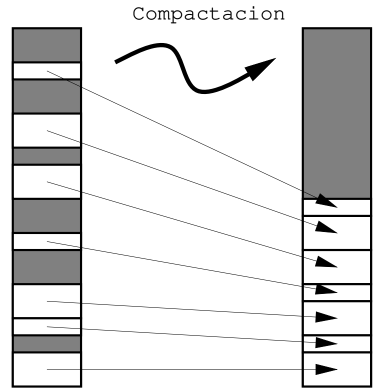

Figura 5.3: Compactación de la memoria real.

Unix puede hacer crecer su espacio de datos, cuando ninguno de los trozos disponibles es del tamaño solicitado.

En el núcleo sin embargo, cuando se trata de administrar la memoria de segmentos, no existe la posibilidad de hacer crecer la memoria real del computador. En cambio, el núcleo sí puede *compactar* el heap dedicado a los segmentos (ver figura 5.3). Para ello basta desplazar los segmentos hasta que ocupen un área de memoria contigua en el heap. Así la lista de trozos libres se reduce a uno solo que contiene toda la memoria disponible en el heap.

Al desplazar los segmentos de un proceso, las direcciones reales cambian, por lo que hay que modificar sus desplazamientos en la tabla de segmentos del proceso, pero *las direcciones virtuales manejadas por el proceso no cambian*. Esto significa que si un objeto ocupaba una dirección virtual cualquiera, después de desplazar los segmentos del proceso, su dirección virtual sigue siendo la misma, a pesar de que su dirección real cambió. Esto garantiza que los punteros que maneja el proceso continuarán apuntando a los mismos objetos.

Compactación en un proceso

Observe que la solución anterior no funciona para `malloc` y `free`, puesto que si se compacta el heap de un proceso Unix, sí se alteran las direcciones virtuales de los objetos. Después de compactar habría que entrar a corregir los punteros, lo que es muy difícil de hacer en C, puesto que ni siquiera se puede distinguir en la memoria un entero de un puntero.

Por esta misma razón es conveniente separar en el núcleo la administración de los segmentos del resto de las estructuras que pueda requerir el núcleo. Si se desplazan segmentos, es simple determinar cuales son los desplazamientos que hay que modificar en los descriptores de proceso. En cambio si se desplazan estructuras como colas o descritores de proceso, es muy complicado determinar desde que punteros son apuntadas estas estructuras.

Existen lenguajes de programación, como Lisp, Java y Smalltalk, que sí permiten implementar compactación. Estos lenguajes se implementan haciendo que cada objeto en el heap tenga una etiqueta que dice qué tipo de objeto es, por lo que un compactador puede determinar si el objeto tiene punteros que deberían corregirse. En todo caso, esta compactación es responsabilidad del proceso y no del núcleo, por la misma razón que en C `malloc` y `free` son parte del proceso.

5.1.5 El potencial de la segmentación

La segmentación es un esquema simple para implementar espacios de direcciones virtuales, necesarios para poder garantizar protección entre procesos. A pesar de ser simple, permite implementar una serie de optimizaciones que permiten administrar mejor la memoria del computador que hasta el día de hoy es un recurso caro y por lo tanto escaso. A continuación se describen estas optimizaciones.

Extensión automática de la pila en caso de desborde

Cuando se va a crear un proceso, es imposible determinar a priori cuanta memoria se requiere para la pila. Si se otorga un segmento de pila muy pequeño, puede ocurrir un desborde de la pila. Por otro lado, si se otorga memoria en exceso se malgasta un recurso que podría dedicarse a otros segmentos.

En un sistema segmentado se puede se puede implementar un mecanismo de *extensión de la pila en demanda*. Este mecanismo consiste en asignar un segmento de pila pequeño y hacerlo crecer sólo cuando la pila se desborda. En efecto, el desborde de la pila ocurre cuando el puntero a la pila se decrementa más alla de la base del segmento de la pila. Cuando se intente realizar un acceso por debajo de este segmento se producirá una interrupción.

Esta interrupción hace que se ejecute una rutina de atención del núcleo que diagnostica el desborde de la pila. Entonces el núcleo puede solicitar un segmento más grande, copiar la antigua pila al nuevo segmento, liberar el antiguo segmento, modificar el desplazamiento del segmento pila en la tabla de segmento y retomar el proceso interrumpido en forma transparente, es decir sin que este se de cuenta de que desbordó su pila.

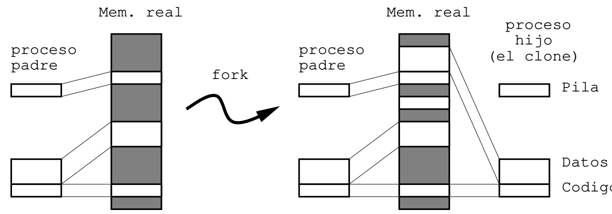

Figura 5.4: Implementación de `fork`.

Con este mecanismo la memoria que necesita un proceso para su pila se asigna en demanda: la memoria se asigna a medida que el proceso la necesita.

Extensión explícita de los datos

La misma estrategia se puede usar para el segmento de datos. Sin embargo, en el caso de los datos, no es fácil distinguir un desborde del área de datos de la utilización de un puntero mal inicializado y que contiene basura.

Por esta razón en Unix "se prefiere" que un proceso solicite explícitamente la extensión de su segmento de datos usando la primitiva `sbrk(size)`, en donde `size` es la cantidad de bytes a agregar a su segmento de datos. Observe sin embargo que el núcleo no está obligado a entregar exactamente esa cantidad. Por restricciones de implementación, el segmento podría crecer en más de lo que se solicita.

El mecanismo de implementación de esta extensión es esencialmente análogo al mecanismo de extensión de la pila.

Implementación de fork

En Unix la llamada al sistema `fork` crea un *clone* de un proceso. El clone es independiente del primero, pero se crea con una copia del código, de los datos y de la pila de proceso que hace `fork`. Ambos procesos comparten los mismos descriptores de archivo abiertos al momento de invocar `fork`. Sólo se diferencian por que el `fork` del proceso original retorna el identificador del proceso clone creado, mientras que el `fork` del clone retorna 0.

Esta llamada al sistema se implementa duplicando los segmentos de datos y pila (ver figura 5.4). El segmento de código puede compartirse entre ambos procesos, pues normalmente un proceso no debería modificar su código. Por lo tanto el segmento de código se marca con el atributo de solo lectura. Si un proceso intenta modificar sus datos, se producirá una interrupción que abortará el proceso que falla.

Lamentablemente esta implementación es muy ineficiente para el uso típico de `fork`. Esta llamada se usa principalmente en el intérprete de comandos cuando el usuario ingresa un nuevo comando. El intérprete hace un `fork` para crear un clone de sí mismo, desde donde de inmediato llama a `exec` con el nombre del comando. El `fork` necesita duplicar los segmentos de datos y pila para crear el clone, pero estos segmentos serán inmediatamente destruidos con la llamada a `exec` que creará otros 3 segmentos.

Es importante hacer notar que los primeros microcomputadores basados en la Z80 y luego la 68000 no ofrecían hardware para segmentación, por lo que en ellos era imposible implementar `fork`, y por lo tanto Unix. La 8088 ofrece un esquema de segmentos de tamaño fijo en 64 KB y sin protección, por lo que era posible implementar Unix en el PC original, siempre y cuando los usuarios se conformaran con procesos de no más de 3*64 KB (después de todo, los primeros Unix corrían en 64 KB) y aceptaran que procesos maliciosos pudiesen botar el sistema completo.

Swapping

Si al crear un nuevo segmento, el núcleo no encuentra un trozo del tamaño solicitado, el núcleo compacta el heap de segmentos. Si aún así no hay memoria suficiente, el núcleo puede llevar procesos completos al disco. Esto se denomina *swapping* de procesos.

Para llevar un proceso a disco, el núcleo escribe una copia al bit de cada segmento en una área denominada área de swap. Luego, la memoria ocupada por esos segmentos se libera para que pueda se ocupada para crear otros segmentos. El descriptor de proceso continúa en memoria real, pero se reemplaza el campo desplazamiento en la tabla de segmentos del proceso por una dirección que indica la ubicación de cada segmento en el área de swap. Además el proceso pasa a un estado especial que indica que se ubica en disco y por lo tanto no puede ejecutarse.

El scheduler de mediano plazo decide qué proceso se va a disco y qué proceso puede regresar a la memoria del computador, de acuerdo a alguna estrategia. Esta estrategia tiene que considerar que llevar un proceso a disco y traerlo cuesta al menos un segundo. Por esto no conviene llevar procesos a disco por poco tiempo. Se llevan por al menos 10 segundos.

En un sistema computacional hay muchos procesos *demonios* que pasan inactivos por largos períodos, como por ejemplo el *spooler* para la impresión si nadie imprime. Este tipo de procesos son los primeros candidatos para irse a disco.

Cuando todos los procesos en memoria están activos (sus períodos de espera son muy cortos), la medida de llevar procesos a disco es de emergencia. Si un proceso interactivo (un editor por ejemplo) se lleva a disco por 10 segundos, la pausa introducida será tan molesta para el usuario afectado que incluso llegará a temer una caída del sistema. Pero de todas formas, como alternativa es mejor que botar el sistema o destruir procesos al azar.

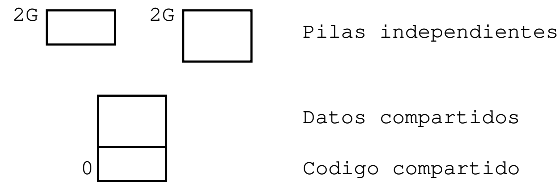

Figura 5.5: Procesos semi-ligeros.

Implementación de procesos semi-ligeros

Los procesos ligeros comparten un mismo espacio de direcciones. El problema es que su pila se ubica en el heap del proceso y por ende es de tamaño fijo e inextensible. Si un proceso ligero desborda su pila, se caen todos los procesos que comparten ese mismo espacio de direcciones.

En un esquema segmentado el núcleo puede ofrecer procesos semi-ligeros en donde se comparte el área de código y de datos (como los procesos ligeros) pero no se comparte la porción del espacio de direcciones que corresponde a la pila (ver figura 5.5).

Cada proceso semi-ligero tiene un segmento de pila asignado en el mismo rango de direcciones. Si un proceso semi-ligero desborda su pila, el núcleo puede extender automáticamente su pila, como si fuesen procesos pesados.

Los procesos semi-ligeros tienen dos inconvenientes con respecto a los procesos ligeros:

- Su implementación debe ser responsabilidad del núcleo, porque requiere manejar la tabla de segmentos. En cambio los procesos ligeros pueden implementarse a nivel de un proceso Unix, sin intervención del núcleo.

- El cambio de contexto es tán caro como el de los procesos pesados.

A pesar de que el cambio de contexto es caro, todavía se puede decir que son procesos ligeros debido a que se pueden comunicar eficientemente a través del envío de punteros al área de datos. Pero observe que es erróneo enviar punteros a variables locales a un proceso semi-ligero.

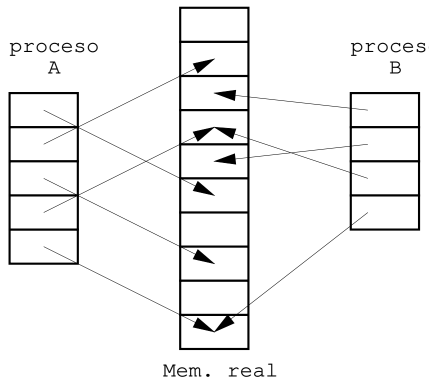

Figura 5.6: Particionamiento de los espacios de direcciones en páginas.

5.1.6 Problemas de la segmentación

Apesar de que un sistema segmentado aporta soluciones y optimizaciones, persisten algunos problemas que solo se resuelven con paginamiento. Estos problemas son :

- La implementación de `fork` es ineficiente, pues copia completamente los segmentos. Con paginamiento, frecuentemente no es necesario copiar completamente datos y pila.
- La compactación introduce una pausa generalizada del sistema que es difícil distinguir de una caída del sistema. Con paginamiento no es necesario efectuar compactación.
- El tamaño de un proceso no puede exceder el tamaño de la memoria real, puesto que un proceso necesita estar completamente residente para poder correr. Con *paginamiento en demanda*, el que veremos más adelante, un proceso no necesita estar completamente residente en memoria para poder correr.

5.2 Paginamiento

Prácticamente el único mecanismo que se usa hoy en día para implementar espacios de direcciones virtuales es paginamiento. En este mecanismo una página es un bloque de memoria de tamaño fijo y potencia de 2 (típicamente 4 KB u 8 KB). La idea es que el espacio de direcciones virtuales y el espacio de direcciones reales se particionan en páginas del mismo tamaño (ver figura 5.6).

Una página siempre se procesa como un todo, si se otorga a un proceso, se otorga la página completa, en ningún caso una fracción de la página. Cada página

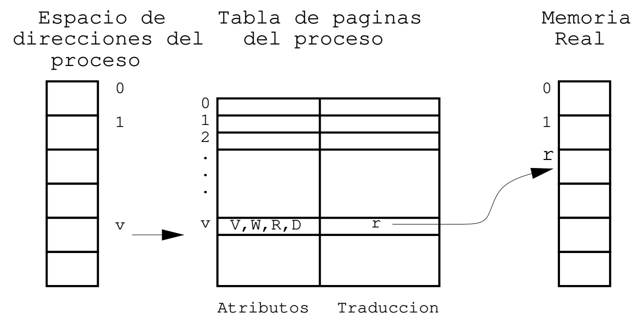

Figura 5.7: La tabla de páginas de un proceso.

del espacio de direcciones virtuales puede residir en cualquiera de las páginas del espacio de direcciones reales, siempre y cuando allí haya memoria física.

5.2.1 La tabla de páginas

En el descriptor de cada proceso se guarda una tabla de páginas que consiste en una arreglo que indica en qué página de la memoria real se ubica cada página del espacio de direcciones virtuales (ver figura 5.7). Para ello se enumeran las páginas virtuales y las páginas reales de 0 en adelante. De esta forma en la fila v de la tabla de páginas se indica en que número de página real se encuentra la página virtual número v.

Atributos de una página

Los atributos de una página son un conjunto de bits que indican qué operaciones es válido realizar en cada página del proceso. Estos bits son :

- Bit V : En 1 indica que es válido leer esta página. En 0, la página no se puede leer ni escribir, simplemente esa página no reside en la memoria real. Si se accesa esa página se produce una *page fault* (falta de página) que invoca una interrupción.
- Bit W : En 1 indica que la página se puede escribir, pero sólo si además V está en 1. Si está en 0 y se escribe en la página, se produce una interrupción.
- Bit R : El hardware coloca este bit en 1 cada vez que se lee una palabra de esta página. Se usa para implementar paginamiento en demanda.
- Bit D : El hardware coloca este bit en 1 cada vez que se escribe en esta página.

Cambios de contexto

El hardware del procesador posee usualmente un registro que indica la dirección en la memoria real de la tabla de páginas del proceso en ejecución. Este registro necesariamente contiene una dirección real pues de ser una dirección virtual, caeríamos en una ciclo infinito al tratar de determinar en qué lugar de la memoria real se encuentra.

Durante un cambio de contexto es necesario cambiar el espacio de direcciones virtuales por el del proceso entrante, por lo que este registro se modifica con la dirección de la tabla de páginas que se encuentra en el descriptor del proceso entrante.

Tamaño de la tabla de páginas

Si las direcciones son de 32 bits y las páginas son de 2^k bits, significa que la tabla de páginas debe poseer 2^{32-k} filas. Esto da 512 K-filas, si las páginas son de 8 KB. Por lo tanto se requerirían 2 MB sólo para almacenar la tabla si se destinan 4 bytes por cada fila, lo que es obviamente excesivo.

Este problema lo resolveremos temporalmente restringiendo el espacio de direcciones virtuales a 1024 páginas u otra cantidad razonable. Por ende el tamaño de la tabla de páginas necesita sólo 1024 filas o 4 KB. Los accesos a la página 1024 o una página superior causan una interrupción.

5.2.2 Traducción de direcciones virtuales

El hardware del microprocesador se encarga de traducir las direcciones virtuales que accesa un proceso a direcciones reales. Para efectuar esta traducción el hardware determina el número de la página virtual a partir de la dirección virtual accesada por el proceso y la traduce al número de la página real en donde reside esa página virtual.

Si las páginas son de 2^k bytes y las direcciones de 32 bits, es muy fácil determinar el número de una página en una dirección. Basta observar que todas las direcciones que caen en la misma página v tienen un mismo prefijo de $32-k$ bits que corresponde precisamente a v, los últimos k bits en la dirección son el desplazamiento de la palabra accesada dentro de la página.

El mecanismo de traducción de direcciones se observa en la figura 5.8. De la dirección virtual se extraen los bits que corresponden al número de página virtual (v). Con ellos se subindica la tabla de páginas y se determina en qué página real (r) se encuentra la

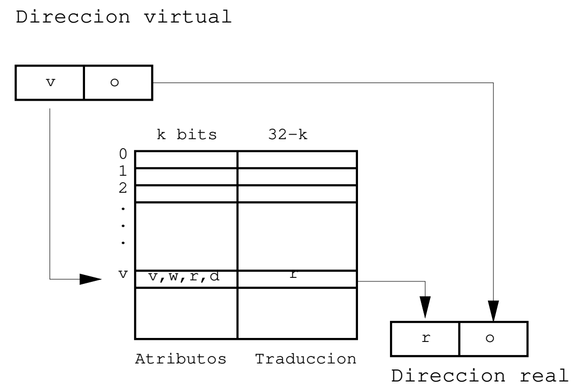

Figura 5.8: Traducción de direcciones virtuales a direcciones reales por medio de la tabla de páginas.

palabra accesada, además se verifica que los atributos presentes en esa posición en la tabla autorizan la operación solicitada (lectura o escritura). Finalmente, la dirección real se forma a partir del número de la página real y el desplazamiento o dentro de la página que se extrae de la dirección virtual.

5.2.3 El potencial del paginamiento

El paginamiento permite implementar espacios de direcciones virtuales y por lo tanto protección entre procesos. También permite realizar todas las optimizaciones del uso de la memoria que ofrece la segmentación:

- Extensión automática de la pila en caso de desborde.

  Cuando un proceso desborda su pila, el núcleo puede asignarle una nueva página en forma transparente para el proceso.

- Extensión explícita de los datos.

  Cuando el proceso solicita la extensión de los datos invocando la llamada al sistema `sbrk`, el núcleo asigna las páginas necesarias según la memoria pedida.

- Swapping.

  Cuando la memoria escasea, el núcleo puede llevar procesos a disco: sus datos, su código, su pila y sus tablas de páginas.

- Implementación de procesos semi-ligeros.

  Los procesos semi-ligeros comparten la porción del espacio de direcciones que corresponde al código y los datos, pero poseen una porción no compartida en donde se ubica la pila. Con esto es posible implementar extensión automática de la pila en caso de desborde.

Pero además, el mecanismo de paginamiento permite realizar mejor algunas optimizaciones que en segmentación.

No hay framentación externa

Se dice que la fragmentación asociada a la segmentación es una fragmentación externa porque se pierden trozos de memoria fuera de la memoria asignada a los procesos. Este problema no existe en paginamiento: se puede otorgar a un proceso un conjunto de páginas reales que ocuparán una porción contigua del espacio de direcciones virtuales, aún cuando estas páginas reales no forman un bloque contiguo en la memoria real del computador.

La administración de páginas de tamaño fijo es un problema simple de resolver eficientemente. Las páginas disponibles se organizan en una lista enlazada simple. La extracción y la devolución de páginas toma tiempo constante y requiere unas pocas instrucciones. Cuando un proceso solicita explícitamente o implícitamente más memoria, se le otorga tantas páginas como sea necesario para completar la memoria requerida, sin importar que las p'aginas disponibles estén diseminadas por toda la memoria.

Aunque no hay fragmentación externa sí puede haber *fragmentación interna*. Este último tipo de fragmentación ocurre cuando a un proceso se le otorga más memoria de la que él solicita. En efecto, al otorgar un grupo de páginas, es usual que la última página contenga un sobrante que el proceso no ha solicitado. Este sobrante existe porque en paginamiento no se pueden otorgar fracciones de página. En todo caso, los estudios indican que la fragmentación interna significa una pérdida de memoria inferior a la fragmentación externa.

Implementación eficiente de fork

Es el intérprete de comandos el que típicamente llama a `fork` para crear una copia de sí mismo. Esta copia invoca de inmediato `exec` del binario de un comando ingresado por el usuario. En un esquema segmentado esto requiere duplicar los segmentos de datos y de pila del intérprete de comandos, para que luego se destruyan de inmediato. Esto es un uso muy ineficiente del tiempo de procesador.

En un esquema de paginamiento, `fork` se puede implementar eficientemente haciendo uso de la técnica *copy-on-write* (copia al escribir). La idea es no duplicar de inmediato las páginas de datos y de pila, sino que sólo duplicar la tabla de página del proceso que invoca `fork` (ver figure 5.9). Inicialmente los procesos padre e hijo comparten todas sus páginas, pero se protegen contra escritura.

Al retomar estos procesos cualquier escritura del padre o del hijo genera una interrupción. El núcleo duplica entonces la página que se modifica y se le coloca permiso de escritura. De esta forma se retoma el proceso interrumpido en forma transparente, sin que se dé cuenta del cambio de página.

La ganancia se obtiene cuando el `fork` va seguido prontamente por un `exec`, puesto que será necesario duplicar sólo unas 2 o 3 páginas, lo que resulta ser mucho más económico que duplicar todas las páginas.

Implementación de procesos semi-pesados

En una esquema de paginamiento se puede implementar todo el rango de "pesos" de los procesos (ver figura 5.10). En esta categorización los procesos se distinguen por la amplitud del espacio de direcciones que comparten. Mientras más espacio comparten, más ligeros son.

Los procesos semi-pesados se ubican a mitad de camino entre los semi-ligeros y los pesados. Los procesos semi-pesados tienen dos porciones del espacio de direcciones para datos:

- Una primera porción no compartida correspondiente a las variables globales y al *heap* administrado por `malloc/free`.
- Una segunda porción que sí es compartida y en donde se coloca un *heap* que se administra con procedimientos especiales: `shared_malloc` y `shared_free`. Estos procedimientos deben adquirir un `lock` antes de realizar cualquier cambio en el heap, para evitar problemas de sección crítica.

Los procesos semi-pesados mejoran los procesos semi-ligeros porque no es necesario reescribir los procedimientos de biblioteca que invocan `malloc` y `free`. Esto porque los procesos semi-pesados no necesitan sincronizarse para llamar a `malloc` o `free`.

Observe que un proceso semi-pesado puede enviar a otro proceso semi-pesado un puntero a una estructura que pidió con `shared_malloc`, pero es un error mandar un puntero a una estructura que se pidió con `malloc`.

Paginamiento en demanda

Por último, el paginamiento permite realizar una optimización sobre el uso de la memoria que no se pue-

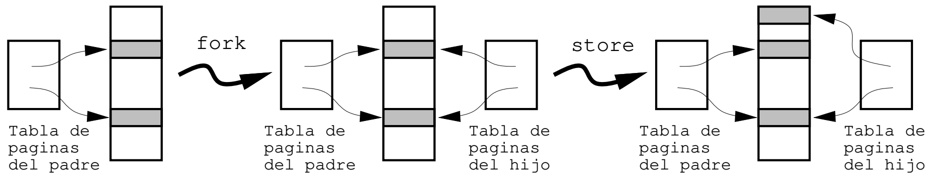

Figura 5.9: Optimización de fork a través de *copy-on-write*.

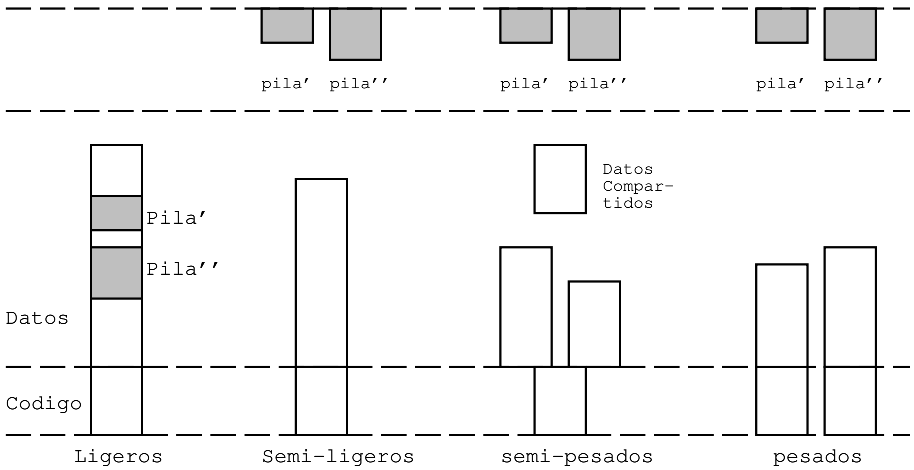

Figura 5.10: Clasificación de los procesos según la porción compartida del espacio de direcciones.

de realizar en un esquema de segmentación: paginamiento en demanda.

En este esquema el núcleo lleva a disco las páginas virtuales que no han sido accesadas por un período prolongado y, a pesar de todo, el proceso propietario de esas páginas puede continuar corriendo, si no las accesa.

El paginamiento en demanda será estudiado en profundidad en la siguiente sección.

5.2.4 Ejemplos de sistemas de paginamiento

Observe que conceptualmente, cada vez que se realiza un acceso a la memoria, es necesario realizar un acceso a memoria adicional para realizar la traducción. De implementarse de esta forma, estaríamos doblando el tiempo de ejecución de cualquier programa, lo que sería inaceptable para los usuarios. A continuación examinamos como se resuelve este problema en casos reales.

Paginamiento en una Sun 3/50

En una Sun 3/50 las páginas son de 8 KB ($k = 13$) y la tabla de páginas posee 1024 filas por lo que el espacio de direcciones virtuales se restringe a 8 MB. Es interesante estudiar este esquema de paginamiento, pues permite apreciar las restricciones de los esquemas primitivos.

La tabla de páginas del proceso en ejecución se guarda en una memoria estática de tiempo de acceso muchísimo más rapido que la memoria dinámica en donde se almacenan datos y programas (la Sun 3/50 no poseía memoria cache). De esta forma la penalización en tiempo de ejecución al realizar la traducción de la dirección virtual es razonable (del orden de un 10% de tiempo adicional en cada acceso a la memoria).

Este mecanismo presenta los siguientes inconvenientes:

- La memoria rápida destinada a la tabla de páginas puede almacenar sólo la tabla del proceso en ejecución. Cada vez se hace un cambio de contexto hay que modificar las 1024 filas en esta memoria. Por esta razón los cambios de contexto entre procesos pesados en una Sun 3/50 son caros en tiempo de CPU (son "pesados").
- El espacio de direcciones virtuales es demasiado estrecho. Este espacio era suficiente en la época de esta estación (años 86-89), pero insuficiente hoy en día.

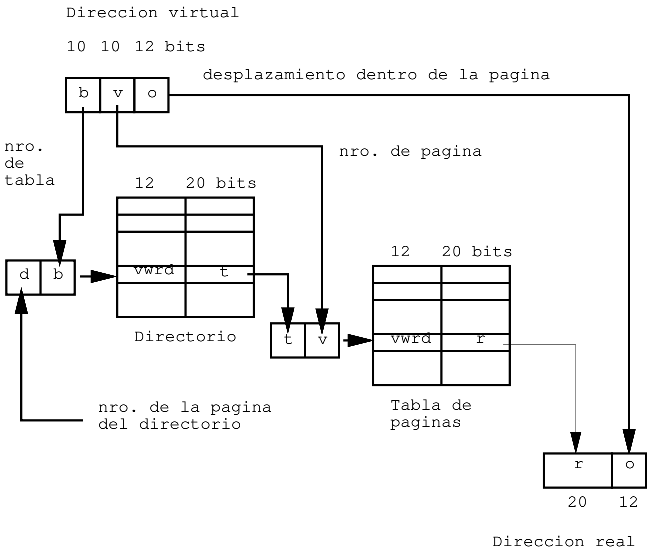

Figura 5.11: Traducción de direcciones virtuales en un procesador Intel 386/486/Pentium.

Paginamiento en un Intel 386/486/Pentium

En esta familia de procesadores, las páginas son de 4 KB ($k = 12$) y el espacio de direcciones virtuales es de 4 GB (2^{32} bytes). Una tabla de páginas está restringida a 1024 filas, pero un proceso puede tener hasta 1024 tablas distintas. Cada tabla de páginas sirve para traducir las direcciones de un bloque contiguo de 4 MB de memoria en el espacio de direcciones virtuales.

Además de las tablas de páginas, existe un *directorio* que indica que tablas posee el proceso en ejecución y cuál es su dirección real en la memoria del computador. El directorio y las tablas de páginas ocupan cada uno una página completa.

El proceso de traducción se observa en la figura 5.11. La organización del directorio es similar a la de una tabla de páginas, sólo que en vez de indicar la traducción de una página, indica la dirección de una tabla de páginas para traducir la páginas que pertenecen a un bloque de 4 MB (una especie de super página). Además en cada fila del directorio hay atributos similares a los de una tabla de páginas. Por ejemplo si el bit V está en 0, quiere decir que todo un bloque de 4 MB en el espacio virtual no está asignado en la memoria real.

El gasto en páginas de memoria para implementar los procesos de un sistema operativo como Unix son los siguientes :

- Una página que contiene el directorio del proceso en ejecución. En cada cambio de contexto es necesario cambiar parcialmente este directorio para reflejar el cambio en el espacio de direcciones.
- Por cada proceso lanzado, se requiere al menos una página para almacenar la tabla de páginas correspondiente al área de código y datos, ubicada en las direcciones bajas de la memoria. Si esta área requiere más de 4 MB se necesita una nueva página por cada 4 MB adicional.
- Por cada proceso lanzado, se requiere una página para almacenar la tabla de páginas correspondiente al área de la pila, ubicada en las direcciones medias del espacio de direcciones (inferiores a 2 GB). Es prácticamente imposible que un programa requiera más de 4 MB para pila, por lo que es razonable prohibir pilas de mayor tamaño.
- Para que en modo sistema se vea por completo la memoria real del computador en las direcciones superiores a 2 GB, se requiere una página por cada 4 MB de memoria real.

El acelerador de la traducción de direcciones : TLB

El directorio y las tablas de páginas se almacenan en memoria real en una Intel 386. Con esto, se reduce el tiempo para realizar un cambio de contexto, pero se multiplica por 3 el costo del acceso a la memoria virtual. En efecto cada acceso a la memoria virtual requiere un acceso real en el directorio para determinar la tabla de páginas, otro acceso real en la tabla de páginas y finalmente el acceso real de la palabra solicitada por el proceso. Este nuevo sobrecosto es escandalosamente inaceptable.

Para acelerar los accesos a la memoria, el hardware del computador posee una TLB (de *translation look-aside buffer*). La TLB es una pequeña memoria asociativa (similar a una memoria caché) de muy rápido acceso que almacena la traducción y los atributos de 32 a 128 páginas virtuales accesadas recientemente (ver figura 5.12).

Cuando se accesa una página cuya traducción se encuentra en la TLB, el sobrecosto es bajísimo. En cambio si la traducción no se encuentra en la TLB, sólo entonces se visitan el directorio y la tabla de páginas en la memoria real, con un sobrecosto de 2 accesos adicionales a la memoria. La traducción de esta página se coloca en la TLB para que los futuros accesos a esta página sean eficientes. Sin embargo, eventualmente será necesario suprimir una traducción previamente existente en la TLB, debido a su tamaño limitado.

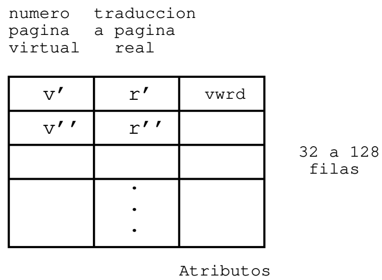

Figura 5.12: Estructura de la TLB : Translation-Lookaside-Buffer.

Empíricamente se ha determinado que más del 99% de los accesos a memoria corresponde a páginas cuya traducción y atributos se encuentran en la TLB.

Una restricción usual es que la TLB mantiene sólo la traducción de páginas pertenecientes al proceso en ejecución. Durante un cambio de contexto es necesario invalidar la TLB, para que no se encuentren en la TLB traducciones que corresponden erróneamente al proceso anterior. Este es un costo escondido del cambio de contexto : el proceso que recibe el procesador encuentra la TLB vacía, la que se irá poblando a medida que el proceso accesa sus páginas virtuales. Para cargar 64 filas de la TLB se habrán requerido 128 accesos adicionales. De todas formas, este sobrecosto es muy inferior al del cambio de contexto en una Sun 3/50.

De esta forma se resuelven los dos problemas de los mecanismos de paginamiento primitivos : el espacio de direcciones virtuales crece a 4 GB y el costo del cambio de contexto se reduce. La mayoría de los procesadores modernos emplean esquemas similares a la excepción de los PowerPC que poseen tablas de páginas invertidas, que por problemas de espacio no podemos discutir este documento.

5.3 Memoria Virtual : Paginamiento en Demanda

Cuando la memoria real de un computador se hace insuficiente, el núcleo del sistema operativo puede emular una memoria de mayor tamaño que la memoria real, haciendo que parte de los procesos se mantengan en disco. A este tipo de memoria se le denomina *memoria virtual*, pues es una memoria inexistente, pero que para cualquier proceso es indistinguible de la memoria real.

El mecanismo que implementa la memoria virtual se denomina *paginamiento en demanda* y consiste en que el núcleo lleva a disco las páginas virtuales de un proceso que tienen poca probabilidad de ser referenciadas en el futuro cercano. Un proceso puede continuar corriendo con parte de sus páginas en disco, pero con la condición de no accesar esas páginas.

La realización de un sistema de memoria virtual se hace posible gracias al *principio de localidad de las referencias* : un proceso tiende a concentrar el 90% de sus accesos a la memoria en sólo el 10% de sus páginas. Sin embargo, para que un proceso pueda accesar *una palabra*, es necesario que la página que contiene esa palabra deba estar completamente residente en memoria. Aún así, empíricamente se observa que un proceso puede pasar períodos prolongados en los que accesa sólo entre un 20 al 50% de sus páginas. El resto de las páginas puede llevarse a disco mientras no son usadas.

Page-fault

Las páginas residentes en disco se marcan en la tabla de páginas del proceso con el bit V en cero, de modo que si el proceso las referencia se produce una interrupción. Esta interrupción se denomina *page-fault*. En la rutina de atención de un page-fault, el núcleo debe cargar en memoria la página que causó el page-fault, por lo que el proceso queda suspendido mientras se realiza la lectura del disco. Cuando esta operación concluye, el proceso se retoma en forma transparente sin que perciba la ausencia temporal de esa página.

La duración de la suspensión del proceso es del orden del tiempo de acceso del disco, es decir entre 8 a 20 milisegundos.

5.3.1 Estrategias de reemplazo de páginas

Toda la dificultad del paginamiento en demanda está en decidir qué páginas conviene llevar a disco cuando la memoria se hace escasa. La idea es que la página que se lleve a disco *debe causar un page-fault lo más tarde posible*. Por lo tanto se puede deducir que el sistema de paginamiento ideal debe llevar a disco aquella página que no será usada por el período de tiempo más largo posible.

Desde luego, en términos prácticos el núcleo no puede predecir por cuanto tiempo una página permanecerá sin ser referenciada, por lo que es necesario recurrir a estrategias que se aproximen al caso ideal. Estas estrategias se denominan *estrategias de reemplazo de páginas*.

Las estrategias de reemplazo de páginas son realizadas por el núcleo del sistema operativos. La componente de código del núcleo que se encarga de esta labor podríamos definirlo como el scheduler de páginas.

Antes de estudiar estas estrategias examinemos el impacto que tiene una tasa elevada de page-faults en el desempeño de un proceso. Para hacer este análisis consideremos los siguientes parámetros.

Parámetro	Significado	Valor aproximado
t_M	Tiempo de acceso a la memoria real	60 nanosegundos
t_P	Tiempo de lectura de una página	12 milisegundos
r	Tasa de page-faults	

Entonces el tiempo promedio de acceso a una palabra en memoria es :

$$t_e = (1 - r)t_M + r * t_P$$

Si la tasa de page-faults es 1 page-fault cada 1000 accesos, el tiempo promedio por acceso es $t_e = 12$ microsegundos, es decir 200 veces más lento que el tiempo de acceso de una palabra residente en la memoria. Esta diferencia abismante se debe a que el tiempo de carga de una página es 200 mil veces el tiempo de acceso a una palabra en memoria real.

Por supuesto no es razonable que en un sistema con memoria virtual una aplicación corra 200 veces más lento que cuando corre en un sistema sin memoria virtual. Para que la penalización de la memoria virtual no exceda más del 10% del tiempo de acceso de una palabra es necesario que :

$$r < \frac{1}{2,000,000}$$

Es decir a lo más un page-fault por cada 2 millones de accesos a la memoria.

A continuación describiremos y evaluaremos las distintas estrategias. Veremos que estas estrategias presentan problemas ya sea de implementación o también de desempeño.

5.3.2 La estrategia ideal

Consiste en llevar a disco aquella página que no será usada por el período de tiempo más largo. Esta estrategia es óptima, pero desde luego, no se puede implementar. Sólo sirve para comparar cifras.

La tabla 5.1 muestra el funcionamiento de esta estrategia ante una traza de accesos a memoria. Esta traza es una lista ordenada con los accesos que hace un proceso a sus páginas virtuales. La tabla muestra por cada acceso a la memoria cuáles son las páginas virtuales que se encuentran en la memoria real. Se observa que con esta estrategia ocurren 9 page-faults en total.

5.3.3 La estrategia FIFO

Entre las páginas virtuales residentes, se elige como la página a reemplazar la primera que fue cargada en memoria.

En la tabla 5.2 se observa el funcionamiento de esta estrategia. En total ocurren 15 page-faults. Aun cuando esta estrategia es trivial de implementar, la tasa de page-faults es altísima, por lo que ningún sistema la utiliza para implementar paginamiento en demanda[1].

5.3.4 La estrategia LRU

LRU significa *Least-Recently-Used*. La estrategia consiste en llevar a disco la página que ha permanecido por más tiempo sin ser accesada. El fundamento de esta estrategia es que estadísticamente se observa que mientras más tiempo permanece una página sin ser accesada, menos probable es que se accese en el futuro inmediato.

En la tabla 5.3 se observa el funcionamiento de esta estrategia. En total ocurren 12 page-faults. Si bien, es posible implementar esta estrategia, sería necesario conocer en qué instante fue accesada cada página. Se podría modificar el Hardware para que coloque esta información en la tabla de páginas en cada acceso, pero en la práctica ningún procesador lo hace. Por lo demás, si esta información estuviese disponible, de todas formas, habría que recorrer todas la páginas en cada page-fault, lo que sería demasiado caro.

5.3.5 La estrategia del reloj

Esta estrategia es una aproximación de LRU. El fundamento es que no es necesario escoger exactamente la página LRU, sino que es suficiente elegir una página que lleva harto tiempo sin ser accesada.

La estrategia usa el bit R de la tabla de página y que el hardware coloca en 1 cuando la página es accesada. El scheduler de páginas coloca en 0 este bit y lo consulta cuando estima que ha transcurrido el tiempo suficiente como para que aquella página sea reemplazada si no ha sido accesada, es decir si el bit R continúa en 0.

La estrategia del reloj ordena las páginas reales circularmente como si fueran los minutos de un reloj (ver figura 5.13). Un puntero señala en cada instante una de las páginas. Inicialmente todas las páginas parten con R en 0. Luego de cierto tiempo algunas páginas han sido accesadas y por lo tanto su bit R pasa a 1.

[1]En cambio sí se usa para implementar estrategias de reemplazo en memorias caché o en TLBs.

	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
a	7	7	7	2	2	2	2	2												
b		0	0	0	0	0	0	4	...											
c			1	1	1	3	3	3												

Tabla 5.1: Reemplazo de páginas para la estrategia ideal

	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
a	7	7	7	2	2	2	2	2												
b		0	0	0	0	3	3	3	...											
c			1	1	1	1	1	4												

Tabla 5.2: Reemplazo de páginas para la estrategia FIFO

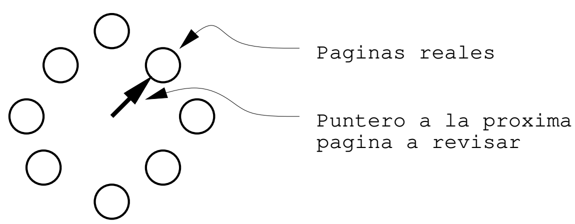

Figura 5.13: Ordenamiento de las páginas en la estrategia del reloj.

Cuando ocurre un page-fault se ejecuta el siguiente procedimiento :

i. Mientras el bit R de la página apuntada sea 1 :
   (a) Colocar el bit R en 0.
   (b) Avanzar el puntero en una página.
ii. La página apuntada tiene su bit R en 0. Elegir esa página para ser reemplazada, es decir que esa página virtual se lleva a disco, lo que libera una página real. Esa página real se usa para cargar la página que causó el page-fault.
iii. Avanzar el puntero en una página[2].

La idea es que la página elegida para ser reemplazada no ha sido accesada por una vuelta completa del reloj. La estrategia considera que este tiempo es suficiente como para suponer que la página no será accesada en el futuro inmediato.

El tiempo que demora el puntero en dar una vuelta depende de la tasa de page-faults. Mientras mayor sea esta tasa, menor será el tiempo en dar la vuelta y por lo tanto el scheduler reemplazará páginas que llevan menos tiempo sin ser accesadas. Al contrario, si la localidad de los accesos a memoria es buena, la tasa de page-faults será baja y el tiempo de una revolución del puntero aumentará. Esto quiere decir que el scheduler tendrá que reemplazar páginas que llevan más tiempo sin ser accesadas.

Una situación anómala se produce cuando todas las páginas tienen el bit R en 1 y por lo tanto el puntero avanza una vuelta completa antes de encontrar el primer bit R en 0. Esto ocurre cuando transcurre mucho tiempo sin que ocurran page-faults y todas las páginas alcanzan a ser accesadas. Esto no es realmente un problema, puesto que ocurre precisamente cuando la tasa de page-faults es muy baja. Una mala decisión en cuanto a la página reemplazada, causará un impacto mínimo en el desempeño del proceso.

Esta estrategia es fácil de implementar eficientemente con el hardware disponible y funciona muy bien con un solo proceso. En caso de varios procesos, la estrategia se comporta razonablemente en condiciones de carga moderada, pero veremos que se comporta desastrosamente cuando la carga es alta.

Trashing

El problema de la estrategia del reloj es que puede provocar el fenómeno de *trashing*. Este fenómeno se caracteriza por una tasa elevadísima de page-faults que hace que ningún proceso pueda avanzar. De hecho, la CPU pasa más del 90% del tiempo en espera de operaciones de lectura o escritura de páginas en el disco de paginamiento.

Una vez que un sistema cae en el fenómeno de trashing, es muy difícil que se recupere. La única solución es que el operador logre detener algunos procesos. Es-

[2] Esta instrucción no es necesaria para que la estrategia funcione. Su presencia significa que una página que acaba de ser cargada de disco, sólo puede volver a ser llevada a disco después de dos vueltas del puntero del reloj.

	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
a	7	7	7	2	2	2	2	4												
b		0	0	0	0	0	0	0	...											
c			1	1	1	3	3	3												

Tabla 5.3: Reemplazo de páginas para la estrategia LRU

to podrá realizarse con mucha dificultad porque incluso los procesos del operador avanzarán "a paso de tortuga".

La explicación de este fenómeno está en la combinación de la estrategia del reloj con un scheduling de procesos del tipo Round-Robin. Las páginas del proceso que se retoma han permanecido largo tiempo sin ser accesadas y por lo tanto es muy probable que el scheduler de páginas las haya llevado a disco. Por lo tanto, cuando un proceso recibe una tajada de tiempo, alcanza a avanzar muy poco antes de que ocurra un page-fault.

El fenómeno de trashing se produce exactamente cuando el puntero del scheduler de páginas comienza a dar vueltas más rapidas que las vueltas que da el puntero del scheduler Round-Robin de los procesos.

5.3.6 La estrategia del Working-Set

Esta estrategia elimina el problema del trashing combinando paginamiento en demanda con *swapping*. La idea es mantener para cada proceso un mínimo de páginas que garantice que pueda correr razonablemente, es decir con una tasa de page-faults baja. Este mínimo de páginas se denomina *working-set*.

Si el scheduler de páginas no dispone de memoria suficiente como para tener cargados los working-set de todos los procesos, entonces se comunica con el scheduler de mediano plazo para que éste haga swapping de procesos. Es decir se llevan procesos completos a disco.

El working-set

Se define $\mathrm{WS}_P(\Delta t)$ como el conjunto de páginas virtuales del proceso P que han sido accesadas en los últimos Δt segundos de tiempo virtual del proceso P.

La estrategia del working-set calcula para un proceso P el valor $\mathrm{WS}_P(\Delta t)$ cada vez que el proceso P completa Δt segundos de uso de la CPU. Este cálculo se realiza de la siguiente forma:

Se consulta el bit R de cada página q residente en memoria real del proceso P:

- Si el bit R de q está en 1, la página está en el working-set y se coloca el bit en 0.
- Sino, la página no ha sido accesada desde hace Δt segundos y por lo tanto no pertenece al working-set de P. La página se agrega al conjunto C que agrupa las páginas candidatas a ser reemplazadas, de todos lo procesos.

Cuando ocurre un page-fault se ejecuta:

i. Mientras C no esté vacío:
   (a) Sea q una página en C.
   (b) Extraer q de C.
   (c) Si el bit R de q está en 0, se retorna q como la página elegida para ser reemplazada.
   (d) Si el bit R está en 1, quiere decir que esa página fue accesada a pesar de no estar en el working-set de un proceso, por lo que se descarta y se contínua la búsqueda con otra página.

ii. El conjunto C está vacío por lo que hay que hacer swapping.

Observe que en esta estrategia si una página pasa al conjunto C, el tiempo transcurrido desde su último acceso está comprendido entre Δt y $2\Delta t$.

Además para que un proceso tenga la oportunidad de correr eficientemente se debe cumplir que su working-set sea inferior al tamaño de la memoria real.

A continuación veremos que esta estrategia se puede optimizar de diversas formas.

Transferencia de páginas a disco

Cuando ocurre un page-fault la estrategia elige una página a reemplazar. La pausa que se introduce en el proceso que causó el page-fault es doble, porque primero es necesario *llevar* a disco la página elegida y luego *cargar* de disco la página del page-fault.

Sin embargo, no es necesario llevar a disco las páginas que ya hayan sido llevadas una vez a disco y no han sido modificadas desde entonces. Esto ocurre especialmente con las páginas de código.

Para evitar llevar nuevamente estas páginas a disco se realiza lo siguiente:

i. Al traer una página q de disco, se marca con el bit D en 0. D es el bit *Dirty*.

ii. El hardware coloca el bit D en 1 si el proceso modifica esa página.

iii. Al reemplazar q, la página se lleva a disco sólo si su bit D está en 1, si no la antigua copia de q en disco es la misma que está actualmente en memoria real.

Esto resuelve el problema de las páginas de código. Sin embargo, resta un conjunto elevado de páginas que todavía hay que llevarlas a disco antes de reemplazarlas. Este tiempo se puede suprimir haciendo que las páginas que se agregan a C (el conjunto de páginas candidatas a ser reemplazadas) se escriban asincrónicamente en disco. Con suerte, cuando producto de un page-fault se escoja esa página para ser reemplazada, ya se tendrá una copia en disco.

Aún así, si el conjunto C es pequeño y la tasa de page-faults es alta, es poco probable que una página alcance a ser escrita en disco. Por esta razón los sistemas operativos tratan de mantener en el conjunto C un 20% de las páginas de la memoria real. Esto garantiza que siempre habrán páginas listas para ser reemplazadas sin que sea necesario llevarlas primero a disco.

Si el tamaño de C está muy por debajo de ese 20%, entonces se comienza a hacer swapping de procesos. Es decir se llevan procesos completos a disco. Al contrario, si se supera ese 20%, se pueden cargar aquellos procesos que estén en el disco.

Período entre cálculos del working-set

El tamaño del working-set de un proceso depende de tres factores:

i. La localidad de los accesos que haga el proceso a la memoria.

   Mientras mejor sea la localidad menor será el tamaño del working-set y por lo tanto menos memoria real necesitará el proceso para correr eficientemente.

   Este es un factor que depende únicamente de las características del proceso. El sistema operativo no puede controlar este factor.

ii. El tiempo Δt entre cálculos del working-set.

   Mientras mayor sea esta cifra, mayor será la probabilidad de que una página haya sido accesada y por lo tanto mayor será el working-set. De hecho, el tamaño del working-set crece monotamente junto con Δt.

   Este factor sí debe ser controlado por el scheduler de páginas del sistema operativo.

iii. El tamaño de la página.

   En la práctica, el tamaño de una página es fijo para una arquitectura dada y no se puede modificar (4 KB en una Intel x86). Pero si fuese posible reducirlo a la mitad, intuitivamente se puede deducir que el tamaño del working-set en número de páginas crecería a un poco menos del doble. Esto se debe a que el proceso accesa frecuentemente sólo una de las mitades de una página.

   Ahora si se considera el tamaño en bytes del working-set (calculado como número de páginas multiplicado por el tamaño de la página), a menor tamaño de página, menor es el tamaño del working-set. Esto significa que el proceso necesita menos memoria real para correr eficientemente. Sin embargo, por otra parte, al disminuir el tamaño de página aumenta el número de las páginas y por lo tanto aumenta el sobrecosto asociado al cálculo de los working-set, lo que resulta negativo en términos de desempeño.

   Por lo tanto no es trivial determinar el tamaño adecuado para las páginas. Pero ésta no es una decisión que efrenta el conceptor del sistema operativo. Es el diseñador de la arquitectura el que fija este valor.

Es decir que el único factor que puede controlar el scheduler de páginas es el valor de Δt. Este valor se hace variar en forma independiente para cada proceso en función del sobrecosto máximo tolerado para el paginamiento en demanda. El principio es que este sobrecosto depende de la tasa de page-faults, y esta última depende a su vez de Δt.

Por ejemplo si cada page-fault cuesta una pausa de 10 milisegundos y la tasa de page-faults de un proceso es de 50 páginas por segundo, entonces por cada segundo de tiempo de uso del procesador se perdería medio segundo en pausas de page-faults. Es decir el sobrecosto de la memoria virtual para el proceso sería de un 50%. Esto muestra que el sobrecosto de la memoria virtual depende de la tasa de page-faults.

La idea es entonces aumentar o disminuir el valor de Δt para ajustar la tasa de page-faults a un nivel fijado como parámetro del sistema. Si se aumenta Δt, aumenta el tamaño del working-set y entonces disminuye la tasa de page-faults.

5.3.7 Carga de binarios en demanda

Al lanzar un proceso no es necesario cargar su binario de inmediato en la memoria real. La carga de binarios en demanda (*demand loading*) consiste en cargar progresivamente las páginas de un binario a medida que el proceso las visita.

Esto se implementa de la siguiente forma:

i. Inicialmente se construye la tabla de páginas del proceso con todas sus páginas inválidas, es decir con el bit V en 0.

ii. Se destina un bit de los atributos de cada página para indicar que todas las páginas se encuentran en el binario. Además en el descriptor de proceso se reserva un campo para el archivo que contiene el binario.

iii. A medida que el proceso se ejecuta, se producen page-faults debido a que se visitan páginas que todavía están en el binario. El scheduler de páginas debe consultar entonces los atributos para determinar en qué lugar se encuentra esa página. En el caso de estar en el binario, la página se carga del archivo binario.

El objetivo de esta técnica es mejorar el tiempo de respuesta al cargar binarios de gran tamaño. Si se carga un binario de 3 MB, probablemente se requiere sólo un 10% de sus páginas para entregar los primeros resultados. Para disminuir el tiempo de respuesta, la carga en demanda lee de disco sólo este 10%, lo que resulta más rápido que cargar los 3 MB completamente.

5.3.8 Localidad de los accesos a la memoria

Como dijimos anteriormente la localidad de los accesos a la memoria es un factor que depende únicamente del proceso. En particular depende de los algoritmos que utilice el proceso para realizar sus cálculos.

Los procesos que manipulan matrices u ordenan arreglos usando *quick-sort* son ejemplos de procesos que exhiben un alto grado de localidad en sus accesos. Al contrario, los procesos que recorren aleatoriamente arreglos de gran tamaño exhiben pésima localidad en los accesos.

La programación orientada a objetos (OOP) produce programas que tienen mala localidad. Este tipo de programas crea una infinidad de objetos de pequeño tamaño que se reparten uniformemente en el heap del área de datos del proceso. Si bien existe una buena localidad en el acceso a los objetos, frecuentemente una página contiene al menos uno de los objetos que están siendo accesados. Por lo tanto, a la larga son pocas las páginas que no quedan en el working-set.

El problema de la localidad en OOP se podría resolver utilizando páginas de tamaño similar al de los objetos[3]. Sin embargo aquí caemos nuevamente en un sobrecosto desmedido para administrar un número elevado de páginas.

[3] Como el tamaño de la líneas de la memoria caché.