ASI

Primera Parte: S.O.

3º Ingeniería de Telecomunicaciones — UPV/EHU

Actualizado por última vez el 20 de junio de 2017

"*Under-promise and over-deliver.*"

Javier de Martín – 2016/17

1. Introducción

1.1. Introducción

Los ordenadores están equipados con una capa de software llamada **sistema operativo**, su trabajo es proveer programas con una interfaz para manejar los componentes del ordenador de una forma más simple.

El **procesador** corre en dos **modos**:

- **Usuario**: Dispone un conjunto limitado de instrucciones.
- **Kernel o Supervisor**: Tiene acceso completo a todo el hardware y puede ejecutar cualquier instrucción que la máquina puede ejecutar. El Sistema Operativo es software que corre en modo kernel y provee la base para cualquier otro software que se ejecute.

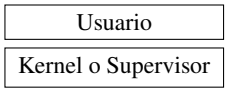


Figura 1: Modos del procesador

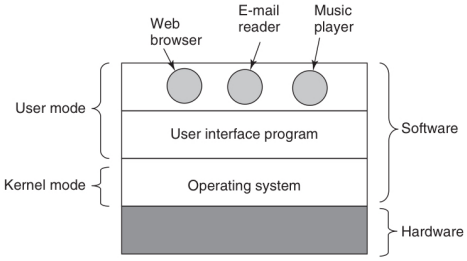
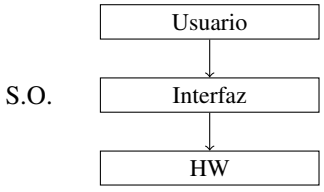


Figura 2: Dónde encaja el sistema operativo

1.1.1. Interface

La **interfaz de usuario** es el nivel más bajo de software ejecutable en modo de usuario, permite al usuario arrancar otros programas.

Es importante distinguir entre el sistema operativo y software normal (modo de usuario). El usuario es libre de cambiar lo último pero no es posible modificar el sistema operativo.



(Máq. Extendida, Máq. Virtual)

Figura 3: Interfaz de usuario

1.1.2. Gestor de Recursos

El sistema operativo es provee abstracciones a programas como una visión descendente. Una alternativa, es un enfoque ascendente, que dice que el sistema operativo está ahí para gestionar todas las piezas del sistema complejo.

La misión de un **gestor de recursos** es ofrecer una asignación ordenada y controlada de los procesadores, memorias y dispositivos entre los diversos programas que compiten por ellos.

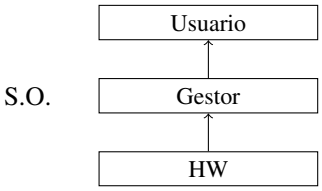


Figura 4: Gestor de Recursos

1.2. Conceptos de los S.O.

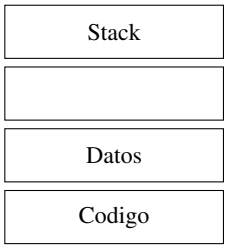
1.2.1. Introducción

Una **llamada al sistema** es un mecanismo utilizado por una aplicación para solicitar un servicio al sistema operativo.

1.2.2. Procesos

Un **proceso** es básicamente un programa en ejecución. Asociado con cada proceso hay un **espacio de direcciones**, una lista de posiciones de memoria las cuales puede utilizar un proceso para leer y escribir. El espacio de direcciones contiene el programa ejecutable, sus datos y su *stack*.

La información de cada proceso se guarda en la **tabla de procesos**, que es un array de estructuras, uno para cada proceso que existe.



+ Entrada en la tabla de procesos

Figura 5: Espacio de direccionamiento de un proceso

Los procesos se rigen por una jerarquía de padres-hijos. Un padre puede crear procesos hijos.

1.2.3. Archivos

Para proveer un lugar para mantener archivos se tiene el concepto de **directorio** como una forma de agrupar ficheros.

En UNIX hay **archivos especiales**, son archivos que se suministran para hace que los dispositivos I/O parezcan archivos. De esa forma pueden ser leídos y escritos utilizando las mismas llamadas al sistema que son utilizadas para leer y escribir archivos.

Existen dos tipos de archivos especiales:

- **Bloque**: Utilizados para modelar servicios que consisten en una colección de bloques aleatoriamente direccionables, como discos.
- **Carácter**: Utilizados para modelar dispositivos que aceptan o ponen en la salida un flujo de caracteres.

1.2.4. Pipe

Un **pipe** es un pseudo-archivo que puede ser utilizado para conectar dos procesos para hacer posible la comunicación entre ellos.

1.3. Llamadas al sistema

1.3.1. Introducción

Los sistemas operativos tienen dos funciones principales:

- Proveer abstracciones a los programas de usuarios.
- Gestionar los recursos de los programas.

Es importante recordar que cualquier ordenador con una única CPU puede ejecutar una única instrucción a la vez.

Si un proceso está corriendo un programa de usuario en modo de usuario y necesita un servicio del sistema tiene que ejecutar una **instrucción trap** para transferir el control al sistema operativo. Las llamadas al sistema se hacen en una serie de pasos.

Una instrucción TRAP difiere de una instrucción procedure-call de dos formas fundamentales:

- Cambia a modo kernel. Mientras que la procedure-call no cambia el modo.
- En lugar de dar una dirección relativa o absoluta donde se encuentra el procedimiento, la instrucción TRAP no puede saltar a una dirección arbitraria. Dependiendo de la arquitectura, salta a una ubicación fija, o hay un campo de la instrucción, que le dará una tabla en memoria que contenga la dirección a donde ir o similar.

1.4. Estructura de un S.O.

1.4.1. Monolítico

Es la organización más común. El sistema operativo corre como un único programa en modo kernel y se escribe como una colección de procedimientos, enlazados entre ellos como un único archivo binario ejecutable.

Cuando se utiliza esta técnica, cada procedimiento del sistema es libre de llamar a cualquier otro procedimiento, siendo esto bastante eficiente.

Pero tener la posibilidad de que muchos procesos que se puedan llamar entre ellos sin restricción puede llevar a que sea díficil de entender. También, un fallo en cualquiera de estos procedimientos tirará el sistema operativo al completo. La estructura básica de una organización monolítica:

1. Programa principal que invoca a la subrutina requerida
2. Subrutinas de servicio que ejecutan las tareas solicitadas
3. Subrutinas de utilidad (ayuda) a subrutinas de servicio

1.4.2. Capas

Consiste en generalizar el sistema operativo como una jerarquía de capas, cada una construida sobre la anterior.

1.4.3. Modelo Cliente Servidor

Una pequeña variación del modelo microkernel es la idea de distinguir dos clases de procesos, los **servidores**, que proveen algún servicio, y los **clientes**, que utilizan estos servicios. Este modelo se conoce como **modelo cliente-servidor**.

La comunicación entre clientes y servidores es generalmente por envío de mensajes.

ASI

Primera Parte: S.O.

3º Ingeniería de Telecomunicaciones — UPV/EHU

Actualizado por última vez el 20 de junio de 2017

"*Under-promise and over-deliver.*"

Javier de Martín – 2016/17

2. Gestión de Procesos

2.1. Introducción

Un **proceso** es una instancia de un programa en ejecución. Conceptualmente, cada proceso tiene su propia CPU virtual.

La diferencia entre un proceso y un programa es que un proceso es algo similar a una actividad. Un programa es algo que puede ser guardado en disco no haciendo nada.

Si un programa corre dos veces cuenta como dos procesos.

2.1.1. Modelo

- El **paralelismo** es la ejecución de varias actividades simultáneamente en varios procesadores.
- El **pseudo-paralelismo** es la ejecución de varias actividades en un mismo procesador.

La **concurrencia** es la existencia de varias actividades ejecutándose simultáneamente que necesitan de sincronización para actuar en conjunto.

Cuatro principales eventos provocan la **creación de procesos**:

1. Inicialización del sistema
2. Ejecución de una llamada al sistema de creación de procesos por un proceso que está corriendo.
3. Petición del usuario de crear un nuevo proceso.
4. Iniciación de una tanda de trabajos

Los procesos que se mantienen en segundo plano para gestionar alguna actividad se llaman **demonios** (*daemons*).

En UNIX sólo hay una forma de hacer una llamada al sistema para crear un proceso `fork`, crea una copia exacta del proceso llamante. Después del `fork`, cada proceso tienen las mismas variables.

Cuando un usuario escribe un comando en la consola, ésta hace un proceso hijo y éste ejecuta el comando introducido.

Un proceso hijo puede formar más procesos, formando una **jerarquía de procesos**. A pesar de que cada proceso es una entidad independiente cada uno puede encontrarse en un estado diferente.

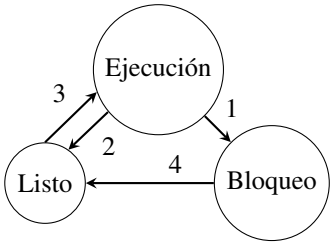


Figura 6: Modelo Básico de los 3 estados de un proceso

1. **Ejecución**: Utiliza la CPU en ese instante
2. **Listo**: Temporalmente detenido para dejar correr a otro proceso)
3. **Bloqueado**[1]: Imposible de correr hasta que un evento externo ocurre

Un proceso **terminará** si:

1. Salida normal (voluntaria)
2. Error exit (voluntaria)
3. Error fatal (involuntario)
4. Muerto por otro proceso (involuntario)

La mayoría de procesos termina porque ha terminado su trabajo.

2.1.2. Implementación

El Sistema Operativo para implementar el modelo de proceso crea una **tabla de procesos**, que no es más que un array de estructuras (una por cada proceso). Se añaden campos de la estructura de cada proceso en la tabla de procesos y esqueleto de lo que hace el S.O cuando se produce una interrupción.

2.2. Comunicación entre Procesos

2.2.1. Condiciones de Carrera

En algunos sistemas operativos, procesos que trabajen juntos pueden compartir espacio común del cual pueden leer y escribir. Hay que prohibir a más de un proceso leer y escribir de la memoria compartida al mismo tiempo.

2.2.2. Secciones Críticas

¿Cómo evitar condiciones de carrera? Hay que prohibir que más de un proceso lean o escriban datos compartidos a la vez, llamado **exclusión mutua**.

Una **sección crítica** son partes del programa donde se accede a las zonas compartidas.

Condiciones que implican una buena solución a las condiciones de carrera:

1. Dos procesos no podrán estar a la vez dentro de sus secciones críticas
2. No se deberán asumir suposiciones sobre velocidades relativas de los procesos
3. Ningún proceso fuera de la sección crítica podrá bloquear otros procesos
4. Ningún proceso esperará indefinidamente para entrar en su sección crítica.

2.2.3. Exclusion Mutua con Espera Activa

Se van a examinar varias propuestas para conseguir la exclusión mutua, mientras que un proceso está ocupado actualizando la memoria compartida en su región crítica, ningún otro proceso entrará su región crítica y causará problemas.

Hay 5 formas:

1. **Inhabilitación de interrupciones**: Solución más simple para sistemas con un único procesador. Cada vez que un proceso entre en la región crítica deshabilitará todas las interupciones y las volverá a

[1] Por ejemplo, cuando un proceso lee de un pipe y no hay nada disponible el proceso se bloquea automáticamente.

activar antes de salir. Con las interrupciones desactivadas, no pueden ocurrir interrupciones de reloj. No es óptima ya que no es conveniente dar a los procesos la posibilidad de deshabilitar interrupciones. Hoy en día la posibilidad de conseguir exclusión mutua por deshabilitación de interrupciones es cada vez menos común debido a que los procesadores tienen más núcleos. Mientras que en un núcleo se desactivan las interrupciones en el otro no se evita que se interfiera en las operaciones.

2. **Variables Candado**: Es una solución por software, se emplea una variable compartida `lock` inicializada a `0`. Cuando un proceso quiere acceder a la región crítica.

   - Mira el valor de la variable `lock`
   - Si `lock = 0`: El proceso la pone a 1 y entra en su región crítica.
   - Si `lock = 1` el proceso espera hasta que se cambie a `0`.

   Cuando la variable está a `0` no hay ningún proceso en su región crítica y si `1` algún proceso está en su región crítica. Desafortunadamente, este método contiene un fallo. Si un proceso leyera el candado y viera que es cero. Antes de que pueda poner el candado a `1` otro proceso lo pone a `1`. Cuando el primer proceso vuelva a ejecutarse, también pondrá el candado a `1` y dos procesos estarán en la región crítica a la vez.

3. **Alternancia Estricta**: Solución por software, evita condiciones de carrera pero no cumple la condición 3[2].

```
  while(TRUE) {
   while(turn != 1)
critical_section();
turn = 0;
noncritical_section();
  }
```

   La comprobación continua de variables hasta que un valor aparece se llama ***busy waiting***, debería ser evitado ya que desperdicia tiempo de uso de la CPU. Sólo cuando hay una necesidad estricta de espera se debería utilizar esta espera. Un candado que utiliza *busy waiting* se le llama ***spin lock***.

4. **Solución Peterson**: Combina la idea de tomar turnos con la idea de variables candado.

```
  #define FALSE 0
  #define TRUE  1
  #define N     2

  int turn;
  int interested[N];

  void enter_region(int process) {

   int other;

other = 1 - process;
interested[process] = TRUE;
turn = process;

while(turn == process && interested[other] == TRUE)
  }

  void leave_region(int process) {

   interested[process] = FALSE;
  }
```

5. **Instrucción TSL** (TEST and SET LOCK): Solución por software con ayuda de harware. TSL es una operación indivisible. Copia el contenido de la variable en un registro.

2.2.4. Dormir y Despertar

Peterson y TSL son soluciones buenas pero tienen un defecto, espera activa e incumplen las reglas de planificación.

Las **primitivas de comunicación** son las operaciones para conseguir la condición de exclusión mutua. Bloquean el proceso cuando no se les permite entrar en su región crítica (no consumen tiempo de CPU).

- `SLEEP` es una llamada que suspende el proceso que la realiza hasta que otro proceso lo despierte
- `WAKEUP` es una llamada que tiene por parámetro el proceso a despertar.

El **problema productor-consumidor**[3]. Por ejemplo dos procesos comparten un mismo buffer de tamaño fijo. El productor, pone información en el buffer y el otro, el consumidor la extrae. EL problema nace cuando el productor quiere poner un nuevo item en el buffer pero está lleno. La solución para el productir es dormirse y ser despertado cuando el consumidor haya liberado uno o más items. Similarmente, si el consumidor quiere eliminar uun item del buffer y ve que el buffer está vacío se duerme hasta que el productor pone algo en el buffer y lo despierta.

2.2.5. Semáforos

Solución atómica que resuelve el problema de pérdidas de llamadas de wakeup. Implementado con una variable entera que tiene dos operaciones:

- `DOWN`: Comprueba si el valor del semáforo > 0
  - SI: Decrementa una unidad su contenido
  - NO: Es cero, el proceso se duerme (bloquea)
- `UP`: Comprueba si el valor del semáforo > 0
  - SI: Incrementa una unidad su contenido
  - NO
    - Ningún proceso dormido - Pone a 1 el semáforo
    - Algún proceso dormido - Aleatoriamente se despierta alguno y el semáforo permanece a 0.

[2] Ningún proceso fuera de la sección crítica podrá bloquear a otros procesos.

[3] Problema de ejemplo de problema de sincronización de multipcocesos. Se definen dos procesos, productor y consumidor, ambos comparten un buffer de tamaño infinito. La tarea del productor es generar un producto, almacenarlo y comenzar nuevamente; mientras que la del consumidor toma (simultáneamente) productos uno a uno. El problema consiste en que el producto no añada más productos que la capacidad del buffer y que el consumidor no intente tomar un producto si el buffer está vacío. La idea para la solución es que ambos procesos se ejecuten simultáneamente y se despierten o duerman según el estado del buffer. Concretamente, el productor agrega productos mientras quede espaio y en el momento en que se llene el buffer se pone a dormir. Cuando el consumidor toma un producto notifica al productor que puede comenzar a trabajar nuevamente. En el caso contrario, si el buffer se vacía, el consumidor se pone a dormir y en el momento enque el productor agrega un producto crea una señal para despertarlo.

Con la implementación de `UP` y `DOWN` el S.O. inhabiblita las interrupciones durante las operaciones a un semáforo.

Para resolver el **problema productor-consumidor** y hacer funcionar correctamente a los semáforos, es esencial implementarlos de una forma indivisible. La forma normal es implementar `up` y `down` como llamadas al sistema, con el sistema operativo desactivando brevemente todas las interrupciones cuando prueba el semáforo, actualizándolo y poniendo los procesos a dormir si es necesario. Como todas estas acciones solo llevan para ejecutarse unas pocas instrucciones, no se producen daños deshabilitando interrupciones.

Si hay múltiples CPUs en uso, cada semáforo tiene que ser protegido por una variable *lock*.

La solución que se propone utiliza tres semáforos:

- *fool*: Lleva la cuenta de los slots que están llenos.
- *empty*: Para contar es igual al número de slots en el buffer.
- *mutex*: Inicialmente vale a 1.

Los semáforos que se inicializan a 1 y son usados por dos o más procesos para asegurar que únicamente uno de ellos puede entrar a su región crítica al mismo tiempo se llaman **semáforos binarios**. Si cada proceso hace un `down` justo antes de entrar a su región crítica y `up` después de salir, **se asegura la exclusión mutua**.

Ahora que se tiene una buena primitiva de comunicación entre procesos. En un sistema que usa semáforos, la forma natural de esconder las interrupciones es tener un semáforo, inicialmente puesto a `0`, asociado con cada dispositivo I/O. Justo después de arrancar un dispositivo I/O, el proceso gestor hace un `down` al semáforo asociado, bloqueando inmediatamente. Cuando llega la interrupción, el gestor de interrupciones hace un `up` al semáforo asociado, que hace que el proceso relevante asociado corra de nuevo.

El otro uso de semáforos es para la **sincronización**. Los semáforos *full* y *empty* son necesarios para garantizar que ciertos eventos de secuencias ocurran o no. En este caso, se aseguran que el productor deja de funcionar cuando el buffer está lleno, y que el consumidor pare cuando está vacío. Este uso es diferente al del uso para exclusión mutua.

2.2.6. Contadores de Eventos

El problema productor/consumidor lo solucionan los semáforos con exclusión mutua. Los contadores de eventos sin solución mutua. Se define una variable entera con 3 operaciones:

- `READ (E)`: Devuelve el valor del contador de eventos `E`.
- `ADVANCE (E)`: Incrementa el valor de `E` en `1` (Atómicamente)
- `AWAIT (E, v)`: Espera hasta que $\texttt{E} \geq \texttt{v}$

Como característica tiene que los contadores de eventos siempre crecen, nunca decrecen y empiezan desde `0`.

2.2.7. Mutexes

Cuando la habilidad de contar de un semáforo no se necesita, se implementa una vesrión simplificada de los semáforos, los mutex. Un mutex es una variable compartida que puede estar en dos estados: bloqueada o desbloqueada.

2.2.8. Monitores

Los **monitores** son un acolección de procedimientos, variables y estructuras de datos que son agrupadas por un paquete o módulo. Los procesos pueden llamar a los procedimientos de un monitor siempre que quieran, pero no pueden acceder directamente a las estructuras de datos internas desde procedimientos declarados fuera del monitor. Los monitores tienen una propiedad importante que los hace útiles para alcanzar la exclusión mutua: un único proceso puede estar activo dentro de un monitor en cualquier instante. Aunque los monitores provean una forma sencilla de conseguir la exclusión mutua, no es suficiente. También se necesita una forma de bloquear procesos cuando no pueden continua.

En el **problema productor-consumidor** es tan fácil como colocar las pruebas de si el buffer está lleno o vacío dentro de los procedimientos de los monitores. Pero, ¿cómo debe de bloquear e productor cuando el buffer está lleno? La solución recae en la introducción de **variables de condición** junto con dos operaciones sobre ellas, `wait` y `signal`. Cuando un procedimiento de un monitor descubre que no puede continuar (el buffer está lleno), realiza un `wait` sobre alguna variable de condición. Esta acción causa al proceso llamante que se bloquee. También, permite a otro proceso que previamente haya sido prohibido de entrar al monitor entrar ahora. Las variables de condición no son contadores, no acumulan señales para uso posterior como hacen los semáforos.

Propiedades:

1. Sólo un proceso puede estar activo en un monitor en cualquier instante $\rightarrow$ Exclusión mutua.
2. Permite el bloqueo de procesos dentro del monitor, espera activa, a partir de la utilización de variables de condición y las operaciones `WAIT` y `SIGNAL`.
   - `WAIT` sobre una variable de condición $\rightarrow$ El proceso se bloquea permitiendo a otro proceso entrar en el monitor.
   - `SIGNAL` sobre la variable de condición en la que se ha dormido un proceso $\rightarrow$ Despierta al proceso.

Hay dos formas de evitar que dos procesos permanezcan activos dentro del monitor después de una operación `SIGNAL`:

1. Hoare: Continúa el proceso recién despertado, suspendiendo el otro
2. Hansen: El proceso que realiza `SIGNAL` deberá abandonar el monitor (`SIGNAL` aparecerá únicamente al final de una función monitor.

Tiene un inconveniente, que utiliza lenguaje concepto y el compilador debe reconocerlo.

2.2.9. Transferencia de mensajes

Ninguna de las primitivas de comunicación anteriores permiten el intercambio. Hay 2 primitivas de intercomunicación entre procesos:

1. `SEND (destino, mensaje)`: Envía un mensaje a un destino dado
2. `RECEIVE (origen, mensaje)`; Recibe un mensaje de un origen dado (Si no hay mensaje disponible el proceso receptor se bloquea hasta que llegue alguno).

Las dos primitivas del método de comunicación entre procesos son llamadas al sistema en lugar de *language constructs* como en los monitores por lo que pueden ser concluidos fácilmente en librerías.

Los sistemas de envío de mensajes tienen muchos problemas que no aparecen con semáforos o monitores, especialmente si los procesos a comunicar están en máquinas conectadas por la red. Por ejemplo, aparecen problemas relacionados con las pérdidas en la red, se necesitarán mensajes especiales de **confirmación** de recepción.

¿Cómo resolver el **problema productor-consumidor** en el paso de mensajes sin memoria compartida? Suponiendo que todos los mensajes son del mismo tamaño y que todos los mensajes enviados pero no recibidos son guardados en el buffer automáticamente por el S.O. El consumidor empieza enviando `N` mensajes vacíos el productor. Siempre que el productor tenga un item que dar al consumidor, coge un mensaje vacío y devuelve uno lleno. De esta forma, el número total de mensajes en el sistema se mantiene constante en el tiempo, por lo tanto pueden ser guardados en un espacio de memoria conocido. Si el productor trabaja más rápido que el consumidor, todos los mensajes acabarán llenos, esperando al consumidor; el productor será bloqueado, esperando a una respuesta vacía. Si el consumidor trabaja más rápido, ocurre lo contrario: todos los mensajes serán vacío esperando al productor que los llene; el consumidor será bloqueado, esperando a un mensaje lleno.

1. Pipes: Todos los mensajes del mismo tamaño, el S.O. almacena mensajes enviados y no recibidos.

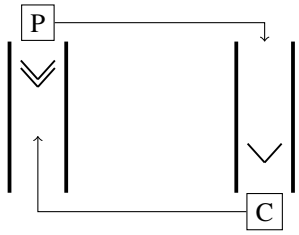


Figura 7: Transferencias de mensajes

2. Buzones (*mailbox*): Emplea un buzón que es una estructura de datos, es una zona de memoria donde caben un cierto número de mensajes. Con buzones, los parámetros de dirección empleados en las llamadas `SEND` y `RECEIVE` son buzones en lugar de procesos.

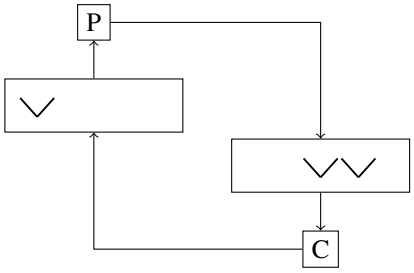


Figura 8: Transferencias de mensajes

   El almacenamiento, el buzón destino coge los mensajes que han sido enviados al proceso de destino pero que aun no han sido recibidos. Cuando un proceso intenta hacer un envío a un buzón que está lleno, es suspendido hasta que se extrae un mensaje.

3. Rendez Vous: Elimina todo el almacenamiento.

   - Si `SEND` antes que `RECEIVE` el proceso `SEND` se bloquea.
   - Cuando se produce `RECEIVE` se copia el mensaje directamente del proceso que lo envía al que lo recibe.

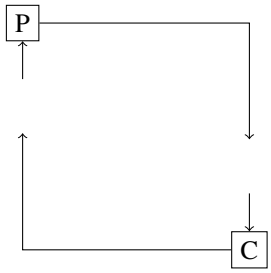


Figura 9: Transferencias de mensajes

2.2.10. Equivalencia de Primitivas

Hay más formas de intercomunicación entre procesos: secuenciadores, expresiones path y serializadores. Todas responden a esquemas similares.

Implementación de monitores a partir de semáfotos:

- Monitor: S(mutex) $\leftrightarrow$ 1Un sólo proceso dentro del monitor
- Variable condición S(c) $\leftrightarrow$ 0 Bloquear/despertar procesos dentro del monitor

La forma de implementar esto:

- Cuandos e llama a una subrutina monitor $\rightarrow$ DOWN (mutex)
- Cuando se abandona una zona monitor $\rightarrow$ UP (mutex)
- `WAIT` a una variable condición `c`:
  - UP (mutex)
  - DOWN semáforo asociado a variable c
  - DOWN (mutex)
- `SIGNAL` a una variable condición c:
  - UP semáforo asociado

Implementación de mensajes a partir de semáforos:

- Proceso: S(P) "0'Bloqueo cuando no se pueda completar `SEND`/`RECEIVE`
- Buzones: S(mutex) "1Ún único proceso inspeccionando los buzones
- Cola Send - Lista de procesos que intentan enviar un mensaje al buzón
- Cola Receive - Lista de procesos que intentan extraer un mensaje del buzón

Forma de interpretación:

1. DOWN (mutex)
2. Mirar variables celdas llenas/vacías
   - Si no vacío o lleno
     - Introducir/extraer mensaje
     - Actualizar variables y lincar
   - Si vacío o lleno
     - `RECEIVE` (vacío):
       - Meterse en la cola RECEIVE
       - UP (mutex) $\rightarrow$ 1 (entrada otros procesos)

    - `DOWN` a su semáforo (bloqueo)
    - DOWN (mutex)
  - `RECEIVE` (lleno)
    - Extraer mensaje, actualizar y lincar
    - Mirar cola SEND, si alguien duerme quitar el primero de la cola y up a su semáforo (desp)
  - `SEND` (lleno)
    - Meterse en la cola SEND
    - UP (mutex)
    - DOWN a su semáforo
    - DOWN (mutex)
  - `SEND` (vacío):
    - Introducir mensaje, actualizar y lincar
    - Mirar la cola `RECEIVE`, si alguien duerme quitar primero de la cola y up a su semáforo
3. UP (Mutex)

Implementación de semáforos a partir de monitores:

- Para un proceso una variable de condición
- Para un semáforo un contador y una lista encadenada

Forma de implementación:

- Procedure `DOWN`
  - if count = 0 → meter el proceso en la lista y WAIT (variable condición)
  - else → decrementar el contador
- Procedure `UP`:
  - if count = 0
    - Comprobar lista
      - ◇ if vacia → incrementar contador
      - ◇ else → eliminar primero de la lista
    - `SIGNAL` (variable condición)
  - `else` → Incrementar contador

Implementación de mensajes a partir de monitores

- Para un proceso una variable condición
- Par aun buzón ESTRUCTURA DE NO SE QUE MIERDAS CON MUCHAS COSAS

TERMINAR ESTA MIERDA DE SECCION QUE NO SE A DONDE ME LLEVARA

2.3. Planificación

Cuando un ordenador es multiprogramado, tiene frecuentemente multitud de procesos o threads compitiendo por la CPU al mismo tiempo.

Esta situación se produce cuando dos o más están simultáneamente en el estado listo. Si sólo una CPU está disponible, se ha de hacer una elección sobre qué proceso correr después, de esto se encarga el **planificador**.

Características de los algoritmos de planificación:

1. **Imparcialidad**: Asegurarse que cada proceso tiene su parte justa de la CPU.
2. **Eficiencia**: Mantener la CPU ocupada el 100 % del tiempo.
3. **Tiempo de respuesta**: Minimizar el tiempo de respuesta para usuarios interactivos.
4. ***Turnaround***: Minimizar el tiempo que una tanda de usuarios tiene que esperar para obtener respuesta.
5. ***Throughput***: Maximizar el número de trabajos procesados por hora.

El **reloj interno** a cada interrupción el S.O. toma el control y decide el siguiente proceso que se ejecutará.

2.3.1. Planificación en Sistemas Batch

- **First-Come, First-Served**: Algoritmo más simple. Los procesos son asignados a la CPU en el orden en el que lo piden. Básicamente hay una única cola para los procesos que están listos para ser ejecutados. La ventaja de este algoritmo es que es sencillo de entender y fácil de programar. Su desventaja es que los procesos de corta duración se ven solapados por los de larga.
- **Shortest Job First**: Se conocen previamente los tiempos de ejecución. Al ejecutar primero los trabajos más cortos se produce el menor tiempo medio de respuesta. Este algoritmo es óptimo cuando todos los trabajos están disponibles simultáneamente.

2.3.2. Planificación en Sistemas Interactivos

- **Round Robin**: A cada proceso se le asigna un intervalo de tiempo en el cual puede ejecutarse, llamado ***quantum***. Si el proceso sigue corriendo al final del *quantum*, la CPU le pasa el control a otro proceso. Si el proceso se ha bloqueado o ha finalizado su ejecución antes de consumir su *quantum* la CPU intercambia el proceso. Es sencillo de implementar, lo único que necesita el planificador es mantener una lista de los procesos que están listos para ejecutarse. El principal problema con este algoritmo es la determinación de la duración del *quantum*.
- **Prioridades**: Round-robin asume implícitamente que todos los procesos son igual de importantes. La asignación de prioridades puede ser estática[4] y dinámica[5].
- **Colas Múltiples**: Variación de la planificación en clases de prioridad. La condición es que siempre que un proceso emplea la totalidad del *quantum* asignado se le baja una clase.
- **Predeterminada**: Consiste en adquirir un compromiso con el usuario y cumplirlo.
- **A Dos Niveles**: Los tiempos de intercambio entre la memoria principal y el disco duro son uno o dos órdenes de magnitud superiores a los de intercambio entre los procesos que se encuentran en la memoria principal.

2.3.3. Evaluación

Existen diferentes métodos de evaluación:

- **Métodos Analíticos**: Consisten en aplicar el algoritmo seleccionado a una determinada carga del sistema y producir una fórmula o número que evalúa la eficacia del algoritmo para esa carga.
  - Modelo determinista: Se toma una carga particular establecida con anterioridad y se determina la eficacia de cada algoritmo para esa carga:
    - Produce números exactos y permite la comparación de algoritmos

[4] No cambia, es de fácil implementación y no responde a cambios de entorno

[5] Dispone de mecanismos para su modificación, tiene esquemas más difíciles de implementar y son sensibles a cambios de entorno

    - Requiere números exactos en sus datos de entrada y los resultados obtenidos se pueden aplicar únicamente a esos datos.
  - Modelo de colas: Según este modelo, el ordenador se describe como un servidor, donde se determinan las distribuciones de probabilidad tanto del tiempo de llegada entre procesos como del tiempo de empleo de cPU. A partir de estas dos distribuciones se pueden calcular los valores medios de throughput, utilización, tiempo de espera...
    - Las distribuciones de llegadas y servicio se definen por relaciones matemáticas que muchas veces suponen hacer simplificaciones que restan exactitud al método, por ser éstas sólo una aproximación a los sistemas reales.

- **Simulación**: Permite conseguir una evaluación más precisa de los algoritmos de evaluación (aunque siempre nos referimos a niveles de confianza). Supone realizar e integrar una serie de tareas entre las que destacan:
  - Modelado y validación del sistema
  - Modelado y validación de los datos
  - Diseño de los experimentos
  - Validación de resultados
- **Implementación**: Única forma exacta de evaluar un algoritmo de planifiacción. Consiste en introducir el algoritmo en el S.O. para ver cómo responde bajo condiciones operativas reales.

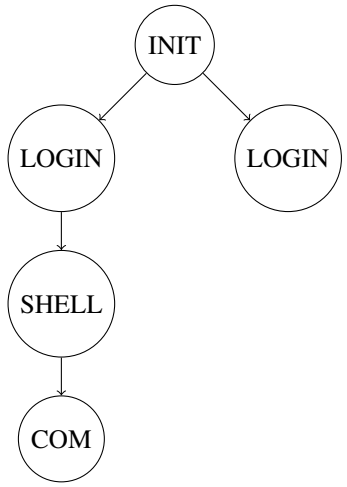


Figura 10: Árbol de procesos de MINIX

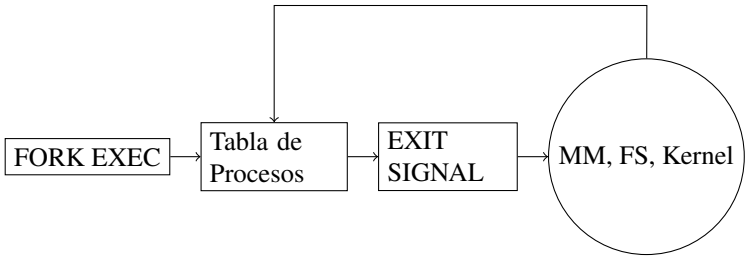


Figura 11: Transferencias de mensajes

2.4. Tratamiento de Procesos MINIX

2.4.1. Estructura Interna

1. **Capa 1**: Modelo de procesos secuenciales independientes que se comunican utilizando mensajes. Tiene 2 tareas
   - Captura de traps e interrupciones (lenguaje ensamblador)
   - Gestión de mensajes (lenguaje C)
2. **Capa 2**: Procesos de E/S → Tareas E/S → Drivers de dispositivo + System Taks. El kernel es capa 1 mas capa 2. Un único programa ejecutable que corre en modo kernel.
3. **Capa 3**: Servicio MM + Servicio FS. el S.O. se encarga de
   - Gestión de recursos - Kernel (capa 1 y 2)
   - Máquina extendida - MM y FS (capa 3)
4. **Capa 4**: Procesos de usuario → Shell, editores, compiladores...

2.4.2. Gestión de Procesos

MINIX - Jerarquía de procesos en forma de árbol donde la raíz es INIT

2.4.3. Comunicación entre Procesos

Mediante el intercambio de mensajes de tamaño fijo (estructura mensaje), se utilizan 3 primitivas para el envío y recepción de mensajes:

- `send`
- `receive`
- `send_rec`

Cada proceso puede enviar y recibir mensajes de los procesos de su propia capa o de aquellos en capas contiguas.

MINIX utiliza el método rendez vous, no necesita gestión de almacenamiento.

2.4.4. Planificación de Procesos

Planificador de clases de prioridad con 3 niveles:

1. Nivel 0 (máxima prioridad) - Tareas capa 2
2. Nivel 1 (prioridad intermedia) - Tareas capa 3
3. Nivel 2 (mínima prioridad) - Tareas capa 4

Las tareas de los niveles 0 y 1 nunca son suspendidas, mientras que las correspondientes al nivel 2 (nivel de usuari) utilizan el algoritmo round robin.

2.5. Implementación de Procesos en MINIX

2.5.1. Organización MINIX

Minix se organiza ne los siguientes directorios:

- h: Archivos de encabezamiento
- Kernerl: Capas 1 y 2 (procesos, mensajes y drivers)
- mm: Gestión de memoria
- fss: Sistema de archivso
- lib: Funciones de libreria - llamadas al sistema
- tools: Programas necesairos para construir minix
- commands: Programas de utilidad
- include: Archivso de encabezamiento utilizados por los command
- test: Programas de comprobacion
- doc: Documetnacion y manuales

MINIX es un conjunto de 4 programas totalmente independientes (init, kernel, mm y fs) que se comunicacn a través de mensajes

2.5.2. RESUMEN

Funciones capa 1 minix:

- Iniciación del sistema
- GEstión de interrupciones
- GEstión de mensajes (envío y recepcion)
- Planificación

Mirate los códigos que hay por aqui

Para esconder los efectos de las interrupciones, los sistemas operativos proveen un modelo conceptual que consiste en procesos secuenciales corriendo en paralelo.

ASI

Primera Parte: S.O.

3º Ingeniería de Telecomunicaciones — UPV/EHU
Actualizado por última vez el 20 de junio de 2017
"*Under-promise and over-deliver.*"

Javier de Martín – 2016/17

3. Gestión de E/S

3.1. HW E/S

3.1.1. Dispositivos E/S

Los dispositivos E/S pueden ser:

- **Dispositivos de bloque**: Almacenan información en bloques de tamaño fijo, cada uno con su propia dirección. Permiten leer o escribrir cada bloque con independencia de los demás. Son los discos.
- **Dispositivos de carácter**: Liberan o aceptan un conjunto de caracteres sin contemplar ninguna estructura de bloque. Por ejemplo terminales o impresoras.
- **Relojes**: No responden a estructura de bloques direccionables, no aceptan o generan conjuntos de caracteres y causan interrupciones en intervalos definidos de tiempo.

La E/S tiene dos puntos de vista:

- Funcional - Parte independiente - Sistema Archivos
- Estructural - Parte dependiente - Drivers del dispositivo

3.1.2. Controladores de Dispositivo

Los dispositivos de E/S:

- Componente mecánico - Dispositivo propiamente
- Componente electrónico - Drivers de dispositivo

El **controlador** es el responsable del control de los dispositivos exteriores y del intercambio de datos entre los dispositivos y la memoria principal y/o registros de la CPU.

Dibujicos que no se sin utiles

3.2. SW E/S

Estructura en capas:

- Capas inferiores: Esconden peculiaridades HW a las superiores
- Capas superiores: Presentan interfaz simple al usuario

3.2.1. Objetivos

- Software independiente de dispositivo
  - Tiene que existir la posibilidad de escribir programas que puedan ser utilizados con archivos que se encuentren en cualquier dispositivo, sin tener que modificarlos para acondicionarlos a las características de los distintos dispositivos.
- Nombramiento Uniforme (independiente del dispositivo)
  - Todos los archivos y dispositivos se direccionan de la misma forma, a partir del path name (sendero), sin que dependa este direccionamiento del tipo de dispositivo.
- Gestión de errores:
  - En general, los errores deben de ser corregidos tan cerca como sea posible, del lugar en que se producen
- Conversión de las transferencias asíncronas en síncronas
  - Hacer que aquellas operaciones que son activadas por interrupcioens aparezcan como bloqueadas a los programas usuario.
- Clasificación de dispositivos
  - Gestionar al mismo tiempo dispositivos "dedicados" y "compartidos".

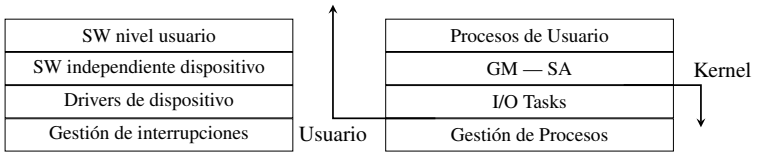


Figura 12: Capas del SW de E/S

3.2.2. Gestión de Interrupciones

Trata de ocultar las interrupciones. Siempre que se produzca un comando E/S y se espere una interrupción tendremos un proceso bloqueado. Cuando se produzca la interrupción se desbloquea el proceso previamente bloqueado:

1. UP semáforo
2. SIGNAL variable condición - monitor
3. SEND mensaje al proceso bloqueado

3.2.3. Drivers del Dispositivo

El driver es un programa dependiente de dispositivo, por eso cada tipo de dispositivo tiene un driver. El driver conoce los registros del controlador y los detalles del dispositivo. Su tarea consiste en aceptar órdenes del SW de la capa superior (independiente), traducirlas a términos concretos (establecer las operaciones que se tienen querealizar y en qué orden) y hacer que se realicen.

Una vez que haya emitido su comando o comandos se autobloqueará hasta que se produzca la correspondiente interrupción, momento en que se desbloqueará.

una vez desbloqueado, comprobará si se han producido errores y pasará los datos al SW independiente.

3.2.4. SW Independiente de dispositivo

Realiza las tareas E/S comunes a todos los dispositivos:

1. Provee una interfaz uniforma a nivel de usuario
2. Cuando se nombra un dispositivo se encarga de asignarle el driver correspondiente
3. Previene a los dispositivos de accesos no autorizados
4. Permite gestionar dispositivos abstractos que emplean un mismo tamaño de bloque, independientemente del tamaño físico real del sector.
5. Gesitona los problemas derivados del almacenamiento, tanto en los dispositivos de bloque como en los de carácter.
6. Asigna el espacio para nuevos archivos que se van a crear.
7. Realiza el tratamiento apropiado según la naturaleza del dispositivo, dedicado o compartido.
8. Gestiona los errores qu eno se han podido corregir en las capas inferiores.

3.2.5. SW Nivel de Usuario

Aunque la mayor parte del SW E/S se encuentra dentro del S.O., una pequeña parte corre fuera de éste.

1. Las llamadas a sistema (incluyendo las relacionadas con E/S), que se implementan a través de subrutinas de librería.
2. Algunas funciones que se implementan trambién a través de subrutinas de librería.
3. Algunos sistemas (procesos) se realizan desde nivel usuario.

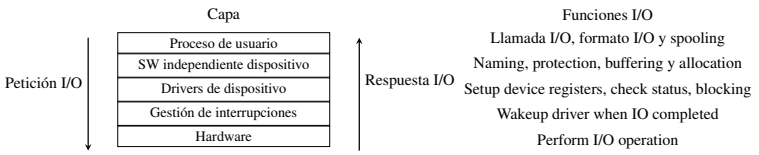


Figura 13: Resumen

3.3. Tratamiento E/S en MINIX

3.3.1. Gestión de Interrupciones

Cuando un driver de dispositivos comienza una operación de E/S, se bloquea en espera de un mensaje. Este mensaje es generado en la parte del S.O. que se encarga de la gestión de interrupciones (capa 1).

3.3.2. Drivers de Dispositivo

Hay un driver por cada tarea. Pertenecen al kernel, lo que permite un fácil acceso a la tabla de procesos y otras estructuras importantes.

Formas de estructurar la comunicación entre el usuario y el sistema:

- Process-structured system: 1-4 son mensajes de petición y respuesta entre 3 procesos independientes

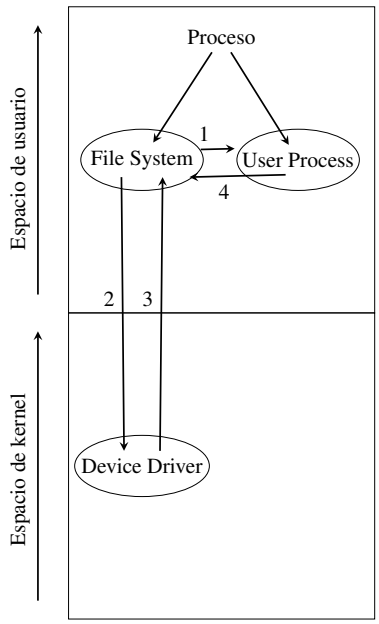


Figura 14: Resumen

- Sistema monolítico: La parte de espacio de usuario llama a la parte de kernel por trapping. El sistema de archivos llama al driver de dispositivo como un procedimiento. El S.O. entero corre como un único programa en modo kernel.

Estructura general de los drivers de dispositivo:

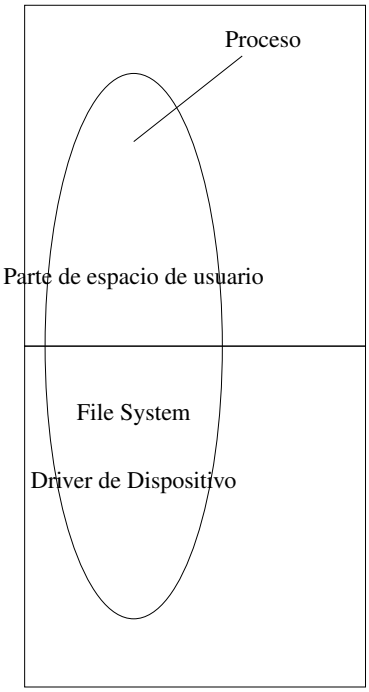


Figura 15: Sistema monolítico

- Capturar un mensaje
- Ejecutar lo que indica el mensaje
- Devolver un mensaje de respuesta

3.3.3. SW independiente de dispositivo

Código E/S independiente del dispositivo → Sistema de archivo (tema 5).

3.3.4. SW Nivel de Usuario

MINIX dispone de funciones de librería para realizar llamadas al sistema y para convertir: binario a ASCII y al revés.

3.4. Memoria RAM

3.4.1. HW Y SW

El disco RAM:

- Dispositivo de bloque más simple de todos, parte de la MP
- Tiene dos comandos: LEER bloque y ESCRIBIR BLOQUE

COsicas aqui

3.4.2. Driver Memoria RAM en MINIX

Número mayor de dispositivo es la memoria RAM. Número menor de dispositivo 0...3

1. `/dev/ram`
2. `/dev/mem` Superusuario
3. `/dev/kmem` Superusuario
4. `/dev/null` Tamaño cero

3.5. Discos

3.5.1. Hardware

Los datos son registrados y recuperados del disco a través de una bobina conductora (cabeza). Durante una operación de lectura/escritura, la cabeza permanece fija, girando el disco debajo de ella.

3.5.2. Software

Tiempo lectura/escritura de un bloque:

1. Tiempo de búsqueda (*seek time*)
2. Tiempo de rotación (*latency time*)
3. Tiempo de transferencia (*latency time*)

La planificación del disco:

- **Multiprogramación**: Procesos generan solicitudes de acceso a disco (leer, escribir) más deprisa de lo que pueden ser atendidas (movimiento de la cabeza de disco).
- **Planificación**: Examen detallado de las solicitudes pendientes para determinar el modo más eficiente de atenderlas.

Planificación:

- Seek optimization
- Rotational optimization

Características:

1. Throughput: Maximizar el número de solicitudes atendidas por unidad de tiempo
2. Minimizar el tiempo de respuesta medio
3. Minimizar la varianza del tiempo de respuesta

Optimización del tiempo de búsqueda:

- FCFS (*First Come First Served*): No hay reordenamiento de la cola
- SSTF (*Shortest Seek Time First*): Sirve la solicitud con menor tiempo de búsqueda, aunque no sea la primera de la cola.
- SCAN: Opera como SSTF excepto que existe la solicitud de menor distancia de búsqueda en una dirección definida.
- N-Step Scan: El brazo de disco se mueve como el caso SCAN, excepto que únicamente da servicio a las solicitudes en estado de espera, cuando comienza un barrido.
- C-SCAN (*Circular scan*): Elimina la discriminación de las estrategias anteriores en relación con los cilindros internos y externos.

Tratamiento de **errores** comunes que se producen en los discos:

1. Error de programación: Petición de un sector inexistente
2. Error de checksum pasajero
3. Error de checksum permanente
4. Error de búsqueda
5. Error de controlador

Cache track-at-a-time. Leer 1 sector es leer 1 track

3.6. Relojes

Los **relojes** son esenciales para la operación de cualquier sistema multiprogramado. Mantienen el tiempo del día y preveen a un proceso monopolizar el uso de la CPU

3.6.1. Hardware del Reloj

Dispositivo que genera interrupciones en intervalos definidos de tiempo.

3.6.2. Software del Reloj

Tareas en las que interviene el reloj:

1. Mantener hora y día.
2. Prevenir que los procesos corran más tiempos del que han sido asignados.
3. Contar el tiempo de uso de la CPU.
4. Manejar la llamada al sistema `ALARM` hecha por procesos de usuario.
5. Proveer temporizadores watchdog para partes del sistema
6. Gestión de estadísticas.

Para simular múltiples relojes se usan los temporizadores watchdog.

3.7. Terminales

3.7.1. HW

Hay dos tipos de terminales en función de la forma en el que el S.O. se comunica con ellos:

- Interfaz RS-232 para comunicación serie
- Terminal Mapeado de Memoria

3.8. System Task

3.8.1. Tarea del sistema en MINIX

Es el SW encargado de comunicar el kernel con MM y FS vía mensajes en aquellos casos en que el kernel necesita estar informado de acciones que se desarrollan en capas superiores.

$$\text{Fork (MM)} \rightarrow \text{Planificación nuevo proceso (kernel)}$$

ASI

Primera Parte: S.O.

3º Ingeniería de Telecomunicaciones — UPV/EHU

Actualizado por última vez el 20 de junio de 2017

"*Under-promise and over-deliver.*"

Javier de Martín – 2016/17

4. Gestión de Memoria

La gestión de memoria

- Seguimiento de zonas de memoria ocupada y desocupada
- Asignación/desasignación de memoria (procesos)
- Intercambio con el disco (cuando no hay capacidad suficiente en la MP)

4.1. G.M. sin Intercambio ni Paginación

Gestión de memoria **sin abstracción**

- Procesos van y vienen entre MP y disco (durante su ejecución, intercambio, paginación...
- Procesos se mantienen en MP

A pesar de no poder correr varios programas de manera simultáneamente, es posible hacerlo. El Sistema Operativo tendrá que guardar el contenido de la memoria a un archivo de disco, y entonces volverlos a traer y correr el siguiente programa. Siempre y cuando haya un único programa a la vez en memoria no habrá conflictos. A esto se le llama *swapping*.
S

No es posible dar acceso a los programas de usuario a todos los registros de memoria.

Para permitir a múltiples programas correr a la vez en memoria hay que resolver los siguientes problemas: protección y relocation.

Se puede resolver con la reubicación de los programas cuando son cargados, es una solución lenta y complicada.

La solución a esto es inventar una nueva abstracción para la memoria: el espacio de direcciones. Un **espacio de direcciones** es un conjunto de direcciones que un proceso puede utilizad para direccionar memoria.

Esta solución es una versión simple de la **reubicación dinámica**, mapea la dirección de cada proceso en una parte diferente de la memoria física de una forma sencilla.

Emplea los registros **base** y **límite**. Cuando un proceso se ejecuta, el registro base se carga con la dirección física donde el programa comienza en memoria ye l registro límite se carga con la longitud del programa.

El uso de estos registros es una forma sencilla de dotar a cada proceso de su propio espacio de direcciones porque cada dirección de memoria generada automáticamente tiene los contenidos del registro base añadidos antes de ser enviados a memoria.

Una desventaja de la reubicación utilizando registros base y límites es la necesidad de sumar y comparar cada referencia a memoria. Las comparaciones se realizan rápido pero las sumas pueden requerir mucho tiempo.

4.1.1. Swapping

Si la memoria física del ordenador es suficientemente grande para albergar todos los procesos no habrá problema, pero este no es el caso.

Hay dos approaches para manejar la sobrecarga de memoria:

- La estrategia más simple es el **swapping** que consiste en traer cada proceso entero, ejecutarlo durante un tiempo y devolverlo al disco. Los procesos en reposo, en su mayoría, se guardan en disco por lo que no consumen recursos cuando no están corriendo.
- La otra estrategia es la **memoria virtual**, permite a los programas ser ejecutados aun cuando sólo están parcialmente en la memoria principal.

Swapping tiene un problema, deja huecos en memoria. Es posible combinarlos en uno único moviéndolos tan abajo como sea posible, esta técnica se conoce como **compactación de memoria**.

SI los procesos son creados con un tamaño fijo que nunca cambia, la reubicación es simple. El sistema operativo reserva tanta memoria como es necesario, ni de más ni de menos.

4.2. Monoprogramación sin intercambio ni paginación

Sólo un proceso a la vez en ejecución, al que se le permite ocupar la totalidad de la memoria.

Modo de funcionamiento:

1. Usuario escribe comando en terminal → S.O. se encarga de cargar el programa del disco y ejecutarlo
2. Cuando el proceso termina, S.O. presenta un prompt en pantalla y espera un nuevo comando del terminal para cargar un nuevo proceso.

4.2.1. Multiprogramación y Empleo de Memoria

Multiprogramación → Mejor utilización de la CPU

4.2.2. Multiprogramación con Particiones Fijas

División de la memoria en `n` partes:

- Colas separadas: Cuando llega un proceso, se pone en la cola de la partición más pequeña donde quepa. El espacio partición no ocupado es espacio perdido. Cola partición grande vacía, colas particiones pequeñas llenas.
- Cola única: cuando queda una partición libre, se encarga el trabajo más cercano al comienzo de la cola que quepa en él. Trabajos pequeños es espacio perdido. La alternativa es coger de la cola el mayor de los trabajos que quepa. Discrimina trabajos pequeños.

Reubicación y protección. La reubicación necesita soporte de HW (data typing y recolocación dinámica).

- Data typing: Se registra el tipo de valor almacenado en cada localización de memoria.
- Recolocación: S.O. utilizando instrucciones privilegiadas examina el tipo de información del código y localiza cada apuntador que se ha de modificar.
  - Recolocación dinámica: Existen 2 registros privilegiados, registro base de recolocación y registro límite. En cada referencia a memoria, se añade automáticamente el contenido del registro base a la dirección efectiva.

La protección necesita soporte HW (candados y llaves y registro límite).

- Candados y llaves: La memoria se divide en bloques de un tamaño determinado y se le asigna un código de protección. Cuando un proceso intenta acceder a una posición de memoria con un código incorrecto se producirá una interrupción de protección.
- Registro límite: Se carga con la máxima dirección del espacio direccionable del proceso y si el preso intenta acceder a una dirección superior a la indicada en el registro límite se produce una interrupción de protección.

4.3. Intercambio (Swapping)

En aquellos sistemas de tiempo compartido en los que hay más usuarios que memoria donde mantener sus procesos, es necesario mantener algunos de los procesos, es necesario mantener algunos de los procesos en el disco e intercambios con los procesos de la memoria principal cada cierto tiempo.

4.3.1. Multiprogramación con Particiones Variables

El intercambio se puede realizar con particiones fijas. En la práctica, cuando la memoria es escasa, se pierde mucho espacio en los casos en que los programas son más pequeños que las particiones.

La alternativa son particiones variables, es un nuevo algoritmo de GM donde el número y tamaño de los procesos en memoria varían dinámicamente.

DIBUJICO

Particiones variables:

- Mayor flexibilidad, mejora el empleo de memoria.
- Complica asignación y desasignación de memoria
- Fragmentación externa
- Complica seguimiento de memoria.

Asignación del tamaño de memoria. Procesos:

- Mantienen mismo tamaño durante su ejecución. Asignación de la memoria que necesitan.
- No mantienen mismo tamaño durante su ejecución. Si se espera que la mayoría de los procesos vayan a crecer cuando se ejecutan, se les asigna una memoria "extra" en lugar de complicar el algoritmo de gestión con un mecanismo que resuelva estas situaciones.

Compactación, solución al problema de la fragmentación. Su objetivo es reunir toda la memoria libre en un único bloque. Supone reubicación de procesos, sólo es posible si se realiza en el tiempo de ejecución (reubicación dinámica).

4.3.2. G.M. con Mapa de Bits

División de la memoria en partes iguales (unidades de asignación). A cada unidad de asignación le corresponde un bit en el mapa.

- bit 0 - unidad libre (desocupada)
- bit 1 - unidad ocupada

El diseño depende del tamaño de las unidades de asignación:

- pequeño: mapa de bits grande
- grande: mapa de bits pequeño, se desperdicia memoria)

El seguimiento del estado de la memoria se realiza de forma sencilla, localizándolo en una zona fija de memoria, determianda por el tamaño de la memoria y el tamaño de la unidad de asignación.

Esto es un problema ya que la búsqueda de un bitmap de una longitud deterinada es una operación muy lenta. Es un argumento suficientemente importante para no usar bitmaps.

4.3.3. G.M. con listas enlazadas

Lista formada pro segmentos de memoria, tanto libres como asignados, todos ellos enlazados. Cada entrada en la lista especifica si es un hueco o un proceso.

Cuando los procesos y los huecos se mantienen en una lista ordenados por direcciones, se pueden utilizar diferentes algoritmos para reservar memoria para un proceso nuevo.

- **First fit**: El gestor de memoria busca en la lista de segmentos hasta que encuentra un hueco suficientemente grande. Este hueco se rompe en dos partes, la parte utilizada por el proceso y el hueco de memoria inutilizado. Es un algoritmo rápido.
- **Next fit**: Variación menor de first fit. Tiene en cuenta donde se hace la última asignación (no vuelve al principio). La siguiente búsqueda la realiza a partir de la última asignación en lugar de empezar por el principio como haría First Fit. Tiene un rendimiento ligeramente menor que First Fit.
- **Best fit**: Mira en toda la lista y coge el hueco más pequeño adecuado. Minimizando así el hueco desaprovechado. Es más lento que ferst fit y desperdicia más memoria que first fit.
- **Worst fit**: Idea opuesta a Best Fit. Asigna el mayor hueco disponible por lo que el nuevo hueco será lo suficientemente grande como para ser útil.
- **Quick Fit**: Mantiene listas separadas para los tamaños más solicitados. Encontrar un hueco de un tamaño determinado es muy rápido pero cuando un proceso termina o es extraído encontrar sus vecinos para ver si se pueden juntar es muy costoso.

Todos estos algoritmos pueden ser acelerados manteniendo listas separadas para procesos y huecos.

4.3.4. G.M. con Sistemas Buddy

Este sistema acelera la fusión de huecos adyacentes (cuando un proceso termina o es intercambiado) utilizando direccionamiento binario.

Forma de trabajo. La G.M. mantiene una lista de bloques libre de 2^k Bytes.

4.4. Memoria Virtual

Mientras que los registros base y límites pueden ser utilizados para crear la abstacción de los espacios de direcciones existe otro problema que hay que resolver, el manejo de bloatware.

El problema de manejar programas que son más grandes que la memoria ha estado presente desde el principio. Una solución adoptada para resolver este programa es dividir el programa en pequeñas piezas llamadas ***overlays*** .

A pesar de que el trabajo de extraer e introducir overlays lo realize el sistema operativo, el trabajo de realizar la división en piezas tiene que ser realizado manualmente por el programa.

Como esto no era una solución óptima se introdujo la **memoria virtual**. La idea básica es que cada programa tiene su propio espacio de direcciones, el cual es dividido en trozos que son **páginas**, siendo cada una un rango contiguo de direcciones. Estas páginas son mapeadas en la memoria física pero no todas las páginas tienen que estar en la memoria física a la vez para correr el programa. Cuando un programa referencia parte de su espacio de direcciones que está en la memoria física, el harware realiza el mapeo necesario al vuelo. Si el programa referencia parte de su espacio de direcciones que no está en dirección física, el sistema operativo es alertado para que vaya a recoger la parte que falta y reejecute la instrucción que falla.

4.4.1. Paginación

Imagen de MMU pag 195

La mayoría de sistemas de memoria virtual utilizan la técnica de **paginación**. El direccionamiento virtual genera unas direcciones llamadas **direcciones virtuales** que forman el **espacio de memoria virtual**. Cuando se utiliza memoria virtual, las direcciones virtuales no van directamente al bus de memoria. En su lugar, van a **MMU** (*Memory Management Unit*) que se encarga de mapear las direcciones virtuales en las direcciones físicas de memoria.

El espacio de direcciones virtuales consiste en unidades de tamaño fijo llamadas páginas. Las unidades que le corresponden en el espacio de memoria física son las **page frames**. Las páginas y pages frames, generalmente, son del mismo tamaño.

Hoy día, un bit de presente/ausente lleva la cuenta de qué páginas están físicamente presentes en memoria.

¿Qué pasa cuando un programa referencia una dirección sin mapear? Provoca que la CPU haga un trap al Sistema Operativo, una **falta de página**. El sistema operativo elige una page frame poco usada y escribe su contenidos de vuelta en el disco (si no está ya ahí). Entonces recupera (también desde el disco) la página que fue referenciada en la página que acaba de ser liberada, cambia el mapa y reinicia la trapped instruction.

El número de página es usado como índice en la **tabla de página**, siendo el número de la page frame correspondiente a la página virtual. Si el bit present/absent está a 0 se causa un trap al sistema operativo. Si el bit está a 1 el número de page frame encontrado en la tabla de página se copia al registro.
Las direcciones empleadas en un programa, son direcciones virtuales (lógicas) y forman el espacio de direccionamiento virtual, que es el que se puede direccionar la GPU.

- Sistemas sin M.V. → Direcciones cargadas directamente en el bus de direcciones (acceso directo memoria física o real)
- Sistemas con M.V. → Direcciones virtuales, MMU (*Memory Management Unit*) es el hardware que transforma la dirección virtual en dirección física.

Cuando se produce una falta de página, el S.O. decide expulsar una de las páginas reales y cargar una página virtual en la dirección que ha quedado libre, y además se encarga de actualizar la table.

Tablas de página

- Dirección virtual:
  - Número de página virtual
  - Offset

El propósito de la tabla de página es convertir páginas virtuales en páginas reales.

Alternativas de diseño:

1. Única tabla de página → Array de registros HW con una entrada por cada página virtual, ordenadas numéricamente. Durante la ejecución se evitan las referencias a la tabla de página. Resulta una solución cara en el caso en el que las tablas sean grandes. Se produce una pérdida de eficacia al tener que cargar diferentes tablas cada vez que se produzca un cambio de contesto.
2. Mantener la tabla de páginas en memoria principal, disponiendo de un registro que apunte al comienzo de la tabla. Un cambio de contexto supondrá únicamente cambiar el contenido del registro. En cambio, la ejecución de una instrucción requerirá una om ás referencias de memoria.

Las tablas de página multinivel evita mantener todas las tablas de página en memoria continuamente.

Si la página no está en memoria, se producirá una falta de página. Si la página está en memoria el número tomado de la tabla de página del segundo nivel se combinará con el offset para construir la dirección física. Una vez obtenida esta dirección se colocará en el bus y se enviará a la memoria.

4.4.2. Tablas de Página

En una implementación simple, el mapeo de las direcciones virtuales a direcciones físicas se puede resumir como:

- La dirección virtual se divide en:
  - Número de página virtual (bits más significativos), se utiliza como índice en la tabla de página para encontrar la entrada para esa página virtual.
  - Offset (bits menos significativos). Es el mismo para la dirección física y la direción virtual.

Así, la finalidad de la tabla de páginas es mapear páginas virutales en page frames.

Estructura de una entrada en la tabla de páginas.

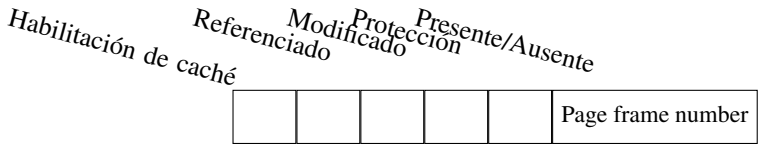


Figura 16: Entrada típica de tabla de página

El campo más importante es el Page Frame Number, después de todo, la principal meta del mapeo de páginas es sacar este valor. El bit de presente/ausente si está a uno indica que la entrada es válida y que puede ser utilizada, si es 0 la página virtual a la que la entrada pertenece indica que no está en estos momentos en memoria. Acceder a una entrada de page table con bit a 0 causa una falta de página. Los bits de protección indican que tipos de acceso son permitidos, si es 0 se puede leer/escribir y si es 1 solo lectura. El bit de Modificado y Referenciado lleva registro del uso de la página, cuando se escribe una página se pone el bit Modificado a 1. Si el SO reclama esta página y este bit ha sido modificado (está sucio) tiene que ser vuelta a escribir al disco y si no ha sido modificada (está limpia) puede ser abandonada ya que la copia del disco es válida. El bit referenciado se pone a 1 cuando una página es referenciada, tanto como por lectura o escritura. Las páginas que no han sido usadas son mejores candidatas que las páginas que han dido usadas en el reemplazo de páginas. Por último el último bit permite deshabilitar el cacheo de la página.

4.4.3. Acelerando la Paginación

En cualquier sistema de paginación hay que resolver dos principales problemas:

1. El mapeo de dirección virtuala dirección física tiene que ser rápido.
2. Si el espacio de direcciones virtuales es grande, la tabla de páginas será grande.

El primer punto es una consecuencia del hecho de que el mapeo de dirección virtual a física tiene que ser hecho en cada referencia a memoria.

Una solución son los **Translation Lookaside Buffers**, está basada en la observación de que la mayoría de los programas tienden a hacer muchas referencias a un pequeño número de páginas y no de la otra forma. De esa forma, una pequeña fracción de las entradas de las tablas de página son leídas; siendo el resto no leídas prácticamente.

Se proponen las TLB es equipar a los ordenadores con un pequeño dispositivo hardware para mapear las direcciones virtuale s adirecciones físicas sin tener que recorrer la tabla de página.

Cada entrada contiene información sobre una página, incluyendo el número de página virtual , un bit que se pone a 1 cuando la página es modificada, el código de protección y la page frame física en la cual la página está ubicada. Estos campos tienen una correspondencia uno a uno con los campos en la tabla de página exceptuando el número de página virtual que no es necesario en la tabla de páginas.

¿Cómo funciona la TLB? Cuando se le pasa una dirección virtual a la MMU para traducirla, primero el hardware comprueba si el número de página virtual está presente en la TLB comparándolo con todas las entradas simultáneamente. Hacer esto requiere hardware especial, el cual todas las MMUs con TLBs tienen. Si se encuentra un match y el acceso no viola el bit de protección, el page frame se toma directamente de la TLB sin tener que ir a la table page. Si el número de página virtual está presente en la TLB pero la instrucción trata de escribir en un apágina de solo escritura se genera una falta de protección.

Hay un caso interesante cuando el número de página virtual no está en la TLB. La MMU detecta la pérdida y realiza una búsqueda en la tabla de páginas ordinaria. Entonces extrae una de las entradas de la TLB y la reemplaza con la tabla de página que acaba de buscar.

Los traps al Sistema Operativo sólo ocurren cuando una página no está en memoria.

Cuando se utiliza software para manejar la TLB hay que entender la diferencia entre diferentes tipos de misses (?????). Un **fallo leve** se produce cuando la página referenciada no está en la TLB pero está en memoria. Para solucionarlo hay que actualizar la TLB. Un **fallo grave** ocurre cuando la página no está ni en la TLB ni en memoria. Se soluciona accediendo al disco para traer la página.

4.5. Segmentación

Compilación en una memoria uni

FAAAALTAN

4.6. Algoritmos de Sustitución de Página

Cuando hay una falta de página, el sistema operativo tiene que elegir una página a expulsar (eliminar de memoria) para hacer hueco a la página que va a llegar. Si la página que va a ser eliminada ha sido modificada mientras estaba en memoria, tiene que voler a ser reescrita en el disco para que se actualice. Si no ha sido actualizada la copia que hay en el disco ya está actualizada por lo que no se necesita volver a reescribirlo. La página a ser leída sobreescribirá la página que va a ser expulsada.

Mientras que podría ser posible expulsar aleatoriamente cualquier página, no es óptimo.
La página a sustituir puede elegirse aleatoriamente corriéndose el riesgo de que la página expulsada sea empleada con mucha frecuencia, lo que supondría una pérdida de rendimiento debido al trabajo extra de gestión que hay que realizar.

4.6.1. Sustitución Óptima

Si cada página que está en memoria tuviera une etiqueta donde se indicara el número de insturcciones que han de pasar antes de que fuera referenciada alguna de las intrucciones contenidas en la página, el algoritmo sustituiría aquella página con el valor más alto en su etiqueta.

4.6.2. Sustitución NRU

Not Recently Used, emplea bits asociados a cada página en memoria virtual:

- `R` (bit referenciado): Se activa por HW, cada vez que la página se referencia (lectura o escritura).
- `M` (bit modificado): Se activa por HW cada vez que la página se escribe

Este par de bits están presentes en cualquier entrada de la tabla de páginas. Deben de ser actualizados cada vez que se hace una referencia en memoria por lo que serán modificados por hardware. Una vez un bit se pone a 1, se queda con ese valor hasta que el sistema operativo lo resetea.

Tan pronto como cualquier página es referenciada ocurrirá una falta de página. El sistema operativo pondrá entonces el bit `R` a uno, cambia la entrada de la tabla de página para apuntar a la página adecuada con modo READ ONLY y reinicia la instrucción.
Algoritmo de sustitución a partir de los bits R y M:

1. Cuando empieza un proceso $\rightarrow$ S.O. pone los bits R y M de todas las páginas a 0.
2. Periódicamente (cada interrupción de reloj) s epone a 0 el bit R (para distinguir aquellas páginas que han sido referenciadas receintemente de las qu eno lo han sido)
3. Cuando se produce una falta de página $\rightarrow$ S.O. inspecciona todas las páginas y las clasifica en 4 categorías en función de sus valores de los bits R y M
   - Clase 0 No Ref no mod
   - Clase 1 no ref, mod
   - Clase 2 ref, no mod
   - Clase 3 ref, mod

Algoritmo NRU, cambia aleatoriamente una página de la clase numerada más baja no vacía.

4.6.3. Sustitución FIFO

El S.O. mantiene una lista de todas las páginas en memoria, ordenadas por el tiempo que llevan en memoria, con la que más tiempo lleva en cabeza y la que menos en cola. Tiene un problema, cuando la página más antigua que es expulsada es muy utilizada.

Haciendo una variación de este algitmo nace **segunda oportunidad**, resuelve el problema de tener que expulsar una página que se usa mucho. todo ello se soluciona inspeccionando el bit `R` de la página más antigua. Si es `0` la página es vieja y no ha sido utilizada por lo que puede ser expulsada. Si el bit `R` es 1, el bit se pone a 0, se coloca la página al final de la lista de páginas y la búsqueda continúa.

Si todas las páginas han sido referenciadas este algoritmo degenera en simplemente FIFO.

4.6.4. Sustitución Clock Page

A pesar de que segunda oportunidad es un algoritmo útil es ineficiente ya que está continuamente desplazando páginas por la lista. Un mejor planteamiento es mantener todas las páginas en una lista circular como si fuera un reloj en la que la mano del reloj apunta a la página más antigua.

Cuando ocurre una falta de página, la página que está siendo apuntada por la manecilla se inspecciona. Si el bit `R` está puesto a `0` la página se expulsa y se inserta una nueva en su lugar avanzando la manecilla. En cambio, si el bit `R` está puesto a `1`, se limpia y la manecilla avanza a la siguiente página. Este proceso se repite hasta que se encuentre una página con `R = 0`.

4.6.5. Sustitución LRU

Least Recently Used:

- Localismo → Páginas que han sido muy utilizadas en las últimas referencias, probablemenet lo serán en las siguientes.
- Algoritmo → Cuando ocurra una falta de página se expulsa la página que no haya sido utilizada por más tiempo o la menos recientemente utilizada
- Implementación → Mantener una lista encadenada de todas las páginas de mmeoria, con la más recientemente utilizada al frente y la menos recientemente utiliada en la cola.

La actualización de la lista en cada referencia en memoria (encontrar página en la lista, suprimirla y ponerla la frente) es un proceso muy lento.

4.7. Principios de Diseño de Sistemas de Paginación

4.7.1. Localismo

Propiedad empírica más que teórica. Los procesos realizan referencias de forma no uniforme, referencias según patrones altamente localizados.

- Sistemas de paginación
  - Referencia subconjunto de páginas
  - Las páginas referenciadas tienden a ser adyacentes en el espacio de direccionamiento virtual
- Localismo
  - Temporal → Referencias recientes tienen muchas posibilidades de ser referenciadas en un futuro cercano.
  - Espacial → Una vez que se realiza una referencia a una localización, es muy probable que sus vecinos sean referenciados.

La consecuencia → ejecución de un programa verá mejorada su eficacia en la medida que el conjunto de páginas referenciadas se encuentre en la memoria principal.

4.7.2. Working Set

Los procesos son iniciados con ninguna de sus páginas en memoria. Tan pronto como la CPU intente recoger la primera isntrucción se producirá una falta de página provocando al sistema operativo que traiga la página que contiene la primera instrucción. Después de esto se producirán otras faltas de páginas para variables globales ausentes.

Después de un rato, elp roceso tendrá la mayoría de páginas que necesita y comenzará a ejecutarse con relativamente pocas faltas de páginas.

A esta estrategia se la conoce como **paginado bajo demanda** ya que las páginas se cargan únicamente bajo demanda y nunca por adelantado.

Gestión de memoria con working set

- Mantener en memoria principal los working set de los programas activos. La decisión de añadir un nuevo proceso al conjunto de procesos activos (incrementar el grado de multiprogramación) se tomará cuando haya espacio suficiente para acomodar el working set del nuevo proceso.
- W, ventana del working set. El working set de un proceso en el instante "t" es el número de página referenciadas por el proceso en el intervalo de tiempo.
- Determinación del tamaño de ventana → Operación crítica para la eficacia en el funcionamiento de la gestión de memoria de los working set.
- El working set varía con la ejecución del proceso.

Una vez que el proceso se estabiliza, el sistema compara las referencias de página en la ventana con las observadas en la ventana anterior y aumenta o disminuye el número de página en el working set.

- La implementación de una auténtica política de gestión de memoria con la working set puede suponer una sobrecarga en la información de gestión, debido principalmente a que la composición de los working set pueden variar rápidamente.

4.7.3. Algoritmo PFF

Page Fault Frequency es una medida de la eficacia en la ejecución de un proceso es la relación de faltas de página.

- Procesos con muchas faltas → No mantienen el working set en MP → Thrasing
- Procesos con pocas faltas → demasiadas páginas → Impiden progresar a otros procesos

El algoritmo PFF ajusta el conjunto de páginas residentes de un proceso en función de la frecuencia de aparición de faltas (o en función del tiempo entre faltas.

Realiza un ajuste dinámico del conjunto residente de página en respuesta al comportamiento variable del proceso.

El elemento clave para un funcionamiento eficiente de este algoritmo es el de establecer los umbrales con valores apropiados.

Ventaja PFF sobre los Working Set → El ajuste conjunto residente de página se realiza después de cada falta, mientras que el working set se realiza después de cada referencia de almacenamiento.

4.7.4. Demanda de página

Ninguna página es trasladada desde el almacenamiento secundario hasta que sea explicitamente referenciada por el proceso en ejecución.

Ventajas:

- Garantía de que las páginas en mp son las que el proceso necesita en ese momento.
- La carga de información de gestión para la determinación de la página que se va a llevar a memoria es mínima (contra técnicas de anticipación).

Problemas:

- La carga de página se realiza página a página → concepto de coste de espera.

4.7.5. Prepaginación

Paginaión anticipada. El SO intenta predecir las páginas que necesitará el proceso y las carga cuando hay espacio disponible.

Mientras el proceso se está ejecutando con sus páginas actuales, el sistema carga páginas nuevas para que estén disponibles cuando el proceso las necesite.

Las páginas referenciasdas se solicita por demanda y las páginas vecinas son prepaginadas.

4.8. Tratamiento GM en MINIX

TERMINA

ASI

Primera Parte: S.O.

3º Ingeniería de Telecomunicaciones — UPV/EHU
Actualizado por última vez el 20 de junio de 2017
"*Under-promise and over-deliver.*"

Javier de Martín – 2016/17

5. Gestión de Archivos

5.1. Archivos

5.2. Nombramiento

Un archivo es un mecanismo de abstracción, cada archivo está identificado por su nombre.

5.3. Estructura

Hay 3 posibilidades:

1. Secuencia de bytes (sin estructura): El S.O. no sabe ni le importa lo que hay en el fichero.
2. Secuencia de registros de longitud fija (principio de estructura): Array de arrays.
3. Árbol de registros: Array de estructuras ordenadas según un campo de la estructura (utilizada en grandes ordenadores)

5.3.1. Tipos

1. Convencionales
2. Directorios
3. Especiales
   - Carácter
   - Bloque

5.3.2. Acceso

1. Secuencial
2. Aleatorio

5.3.3. Atributos

5.3.4. Operaciones

5.3.5. Mapeado de Archivos en Memoria

Otra forma de acceder a los archivos es volcar (mapear) los archivos dentro del espacio de dirección de un proceso (MULTICS).

El mapeo, dad una dirección virtual y el nombre de un archivo hace que el S.O. mapee el archivo dentro de lespacio de direccionamiento de la dirección virtual.

el mapeo de memoria trabaja mejor en los sistemas que soportan segmentación. De esta forma, cada arvhico puede ser mapeado en su propio segmento.

Problemas del mapeado de archivos en memoria:

1. Resulta difícil al sistema conocer la longitud exacta del archivo de destino (en el caso de que se tratarad e una segmentación paginada, podríamos conmocer el número más alto de la página que ha sido escrita pero no cuantos ytes han sido escritos en ella).
2. (Potencialmente) Puede ocurrir que si un archivo es mapeado por un proceso y también es abierto por una operación convencional de lectura, si el proceso modifica una página, ese cambio no se reflejará en el archivo hasta que la página sea expulsada (inconsistencia).
3. El archivo puede ser mayor que el segmento.

5.4. Directorios

5.4.1. Sistemas Jerárquicos de Directorios

Un directorio, típicamente contiene un número de entradas, una por cada archivo. Hay dos modalidades

- Atributos en la entrada del directorio
- Atributos en otro lugar

Abrir un archivo:

1. S.O. busca en el directorio hasta que encuentra el nombre del archivo.
2. Extrae los atributos y las direcciones del disco y las coloca en una tabla de la memoria principal.
3. Todas las referencias posteriores al archivo utilizan la información de la memoria principal.

5.4.2. Paths

Cuando el sistema de archivos está organizado como un árbol de directorios se necesita algunan forma de especificar los nombres del archivo:

1. Path absoluto (desde el directorio raíz)
2. Path relativo (directorio de trabajo actual)

5.4.3. Operaciones

Las operaciones (llamadas al sistema) para la

5.5. Implentación de un Sistema de Archivos

La principal tarea al implementar un sistema de almacenamiento es mantener una lista de qué bloques de disco van con qué archivo.

5.5.1. Asignación Continua

El esquema de asignación más simple, almacenar cada archivo seguido uno detrás de otro. La ventaja de este esquema es que es simple de implementar ya que se reduce la complicación de saber dónde está cada archivo, solo se necesita la dirección de disco del primer bloque y el número de archivos en cada bloque. La tasa de lectura es muy buena ya que el archivo entero puede ser leído del disco en una única operación. Sólo se necesita la operación *seek* del primer bloque. Después de sto no se necesitan más *seeks* o retrasos rotacionales. Tiene una desventaja importante, con el paso del tiempo el disco se fragmentará. Cuando un archivo se elimina, sus bloques se liberan, dejando unos bloques libres en el disco. El disco no se compacta hasta el punto de perder el hueco ya que habría que copiar todo de nuevo. como resultado, el disco consistirá en huecos y datos.

5.5.2. Listas Enlazadas

Hay que mantener cada archivo como una lista enlazada de bloques de disco. La primera palabra de cada bloque se usa como puntero al siguiente. El resto del bloque son datos.

Al contrario que la asignación contigua, todos los bloques del disco pueden ser utilizados según este método. No se pierde espacio disco por la fragmentación[6]. Es suficiente para la entrada del directorio solo almacenar la dirección de disco del primer bloque, el resto se pueden encontrar a partir de ahí.

A pesar de que leer un archivo de forma secuencial es sencillo, el acceso aleatorio es extremadamente lento. Para llegar al bloque n, el sistema operativo tiene que empezar desde el principio y leer $n-1$ bloques antes, haciéndolo muy lento.

[6] Excepto por la fragmentación interna en el último bloque

El tamaño de datos a almacenar en un bloque ya no es una potencia de dos ya que el puntero usa unos. Esto no es un problema grave pero añade ciertas complicaciones ya que los programas leen y escfriben en potencias de base dos.

¿Cómo eliminar el bloque n? Hay que acceder al bloque $n+1$, pasando antes por la lectura de n bloques. Escribir la dirección del bloque $n+1$ en el bloque $n-1$ para saltar su lectura. Además, hay que marcar el bloque n como libre en el mapa de bits (o en la lista) de bloques libres.

5.5.3. I-nodos

Asocia a cada archivo una estructura de datos llamada **i-nodo** (*i-node*), que almacena los atributos y direcciones de disco de los bloques del archivo. Dado un i-nodo, es posible, encontrar todos los bloques de un archivo.

La principal ventaja de este esquema sobre las listas enlazadas es que los i-nodos sólo tienen que estar en memoria cuando el correspondiente archivo está abierto.

Si cada i-nodo ocupa n bytes y hay un número máximo de k archivos que pueden ser abiertos a la vez, la memoria total ocupada por el array que alberga los i–nodos para los archivos abiertos ocupa $k \cdot n$ bytes. Y sólo este tamaño tendrá que ser reservado por adelantado.

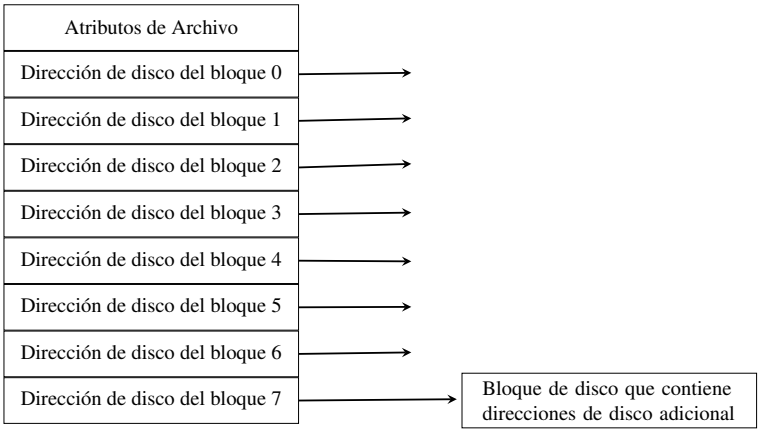


Figura 17: Ejemplo de un i-nodo

Este array es bastante menor que el espacio ocupado por la tabla de archivo. La razón es simple. La tabla para guardar la lista enlazada de todos los bloques de disco es proporcional en tamaño al disco en sí. Si el disco tiene n bloques, la tabla necesita n entradas.

Un problema con los i-nodos es que cada uno tiene espacio para un número fijo de direcciones de disco. ¿Qué pasa si un archivo crece por encima de este límite? Una solución es reservar la última dirección de disco no para un bloque de datos, sino para la dirección de un bloque que contenga más direcciones de bloque de disco.

5.5.4. Implementación de directorios

Antes de que un archivo pueda ser leído, tiene que ser abierto. Cuando se abre un archivo, el Sistema Operativo usa el path name provisto por el usuario para localizar la entrada en el disco. La entrada del directorio provee información necesaria para localizar los bloques de disco. Dependiendo del sistema, esta información puede ser la dirección de disco del archivo completo (con asignación contigua), el número del primer bloque (esquemas de listas enlazadas) o el número del i-nodo.

En cualquiera de los casos, la principal función del sistema de directorios es mapear el nombre ASCII del archivo en la información necesaria para encontrar los datos.

La tarea principal de un sistema de directorios es convertir el nombre en ASCII de un archivo en la inforamción necesaria par alocalizar sus datos.

5.5.5. Archivos compartidos

La conexión entre un directorio y un archivo compartido se llama *link*.

Esto introduce algunos problemas. Si los directorios contienen una copiad e las direcciones de disco, si alguno de los archivos se modifica, esta modificación no se verá reflejada en el otro archivo. Esto se soluciona con i-nodos o enlaces simbólicos[7].

Inconvenientes de los i-nodos, la realización de un link nocambia el propietario, únicamente cambia el valor del campo contador de enlaces. Si un usuario trata de biorrar el archivo hay un problema.

Inconvenientes de los enlaces simbólicos, si un usuario C elimina el archivo, cuando B quiera acceder a él el sistema le informará de que el fichero no existe.

5.5.6. Gestión del Espacio de Disco

Estrategias de almacenamiento:

- Asignación consecutiva
- División del fichero en bloques

Si el tamaño de lbloque es grande se tiene poca eficacia en la utilización del disco, pero si espequeño habrá muchos bloques siendo el acceso muy lento.

Para el seguimiento de bloques libres hay dos métodos: lista enlazada y mapa de bits.

Con las **cuotas de disco** el administrador del sistema asigna a cada usuario un número máximo de archivos y de espacio, y el sistema operativo se asgura de que el usuario no sobrepasa esas cuotas.

5.5.7. Fiabilidad del S.A.

Hay varias tares para mejorar la seguridad del sistema de archivos:

- **Gestión de bloques en mal estado**:
  - El hardware dedica un sector del disco par aguardar la listad e bloques malos. Cuando el controlador se inicializa, lee la lista de bloquea malos y coge un bloque de repuesto para sustituir los defectuosos, registrando el mapeado en la lista de bloques malos. En lo sucesivo, todas las referencias a los bloques malos, utilizarán los de repuesto.
  - El software, el usuario construye un archivo que contenga los bloques malos y quita estos de la lista de bloques libres.
- **Backups**
- **Consistencia**: Correspondencia entre la información del S.A. (dirección donde se encuentra la información) y l ainformación propiamente dicha. Programas que analizan la consistencia: nivel bloque y nivel archivo. ¿Añadir algo más?

5.5.8. Eficacia

El acceso a disco es muchos órdenes más lentoq ue el acceso a memoria. Para reducir los accesos a disco se utiliza la caché[8]

Algoritmos de gestión de caché. El más sencillo consisste en comprobar todas las solicitudes de lectura para ver si el bloque que se necesita está en la caché. Si está, la solicitud puede ser atentidad sin acceso al disco. Si el bloque no está en la caché primero se lee dentro de la caché y después es copiadoa. donde sea necesario.

[7] Se crea un nuevo fichero de tipo `LINK` en el directorio que quiere compartir el archivo, que contenga únicamente el path del archivo al que es enlazado

[8] Colección de bloques que pertenecesn lógicamente al disco, pero que se encuentran en memoria, por razones de eficacia

Cuando la caché está llena y hay que expulsar algún bloque para dar cabida a otro, cualquiera de los algoritmos de sustitución vistos en paginación es válido (FIFO, segunda oportunidad o LRU).

Las referencias a caché son relativamente infrecuentes y por tanto es viable mantener los bloques en exacto orden LRU (con una lista enlazada).

Problema de consistencia del sistema de archivos cuando els sistema cae. Si un bloque crítico, como el i-nodo, se lee dentro de caché y se modifica, pero no se actualiza en disco, si el sistema cae, dejará el SA en un estado inconsciente.

5.6. Tratamiento Sistemas de Archivos en MINIX

5.6.1. Mensajes

El sistema de archivos acepta 29 tipos de mensajes (todos excepto 2, llamadas al sistema).

5.6.2. Distribución S.A.

Distribución típica del Sistema de Archivos:

1. **Bloque Boot**: Ocupa 1 bloque.
2. **Super Bloque**: Información de la distribución del Sistema de Archivos (su tarea principal es dar información del tamaño de las 6 partes que lo forman). Ocupa 1 bloque.
3. **Mapa de Bits de los i-nodos**: Es necesario un 1 bit para referenciar cada archivo.
4. **Mapa de bits de las zonas**:

$$\frac{\text{Tamaño total}}{\text{Tamaño de bloque}} = \text{zonas} \rightarrow \frac{\text{zonas}}{8} = \text{bytes} \rightarrow \frac{\text{Bytes}}{\text{Tamaño Bloque}} = \text{bloques}$$

5. **i-nodos**:

$$\frac{\text{Tamaño de bloque}}{\text{Tamaño de i-nodo}} = (\text{i-nodos/bloque}) \rightarrow \frac{\text{Número de i-nodos}}{\text{i-nodos/bloque}}$$

6. Datos: Bloques libres que queden.

Cuando se arranca MINIX, el superbloque del dispositivo raíz, se lee dentro de una tabla de memoria. De igual forma, cuando se montan otros Sistemas de Archivo, se cargan sus superbloques en memoria.

La tabla de superbloques tiene algunos campos no presentes en el disco, como son: el dispositivo del cual viene, si es sólo lectura o no y un capo que se pone a 1 siempre que la versión de memoria sufre alguna modificación.

Antes de que un disco pueda ser utilizado como Sistema de Archivos, se le debe dar la estructura anterior.

5.6.3. Mapas de Bits

En MINIX, el seguimiento de los i-nodos y de las zonas libres se realiza a través de dos mapas de bits.

Cuando se crea un archivo, el Sistema de Archivos busca a través de los bloques de mapas de bits hasta que encuentra un i-nodo libre. Si todos los i-nodos de unbloque están ocupados, la rutina de búsqueda devuelve un 0, por eso nunca se emplea el i-nodo 0.

Al usar zonas en lugar de bloques se mejora la eficacia cuando se realiza un acceso al fichero (zona es una localización contigua de bloques).

5.6.4. i-nodos

En MINIX un i-nodo ocupa 32 bytes mientras que en UNIX ocupan 64 bytes. Cuandos e abre un archivo se carga su i-nodo en memoria (tabla de i-nodo). La tabla de i-nodo tiene más campos que los que están en disco: Número de i-nodo, dispositivo y contador.

5.6.5. Bloque caché

La implementación de la memoria caché del SA está formada por buffers. Cada uno de ellos posee una cabecera y un bloque de disco. Todos los bloque están formando una lista doblemente encadenada o enlazada, teniendo al inicio de la lista el bloque menos recientemente usado (LRU) y en la última el más recientemente usado.

En caso que se necesite un bloque de caché:

- Está el bloque en caché, se devuelve al cliente y se incrementa el contador de la cabecera de buffer para saber que ha sido usado.
- No está en caché: Se busca un bloque para expulsar:
  - Contador de bloque a 0 $\rightarrow$ Bloque expulsado
    - FLAG = 0 $\rightarrow$ Se expulsó
    - FLAG = 1 $\rightarrow$ Se copia en disco porque ha sido modificado
  - Contador distinto de 0 $\rightarrow$ Se inspecciona el siguiente bloque.
- La función de petición de bloque desperdicia la naturaleza de bloque (i-nodo, directorio, datos...) pone el bloque en primer lugar de la lista de caché o al final. En LRU no se usa y en MRU se usa.
- Dependiendo del tipo:
  - Si no es convencional y se ha modificado se escribe en el disco en el instante de la modificación.
  - Si es convencional y se ha modificado se espera al SYNC o a que sea expulsado para ser escrito en el disco.

5.6.6. Directorios y Paths

El Sistema de Archivos, a partir del path viaja a lo largo del árbol de directorios hasta encontral el i-nodo del fichero. El directorio en MINIX es un archivo con entradas de 16 bytes, 2 para el número de i-nodo y 14 para el nombre del archivo.

Dibujo como do scuadrados con las cosas

5.6.7. Descriptores de Archivo

Cuando se abre un archivo, se devuelve un descriptor de archivo (número asociado a un archivo abierto) al proceso usuario para utilizarlo en posteriores llamadas.

Al igual que en el kernel y en el GM el SA mantiene parte de la tabla de procesos. 3 de sus campos son de especial interés

1. Apuntador directorio raíz (i-nodo)
2. Apuntador directorio de trabajo (i-nodo)
3. Array de apuntadores ordenador poe el número del descriptor del archivo

El tercer campo podría parecer suficiente que el contenido de este array fueran apuntadores a los i-nodos de los respectivos descriptores de archivo.

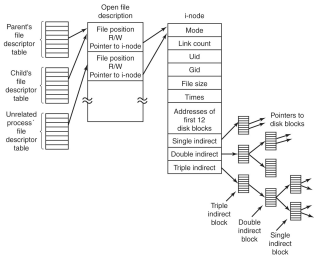
Desafortunadamente, esta solución no funciona por el hecho de que los archivos pueden ser compartidos. Asociado a cada archivo hay un número de 32 bits que indica la posición del siguiente byte para leer o escribir, ¿dónde se guarda este número?

1. En el i-nodo, desfortunadamente 2 o más procesos pueden tener el mismo archivo abierto a la vez y tendrían que tener sus respectivos apuntadores a sus posiciones, y el i-nodo es único.

2. En la tabla de procesos. En un segundo array, paralelo al array de descriptores de archivo, dando la posición de cada archivo.. Est ano es solución por la situación que plantea `FORK`, donde el padre y el hijo comparten un único puntero a la posición de cada fichero abierto.

Como resultado, no es posible poner la posición en la tabla de procesos. La posición debe ser compartida.

La solución es introducir una nueva tabla, **FILP** (File Position) que contiene las posiciones de los archivos.



En cada entrada de la tabla FILp se introduce además el apuntador al i-nodo y un contador que cuenta el número de procesos que lo está utilizando (así el SA puede saber cuando el último proceso que ha utilizado el archivo, ha terminado) para poder liberar la entrada.

Por tanto el array de descriptores de archivo de la tabla de procesos apunta a la entrada FILP.

5.6.8. Pipes y Archivos Especiales

El concepto de completar una operación de lectura o escritura en un archivo especial difiere de los archivos ordinarios. Depende de por ejemplo si el pipe está vacío o no y en el caso de un archivo especial si se introducen datos o no.

En los **pipes** un proceso que intenta leer de un pipe vacío debe de esperar hasta que otro proceso ponga información en el pipe. El S.A. puede verificar el estado del pipe y suspender la solicitud si ve que está vacío para volver a iniciarla después. El S.A. registra los parámetros de la llamada l sistema en la tabla de procesos para poder reiniciarlo después.

En los **archivos especiales** el i-nodo de los archivos especiales tiene dos campos:

1. Número de dispositivo mayor (clase de dispositivo, se utiliza como índice en la tabla S.A. para acceder a la tarea correspondiente)

2. Número de dispositivo mejor (se pasa al driver como parámetro).

Cuando un proceso lee de un archivo especial:

1. El S.A. extrae los números mayor y menos de dispositivo de la tabla de i-nodo.

2. Con el número mayor en la tabla S.A. obtiene el número de tarea.

3. El S.A. envía un mensaje que incluye: Parámetros del disposivito menor, operación, número del proceso que efectúa la llamada y número de bits que se transfieren.