TEMA 1

INTRODUCCIÓN

- ***Diferentes niveles en la arquitectura de un computador***

Un computador digital es una máquina que puede resolver problemas ejecutando ciertas instrucciones.

Un programa es una secuencia de instrucciones que definen una tarea.

Los circuitos electrónicos de cada computadora reconocen un conjunto limitado de instrucciones muy simples. Al conjunto de instrucciones básicas de una computadora se le denomina lenguaje máquina (por ejemplo, sumar dos números). Es el único lenguaje que entiende el computador pero es tan elemental que es difícil y tedioso programar o depurar en él. La solución es la siguiente:

Sea L1 el lenguaje máquina y L2 un lenguaje de más alto nivel, fácil de utilizar. Entonces el programador escribe en L2, y luego el programa se traduce a L1, pues L2 no lo entiende la máquina. Esa traducción puede ser:

- Compilación: El compilador traduce la secuencia de instrucciones de L2 a una secuencia en L1. El resultado se almacena en un programa en L1.
- Interpretación: El intérprete no genera ningún programa en L1, sino que convierte cada instrucción de L2 en una secuencia de instrucciones en L1, las cuáles ejecuta directamente.

Se puede imaginar así la existencia de una máquina virtual cuyo lenguaje máquina sea L2 (tenemos un compilador o intérprete a L1, y nos podemos olvidar de la máquina que trabaja con L1). L2 no debe diferir mucho de L1 para que la traducción sea práctica, pero para que nosotros entendamos L2 sí que debe de haber bastante diferencia. Por tanto, lo que se puede hacer es crear L3, L4, ..., cada uno más fácil de utilizar. A cada nivel le corresponde una máquina virtual (M1, M2, ..., Mn). Cada programador humano sólo necesita conocer un lenguaje (por ejemplo, el L3), y puede olvidarse de los niveles inferiores (L1 y L2 en este caso).

- ***Máquinas multinivel actuales***

La mayoría de las máquinas actuales constan de seis niveles, que son:

Nivel 5

Nivel 4

Nivel 3

Nivel 2

Nivel 1

Nivel 0

Los niveles son una abstracción. Quien trabaja en un nivel no tiene que preocuparse de los inferiores.

Cada nivel es soportado por un programa. Por ejemplo, para el nivel cinco, el compilador. En este curso nos

dedicaremos al nivel tres.

Los microprogramas son directamente ejecutados por el hardware.

Ahora vamos a ver cada nivel más detalladamente.

Nivel cero: nivel de lógica digital.

Es el hardware de la máquina. Habría aún un nivel inferior, el nivel de dispositivo. En este nivel se estudian: las puertas lógicas, los circuitos integrados (SSI, MSI, LSI, VLSI), circuitos combinacionales, circuitos aritméticos, relojes, memorias, microprocesadores, buses, etc.

Nivel uno: nivel de microprogramación.

Aquí existe un programa llamado microprograma, cuya función es interpretar las instrucciones del nivel dos. El microprograma es un intérprete, que pasa cada instrucción de lenguaje máquina a microinstrucciones, las cuáles son ejecutadas.

En algunas máquinas no existe este nivel.

Nivel dos: nivel de máquina convencional.

Cada fabricante publica el Manual de referencia del lenguaje máquina para cada uno de sus computadores (dice las instrucciones de lenguaje máquina que éstos tienen).

Las instrucciones del nivel de máquina las interpreta el microprograma. En las máquinas en las que no existe el nivel de microprogramación, sin embargo, las instrucciones del nivel de máquina son realizadas directamente por los circuitos electrónicos (el hardware, el nivel cero).

Nivel tres: nivel de sistema operativo.

La mayoría de las instrucciones de este nivel se encuentran también en el nivel dos, pero, además, tienen un nuevo conjunto de instrucciones añadidas, así como una organización diferente de la memoria, posibilidad de ejecutar dos o más programas, etc.

Las nuevas instrucciones las interpreta el sistema operativo, mientras que las que son idénticas a las del nivel dos las lleva a cabo el microprograma.

Nivel cuatro: nivel del lenguaje ensamblador.

Los niveles cuatro y superiores son utilizados por los programadores de aplicaciones, los niveles inferiores no están pensados para programar aplicaciones directamente en ellos, sino que están diseñados para ejecutar los intérpretes y traductores de los niveles superiores y son escritos por los programadores de sistemas.

El ensamblador es un lenguaje de nivel tres, que lleva a cabo la traducción de un programa de nivel cuatro al nivel tres.

Nivel cinco: nivel de lenguajes de alto nivel.

Los lenguajes de alto nivel son más fáciles de usar que los niveles inferiores. Son utilizados por los programadores de aplicaciones.

Los traductores de programas en lenguaje de alto nivel pueden ser compiladores o intérpretes.

- ***Evolución histórica de las máquinas multiniveles***

La arquitectura de los computadores ha ido evolucionando a lo largo de la historia, la cual se divide en distintas etapas llamadas generaciones. Vamos a ver dichas generaciones (las fechas difieren de una bibliografía a otra):

Generación cero (1642−1945).

- Tecnología:

Computadores mecánicos o electromecánicos con muchas limitaciones (a partir de engranajes, manivelas, etc.).

- Personas destacadas:

Blaise Pascal construyó en 1642 una máquina calculadora para sumar y restar, con el fin de ayudar a su padre (que era recaudador de impuestos).

Charles Babbage construyó en 1834 una máquina de propósito general que constaba de tres partes: almacén (memoria), taller (CPU) y sección de entrada y salida (unidad de E/S). Contrató a una mujer llamada Ada para programar la máquina, convirtiéndose así en la primera programadora de la historia. Más tarde se daría el nombre de Ada a un lenguaje de programación.

Aiken construyó la MARK I en 1944, inspirado en los estudios de Babbage.

Primera generación (1945−1955).

- Tecnología:

Válvula electrónica de vacío. Las válvulas eran voluminosas, caras y poco fiables, lo que conllevaba computadores grandes e incómodos.

- Modelos:

ENIAC (1946): 18000 válvulas, 30 toneladas, 1400 m2, 100 Kw, 5000 sumas por segundo.

EDSAC (1949): Primer ordenador con programas almacenados.

UNIVAC: Primer ordenador en ser comercializado.

- Personas destacadas:

John Von Neumann establece un modelo de la estructura de un ordenador (memoria, unidad aritmético lógica, unidad de control y unidad de entrada y salida) que es la que sigue aún vigente. Realza la idea de computador con programa almacenado.

- Modo de funcionamiento:

Se programa en lenguaje máquina, propio de cada máquina y muy complicado (se desconocen los lenguajes de programación), no existe sistema operativo, se realiza el programa cableado, se solicita hora para la

máquina, y se inserta el panel de conexiones (donde tenemos el programa cableado) en el computador para ejecutar el programa. A principios de los cincuenta se mejora el procedimiento con las tarjetas perforadas.

Segunda generación (1955−1965).

- Tecnología:

Transistor (inventado por Bardeen y Brattain en 1947). Los transistores sustituyen a las válvulas de vacío ya que presentan las siguientes ventaja: menor espacio, menor consumo, menor precio y mayor fiabilidad. Esto hace disminuir el precio y tamaño de las computadoras.

- Modelos:

PDP−1 de DIGITAL.

- Modo de funcionamiento:

Aparecen los lenguajes de alto nivel: FORTRAN, COBOL, ALGOL, PL/1.

Se escribe el programa en papel, luego se perfora en tarjetas, se lleva al operador y finalmente se recoge el listado de impresora. Sólo las grandes empresas y universidades podían tener un computador, pues aún eran muy caros. Además, hacia falta gente especializada tanto parar perforar las tarjetas como para utilizar la máquina. Había también que hacer comprobaciones por si la perforación de la tarjeta no era correcta. En definitiva, el proceso completo hasta que se obtenía el listado de impresora podía durar varios días.

Aparece el sistema de procesamiento por lotes (con el sistema operativo). Se trata de un método para reducir el número de veces que había que ir a buscar el compilador (por ejemplo, el de FORTRAN). Vemos cómo funciona con un caso particular.

El 7094 es un computador más potente que el 1401 pero más caro, por lo que, para no quitarle tiempo, se usan los dos 1401 menores. Con uno de ellos se pasa de las tarjetas perforadas a cintas. Las cintas se llevan al 7904, que así funciona más rápido que con las tarjetas, la salida se almacena en otra cinta, y con el otro 1401 se saca por impresora.

Es en el 7094 donde se carga el compilador (por ejemplo, el de FORTRAN):

Con $JOB comienza el trabajo. $FORTRAN es una tarjeta de control que le indica al sistema operativo que cargue el compilador de FORTRAN. Vemos, por tanto, que se trata de un sistema operativo muy simple.

Tercera generación (1965−1980).

- Tecnología:

Circuitos integrados SSI (hasta 100 elementos integrados) y MSI (100−3000).

- Modelos:

IBM sistema 360 y PDP−8 (DIGITAL).

- Modo de funcionamiento:

Aparecen más lenguajes de alto nivel: BASIC y PASCAL.

También aparece un sistema operativo con multiprogramación, para no desaprovechar los tiempos de E/S. Para ello se produce una división de la memoria, con el objetivo de tener varios programas. Hay que disponer asimismo de un sistema de protección de la memoria para que el usuario tenga sus programas seguros.

Existen procedimientos de spooling, que consiste en la operación simultánea de periféricos conectados en línea. Es decir, se usan los dispositivos E/S al mismo tiempo que trabaja el computador. Por ejemplo, se cargan las tarjetas al mismo tiempo que se trabaja en otra cosa.

Lo que realmente lo diferencia del sistema operativo por lotes es el tiempo compartido: El computador no dedica todo el tiempo a un solo trabajo.

Cuarta generación (1980−1990).

- Tecnología:

Se integra la UCP en un solo chip: el microprocesador.

Los circuitos integrados son LSI (3000−30000) y VLSI (más de 30000).

- Modelos:

IBM PC (1981), IBM PC XT (1982), IBM PC AT (1984), IBM PS/2 (1987), VAX (DIGITAL, 1980), CRAY X−MP (1983).

- Modo de funcionamiento:

El software es fácil de usar. Aparecen los sistemas operativos MS−DOS y UNIX. Aparecen los sistemas operativos de red y los sistemas operativos distribuidos. En los primeros, el usuario que se conecta a la red es consciente de la existencia de muchos ordenadores de manera que debe saber como conectarse al ordenador que quiera. En los sistemas operativos distribuidos, en cambio, el usuario no sabe en que ordenador se ejecuta su programa o donde están los datos, se trata de algo transparente al usuario.

Quinta generación (1990−¿?).

- Tecnología:

Circuitos con más de un millón de componentes. Aparecen nuevas arquitecturas (paralelismo: los programas se ejecutan en paralelo). Surge la tecnología óptica: fibra óptica, CD−ROM, etc...

- Modelos:

CONNECTION MACHINE: Máquina masivamente paralela.

- Modo de funcionamiento:

Inteligencia artificial y sistemas expertos (se tienen grandes bases de datos para simular la inteligencia artificial).

- ***Evolución de los niveles***

Los primeros computadores digitales (años 90) sólo tenían dos niveles (convencional y lógico digital). Los circuitos digitales eran voluminosos, poco fiables, y difíciles de construir.

El nivel de programación se añadió para:

- Simplificar la electrónica: Por ejemplo, un microprograma puede dividir una multiplicación en sumas, las cuales son ejecutadas directamente.
- Facilitar las escritura de compiladores: Que ya no tienen que hacer una traducción tan fina.
- Ejecutar los programas más rápidamente: El microprograma está en ROM, que es más rápida que la RAM.

En los setenta el nivel de microprogramación estaba ya plenamente difundido.

En los cincuenta aparecieron los ensambladores y compiladores, en los sesenta el sistema operativo

En los años ochenta (a principios) se elimina el nivel de microprogramación para dar paso a las máquinas RISC, por los siguientes motivos:

Cuanto más complicado se hace el lenguaje máquina, más grande, complicado y lento se vuelve el microprograma (ya que tiene que llamar a procedimientos).

La velocidad de la memoria RAM se aumentó con la tecnología (memorias de semiconductores).

Es difícil escribir, depurar y mantener el microcódigo. Así, por ejemplo, si hubiera un fallo en el microprograma, por estar en ROM habría que cambiar la ROM de cada ordenador con ese microprograma.

En cambio, en las máquinas RISC el número de instrucciones es muy reducido, por lo que no hace falta microprogramación.

TEMA 2

ORGANIZACIÓN DE COMPUTADORES

- ***Introducción***

Un computador consta de procesador (también llamado CPU), memoria y dispositivos de E/S.

Esquemáticamente lo podemos representar como:

- ***Procesadores***

La CPU es el cerebro del computador. Su función es ejecutar programas almacenados en la memoria principal. La CPU se compone de:

- Unidad de control (UC): Se encarga de leer una tras otra las instrucciones del programa que está en la memoria principal, así como de generar las señales necesarias para su ejecución. Estas señales están sincronizadas con un reloj.
- Unidad aritmético lógica (UAL): Es la encargada de realizar las operaciones elementales sobre los datos de la memoria (sumas, restas, etc.).
- Registros: Pequeña memoria de alta velocidad para almacenar resultados intermedios y cierta información de control. Por ejemplo: el PC (contador de programa).

Ejecución de una instrucción.

El proceso que se sigue es:

- La CPU extrae de la memoria la siguiente instrucción y la lleva al IR (registro de instrucción.
- Se incrementa el PC.
- Se determina el tipo de la instrucción.
- Se consulta si la instrucción necesita datos de la memoria.
- Se extraen los datos y se cargan en los registros.
- Se ejecuta la instrucción.
- Se almacenan los datos en el lugar apropiado.

Ejecución de instrucciones en paralelo.

Cada vez se intenta que las máquinas sean más rápidas, pero existen límites económicos y físicos. Por eso se recurre a una máquina con varias ALU's o incluso varias CPU's.

Las máquinas paralelas se dividen en tres categorías (establecidas por Flynn en 1972) según el número de instrucciones y datos:

- SISD: Flujo de instrucciones simples, flujo de datos simple.
- SIMD: Flujo de instrucciones simples, flujo de datos múltiple.
- MIMD: Flujo de instrucciones múltiple, flujo de datos múltiple.

Máquinas SISD.

Constan de un programa y un conjunto de datos. Se extrae una instrucción y se ejecuta, se extrae otra y así. El paralelismo se logra extrayendo e iniciando la siguiente instrucción antes de terminar la que está en curso.

Existen 2 modelos posibles:

- Varias unidades funcionales.

La UC extrae una instrucción y la manda a una unidad funcional, extrae la siguiente, y así hasta que no se pueda avanzar. No se podrá avanzar en los dos casos siguientes: si todas las unidades funcionales están ocupadas o si hace falta un operando que se está calculando en otra instrucción.

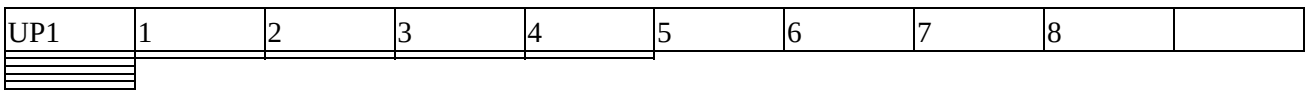
Se supone que el tiempo en ejecutar una instrucción es mayor que el de extraerla. Si no fuera así, sólo se usaría una unidad funcional de forma simultánea.

El esquema de funcionamiento es:

- Procesamiento en línea.

Consiste en separar la ejecución de cada instrucción en partes. Cada parte la ejecuta una unidad de procesamiento de la CPU. Por ejemplo:

En este ejemplo la CPU tiene cinco unidades de procesamiento. Cada instrucción debe pasar por todas ellas. Así, la instrucción uno tarda cinco unidades de tiempo en ejecutarse, pero las demás es como si sólo tardaran una unidad de tiempo a todos los efectos.



Máquinas SIMD.

En ella tendremos un programa con múltiples conjuntos de datos.

También existen dos modelos:

- Máquina vectorial.

Para cada entrada de la ALU se tiene un vector con `n' entradas y una sola variable. O sea, la ALU es una unidad vectorial capaz de realizar operaciones con vectores. Esquemáticamente:

- Procesador de arreglos.

Consiste en una malla cuadrangular de elementos procesador/memoria. Éstos reciben las instrucciones de la UC, las cuales son ejecutados por todos los procesadores. El esquema es:

Con esta estructura se pueden hacer operaciones con matrices. Simplemente con dar una instrucción, cada procesador realizaría dicha operación con el elemento de la matriz correspondiente.

Máquinas MIMD.

En estas máquinas distintas CPU's manejan distintos programas compartiendo a veces una memoria común. El esquema habitual del sistema multiprocesador es el siguiente:

Todas las CPU's acceden a la memoria compartida a través del bus, por lo que pueden surgir dificultades si un gran número de procesadores quieren acceder a la memoria al mismo tiempo. Posibles soluciones para este problema pueden ser:

- Tener memorias locales para cada procesador, en las cuáles se almacenan los datos y programas no comunes, mientras que sólo los comunes estarán en la memoria compartida.
- Tener más de un bus.
- Usar memoria caché, que es una técnica para mantener en cada procesador las palabras de memoria usadas con mayor frecuencia.

- ***Memoria***

Sirve para almacenar programas y datos. Su unidad básica es el bit.

Se divide en celdas, cada una de las cuales se identifica por una dirección. La unidad más pequeña direccionable es el byte (8 bits) (por ejemplo, una máquina de 16 bits tendrá instrucciones para operar sobre palabras de 2 bytes). Los bytes se agrupan en palabras, la mayor parte de las instrucciones operan sobre palabras.

Los bytes en una palabra se pueden numerar de izquierda a derecha (lo que se llama big endian, usado por Motorola) o de derecha a izquierda (lo que se llama little endian usado por Intel). Esto provoca un problema: la falta de una norma en el ordenamiento puede ocasionar cierta incompatibilidad en la transferencia de datos. Dicho problema no se soluciona con el simple intercambio de los bytes, ya que sólo habría que hacer el cambio para los bytes que guardan cifras numéricas, no para las cadenas de caracteres. Posibles soluciones son:

- Incluir un encabezado que indique el tipo y el tamaño de los bytes.
- Que ambas máquinas, antes de la transferencia, se pongan de acuerdo en qué formato usan.

Memoria secundaria.

Se usan para memoria secundaria los siguientes dispositivos: Cintas magnéticas, discos magnéticos, discos flexibles, discos ópticos, discos RAM. En ellos se utiliza como disco una parte de la memoria principal previamente reservada. Sus ventajas son: acceso instantáneo, facilidad para un ordenador sin disco, etc.

- ***Dispositivos de E/S***

Sirven para la comunicación con el exterior. Aquí se incluyen dispositivos tales como terminales, módems, ratones, impresoras. Existen dos enfoques distintos para tratar la E/S:

Enfoque uno.

Para grandes computadores. En él existen unos procesadores de E/S (también llamados canales), a los cuales se conectan los periféricos; un procesador de E/S es un pequeño procesador que se encarga de la comunicación con el exterior.

La CPU carga en los procesadores de E/S, a través del bus de E/S, el programa donde se indica lo que hay que hacer, con lo cual ella se libera de trabajo. Cuando el procesador de E/S acaba, mando una interrupción por el bus de E/S.

Esquemáticamente:

Enfoque dos.

Es el utilizado en computadores personales. En él, se añaden a la tarjeta matriz los controladores necesarios (de vídeo, de teclado, de disco, etc.). Estos controladores manejan la E/S al dispositivo y el acceso al bus, y funcionan con interrupciones o mediante DMA (acceso directo a memoria). Es el arbitrador el que decide quién accede al bus.

El esquema es:

TEMA 3

CONCEPTOS FUNDAMENTALES

DEL NIVEL DEL SISTEMA OPERATIVO

- ***Definición y objetivos del sistema operativo***

El sistema operativo es un conjunto de programas que se encargan de algo. Funciones:

- Ocultar toda la complejidad del hardware al programador: Para ello el sistema operativo presenta una serie de funciones más fáciles de usar que el propio hardware; o sea, el sistema operativo es como una máquina virtual que hace de capa que envuelve el hardware. Los programas de aplicación no tratan directamente el hardware, sino que lo hacen a través del sistema operativo Por ejemplo, un programa de aplicación puede ser leer un bloque de ficheros. Es el sistema operativo quien maneja el disco o disquete (el programa no se preocupa de ello) y transfiere los datos. El programa sólo da la orden de leer.
- Administrar los recursos de la máquina: El sistema operativo asigna estos recursos (CPU, memoria, dispositivos de E/S) entre los distintos programas de aplicación que se ejecutan. Además, el sistema operativo contabiliza los recursos usados por los usuarios, y decide quien utiliza cada recurso en caso

de conflicto.

- ***Estructura, componentes y servicios del sistema operativo***

- ***Llamadas al sistema***

El sistema operativo se divide en varios módulos cada uno con una función determinada. Tendremos una interfaz muy bien determinada para usar estas funciones, los programas acceden a estas funciones mediante una llamada al sistema. Cada sistema operativo tiene llamadas al sistema distintas.

A cada llamada al sistema le corresponde un procedimiento que puede ser llamado por el programa de usuario. El procedimiento no es sólo la llamada al sistema, incluye más cosas. El procedimiento debe comenzar con una llamada al sistema. Por ejemplo:

count=read(file, buffer, nbytes);

Esto lee de file, lo mete en buffer, y lee nbytes, que es lo que le pedimos. Lo que ha podido leer (número de bytes) lo mete en count. Este ejemplo es de UNIX, es como en C ya que UNIX está escrito en C. La función de biblioteca `read' es el procedimiento que incluye `READ', lo que es propiamente la llamada al sistema. Pero antes de `READ' el procedimiento debe incluir un TRAP, para avisar que va a llamar al sistema. Sin embargo, desde el programa usuario lo que se ve es la llamada al procedimiento `read'.

- ***Procesos***

Un proceso es un programa en ejecución. Consta de un código ejecutable, datos, pila del programa, contador de programa, puntero a la pila y otros registros e información necesaria para ejecutar el programa.

Por ejemplo, en un sistema donde hay tiempo compartido, cada cierto tiempo el sistema operativo decide parar un proceso y arrancar otro nuevo. Cuando se rearranca el primer proceso, debe hacerlo en el mismo punto en que se paró. Así, si estaba leyendo un fichero en una posición, habrá que seguir en la posición siguiente. Por lo tanto, el sistema operativo debe guardar la información necesaria para conseguirlo. Para ello, los sistemas operativos tienen una tabla de procesos, en la que hay un registro por cada proceso, y que contiene la información precisa.

Un proceso (en ejecución o parado) consta de: su imagen en memoria en el espacio de direcciones que ocupa dentro de la memoria y una entrada en la tabla de procesos.

Un proceso a su vez puede crear nuevos procesos, llamados procesos hijos del primero (se puede crear así una estructura jerárquica de procesos).

Cuando un proceso recibe una señal del sistema operativo se detiene para iniciar un procedimiento de tratamiento de la señal (tras salvar por donde iba). Las señales son las equivalentes software de las interrupciones en hardware. Por ejemplo, un proceso le puede decir al sistema operativo que le mande una señal si un mensaje que es mandado a otra máquina no recibe contestación pasado cierto tiempo, con el objeto de reenviar el mensaje.

El sistema operativo asigna a cada usuario lo que se llama un `uid' (identificador de usuario). Cuando un proceso arranca, lo hace con la identificación `uid' del usuario.

- ***Ficheros***

Sirven para almacenar información. Se agrupan en directorios. El sistema de ficheros será así una estructura

jerárquica compuesta de ficheros y directorios. Cada fichero o directorio se puede nombrar con una ruta de acceso o `path'. La ruta absoluta es la que parte del directorio raíz, indicando también todos los directorios intermedios. Pero se puede dar una ruta relativa partiendo del directorio actual (directorio de trabajo).

Los ficheros deben tener un sistema de protección para que en un sistema multiusuario no todos los usuarios puedan acceder a cierta información restringida. Por ejemplo, en UNIX hay nueve bits para indicar los permisos, tres para el usuario, tres para el grupo y los bits restantes para los demás usuarios. Cada grupo de tres indica respectivamente `rwx' (lectura, escritura y ejecución). Podemos activar los bits que queramos. El sistema operativo comprueba primero, antes de hacer algo, si lo que queremos hacer nos está permitido.

En muchos sistemas operativos se proporcionan mecanismos para facilitar la E/S al usuario. Esto se hace mediante una abstracción en la que se crean ficheros especiales que representan dispositivos E/S. Por ejemplo, para leer o escribir en una impresora podemos usar su fichero especial. Hay dos tipos de ficheros especiales.

- De bloques: Para dispositivos formados por bloques y de acceso aleatorio. Ejemplo: discos.
- De caracteres: Para dispositivos que se comportan como cadenas de caracteres. Ejemplo: terminales, impresoras, redes,...

Cuando se arranca un proceso ya hay tres descriptores de ficheros que se pueden usar.

- Descriptor de fichero 0: entrada estándar (normalmente el teclado).
- Descriptor de fichero 1: salida estándar (normalmente la pantalla).
- Descriptor de fichero 2: salida estándar de errores (normalmente la pantalla).

Una característica adicional relacionada con ficheros y procesos son los tubos o `pipes', que sirven para interconectar dos procesos.

- ***Intérprete de comandos***

Cuando se hace una llamada al sistema operativo se entra en su núcleo, pero hay más programas que no forman realmente parte del sistema operativo, como linkadores, compiladores, etc. El intérprete de comandos tampoco pertenece al sistema operativo pero todos los sistemas operativos están asociados a uno.

Cuando un usuario inicia una sesión, arranca el intérprete de comandos (shell), que presenta el símbolo `>'. Cuando le damos una orden, se crea un proceso hijo, y al acabar se vuelve al proceso del intérprete de comandos.

El sistema operativo no debe entenderse sólo como el código que da servicio a las llamadas al sistema que podríamos denominar núcleo. Existen también otras utilidades auxiliares que también forman parte de él. El intérprete es uno de estos programas que no están incluidos en el núcleo. Se trata de una interfaz entre el sistema operativo y el usuario. Cada vez que un usuario entra en una máquina se arranca un `shell', que envía un símbolo para indicarle que espera una orden. Si el usuario escribe un comando, el `shell' crea un proceso hijo que ejecuta ese comando y el `shell' espera un comando. Si el usuario escribe uno de ellos, el `shell' crea un proceso hijo que ejecuta ese comando y el `shell' espera a que termine el proceso hijo y vuelve a estar en espera a un nuevo comando.

Se puede hacer que el intérprete redirija la entrada y la salida. Ejemplo:

date > fichero

La salida de date se almacena en fichero.

sort < f1 > f2

sort toma como entrada f1 y la salida la envía a f2.

También el shell puede crear un `pipe' entre dos procesos, enviando la salida del primero a la entrada del segundo. Por ejemplo:

sort f1 f2 f3 | sort > /dev/lp

De esta forma sort ordena f1, f2, f3 y lo manda al fichero especial lp, que en UNIX es la impresora.

Se le puede indicar al `shell' que no espere a que acabe el proceso o procesos hijos que haya creado. Para ello se pone & al final de la orden. Esto se llama ejecución en `background'.

- ***Componentes y servicios del sistema operativo***

Hay varios componentes en un sistema operativo:

- Administrador de procesos: Se encarga de crear y eliminar procesos, suspenderlos y reanudarlos, proporcionar mecanismos para la comunicación y sincronización de procesos, y para el manejo de bloqueos de procesos.
- Administrador de memoria: Se encarga de controlar las zonas de memoria que están siendo utilizadas y quién las usa, decidir que proceso se va a cargar en memoria en el caso de que haya espacio, gestionar y recuperar el espacio de memoria.
- Administrador del sistema de E/S: Su función es la de presentar una interfaz general con los manejadores de dispositivos (que son la parte del sistema operativo que controla los periféricos de la máquina), es decir, una interfaz igual para todos esos manejadores y, además, ha de tener manejadores para dispositivos específicos (los manejadores pueden ser diferentes, pero la interfaz debe ser la misma).
- Administrador de archivos: Se encarga de gestionar el espacio en disco, gestionar ficheros (crear, borrar, leer, escribir...), directorios, copias de seguridad y establecer una correspondencia entre archivos y almacenamiento secundario (debe saber en que lugar del almacenamiento secundario está cada archivo).
- Sistema de protección: Sirve para controlar el acceso a los recursos (viendo si el usuario tiene o no permiso).
- Sistema de comunicación: Gestiona los accesos a la red (se encarga de la conexión de los procesadores del sistema para compartir los recursos de la red).

Los servicios de un sistema operativo son las tareas realizadas por todos los componentes del sistema operativo

Distintas estructuras en sistemas operativos.

- Sistemas monolíticos.

No hay una estructura definida. Son sistemas simples. El sistema operativo se compone de una serie de procedimientos que se encuentran al mismo nivel cada uno de los cuáles puede llamar a cualquier otro.

En cierto momento el programa de usuario hace una llamada al núcleo del sistema operativo, cuando se va a hacer la llamada se ejecuta la instrucción `TRAP', que cambia la máquina de modo usuario a modo privilegiado, transfiriendo el control al sistema operativo (en este modo privilegiado se pueden ejecutar todo tipo de instrucciones, en modo usuario algunas instrucciones no se pueden realizar, como por ejemplo las de

E/S).

En ese instante el sistema operativo examina la llamada al sistema y mira cuál es el procedimiento de servicio requerido, se busca en la tabla ese procedimiento de servicio, y tras ejecutarlo se devuelve el control al programa de usuario.

Así pues, en este esquema tenemos un procedimiento principal, que llama a los procedimientos de servicio solicitados por el servicio, éstos, a su vez, pueden llamar a procedimientos auxiliares para ejecutar diversas operaciones. Por tanto, un sistema operativo monolítico se estructura en tres niveles: procedimiento principal, de servicio y auxiliares.

El MS/DOS es de este estilo, pero no tiene protección de hardware (no tiene modo privilegiado, por lo que el usuario puede hacer lo que quiera).

- Sistemas nivelados.

Existen varios niveles jerarquizados. Por ejemplo, el sistema operativo IHE (1968) tiene 6 niveles: El operador (5), programas de usuario (4), gestión de E/S (3), comunicación entre operador y procesos (2), gestión de memoria principal y secundaria (1) y asignación del procesador y multiprogramación (0).

Los niveles no se tienen que preocupar de los niveles inferiores. Por ejemplo, el nivel cuatro no se tiene que preocupar de la E/S, de la comunicación con la consola del operador, etc... El nivel cinco es parecido al intérprete de comandos. Con este sistema operativo no existe protección de un nivel a otro.

Otro ejemplo es el sistema MULTICS en el que existe una especie de protección de niveles: el acceso a un nivel inferior desde uno superior se realiza a través de una interrupción software tipo `TRAP'.

- Máquinas virtuales.

Un sistema operativo (o un sistema en tiempo compartido) debía ofrecer dos tipos de servicios, gestionar la multiprogramación y presentar una máquina virtual más fácil de utilizar por el usuario. Un sistema operativo en máquina virtual separa los dos tipos de servicios mencionados. Esto se aplicó, por ejemplo, en la máquina 370 de IBM: el monitor de la máquina virtual es el VM/370 y se ejecuta sobre el hardware de la 370, el VM/370 es como si dividiera la máquina en varias, cada una con sus propios recursos. Cada una de ellas es un sistema monoprogramable (un solo programa). Es el VM (Virtual Machine) quien gestiona la multiprogramación.

Cuando un usuario quiere hacer algo lo recibe el CMS (Conversational Monitor System), y luego el VM es el que lo gestiona todo. El CMS actúa sobre el VM, no sobre el hardware.

Este ha sido el esquema general de las máquinas virtuales. El VM intenta que las máquinas virtuales sean iguales a la máquina real. O sea, intenta que parezca que cada usuario tenga sus propios recursos (disco, etc.), aunque sea el mismo para todos. Cada usuario puede tener su propio sistema operativo. Este esquema no tuvo mucho éxito ya que es difícil de implementar.

- Modelo cliente−servidor.

Es un modelo estándar para las comunicaciones de red. Ahora, la máquina tendrá un núcleo que se encarga de gestionar el hardware y comunicar los procesos clientes y servidores.

Esto tiene la ventaja de que es muy simple adaptar este esquema a un sistema distribuido. El cliente no tiene por que saber si el servidor está en su misma máquina o no. En este modelo un servidor no es más que un

proceso que espera a recibir una petición del proceso cliente. El caso típico de funcionamiento es:

- Se comienza el proceso servidor.
- El servidor espera a un proceso cliente mientras duerme.
- Se comienza un proceso cliente, en el mismo sistema del proceso servidor o en otro.
- El cliente manda su petición al servidor (ejemplo: pedir la hora, leer o escribir en un fichero sito en la máquina del servidor, ejecutar un comando en la máquina del servidor, imprimir algo, escribir algo en la pantalla del servidor, etc.).
- El servidor despierta, hace lo que se le ha pedido, y luego vuelve a la cama. Normalmente es el sistema operativo el que despierta al servidor.

Hay dos tipos de servidores:

- Iterativos:

Esto ocurre si la petición del cliente se puede ejecutar en un tiempo corto y conocido. El propio proceso servidor gestiona la petición del cliente; cuando termina, gestiona la petición de otro cliente, etc. Ejemplo: los servidores de la hora.

- Concurrentes:

Esto ocurre si la petición requiere un tiempo para resolverla que depende de la petición en sí, y no se sabe cuánto tiempo se puede tardar (ejemplo: leer o escribir ficheros depende de la longitud del fichero). En este caso, los servidores, al llegar la petición, crean un nuevo proceso que es el que realmente gestiona la petición del cliente, mientras que el servidor espera una nueva petición. Si llega otra, vuelve a crear otro proceso, y así sucesivamente (por eso son concurrentes, pues no hace falta esperar a que acabe el primer proceso creado).

En un momento dado puede haber varios procesos ejecutándose concurrentemente en la misma máquina dando servicio a distintos clientes. Esto no ocurre en los procesos iterativos. Se utilizan para leer o escribir ficheros.

TEMA 4

PROCESOS CONCURRENTES

- ***Planteamiento del problema de sincronización y planificación***

Un proceso es un programa en ejecución. En un sistema en tiempo compartido cada proceso es ejecutado unos cuantos milisegundos, luego se pasa a otro, y así sucesivamente. A esto se le llama pseudoparalelismo. El mismo procesador cambia de un programa a otro. El sistema operativo es el que simula el paralelismo. Al cambio entre un proceso y otro se le llama multiprogramación.

La ejecución de un programa no es reproducible en sucesivas ejecuciones del mismo (en una situación se ejecutarán cuatro instrucciones de golpe, en otras seis...).

Los procesos no se van a ejecutar a una velocidad uniforme. La velocidad de ejecución dependerá de lo cargado que esté el sistema. Por tanto, no podemos hacer suposiciones de tiempo en nuestros programas. No podemos decir: cuando pase tal tiempo leer de cinta, por ejemplo.

En todos los sistemas operativos habrá mecanismos para la creación de procesos. Por ejemplo, en UNIX, la llamada al sistema FORK crea otro proceso: hace una copia exacta del proceso que llama a FORK, de manera que ambos procesos, padre e hijo, se ejecutarán en dos a partir de ese momento. Para distinguir si estamos en

el proceso padre o en el hijo se mira lo que devuelve FORK:

```
if (fork()==0) /* es el hijo */

{

...

}

else /* es el padre */

{

...

}
```

De esta forma podemos hacer cosas diferentes para los procesos padre e hijo.

En MS/DOS hay una llamada equivalente a FORK que carga un fichero binario en memoria, pero como este sistema operativo no es multiprogramado, realmente primero se ejecuta el hijo, y luego el padre. No hay multiprogramación pues hay una llamada al sistema que carga desde el proceso padre un fichero binario en memoria que pasa a ejecutarse mientras el proceso padre permanece suspendido.

Aunque los procesos son independientes, a veces necesitaremos que interactúen entre ellos. Por ejemplo:

```
cat f1 f2 f3 | grep palabra
```

De esta forma grep busca palabra en f1, f2, f3. Dependiendo de la velocidad relativa de grep y cat puede que grep deba esperar a que cat mande algo; entonces grep debe bloquearse de alguna forma mientras espera.

En general un proceso puede bloquearse si no tiene la entrada disponible. Pero también el sistema operativo puede decidir bloquear un proceso para ejecutar otro.

Los estados de los procesos son tres:

- En ejecución: Si está usando el procesador.
- Bloqueado: Si no está usando el procesador.
- Listo: Se puede ejecutar (no está bloqueado) pero no le toca ejecutarse pues hay otro proceso ejecutándose en ese momento.

Los posibles paso de uno a otro son:

- Transición 1: Si requiere una E/S que no está lista; si lo decide el sistema operativo; o a veces el propio proceso mediante la instrucción BLOCK.
- Transición 2 y 3: Es el planificador el que decide qué proceso se ejecuta en cada momento.
- Transición 4: Si el proceso esperaba algo y ya ha llegado (no puede pasar de bloqueado a en ejecución directamente).

Cuando hay una transición de tipo 1, el planificador decide que proceso del grupo listo es el que va a ejecutarse. También es el planificador el que decide cuando hacer las transiciones 2 y 3. Asimismo, decide

qué proceso se ejecuta si el que estaba ejecutándose acaba. También decide, si hay una transición 4, si ese proceso debe pasar a ejecutarse o esperar en listo. O sea, el planificador decide en cuatro casos diferentes.

En la tabla de procesos se dice en que estado está cada proceso. Además, en esta tabla tendremos la información necesaria para que un proceso listo pase a ejecutarse (a esto se le llama cambio de contexto).

Funcionamiento del planificador.

En memoria habrá unos vectores de interrupción que contienen las direcciones del procedimiento de servicio de la interrupción correspondiente. Los pasos que se siguen son:

- Supongamos que estamos ejecutando un proceso y que llega una interrupción debido a otro proceso que la solicita (por ejemplo, del disco). El hardware es quién guarda el PC y los datos necesarios, y también quien carga el nuevo PC que le indica el vector de interrupciones.
- Ahora pasa a ejecutarse el procedimiento de servicio, que es el que copia en la tabla de servicio lo necesario para que este proceso, que pasa a listo, pueda volver a ejecutarse en el futuro.
- Una vez que llega la interrupción del disco que indica que se ha acabado la E/S este proceso, que estaba bloqueado, pasa a listo (se pone el estado del proceso que esperaba E/S de disco de bloqueado a listo).
- Ahora es el planificador quien decide qué proceso se va a seguir ejecutando, pues ahora hay dos procesos listos.
- Es el despachador quien realmente realiza el cambio de contexto para ejecutar el proceso decidido por el planificador.

- ***Comunicación entre procesos***

Los procesos se comunican a menudo entre sí (por ejemplo, si un proceso quiere imprimir debe comunicarse con el proceso de impresora para decirle el nombre del fichero que quiere imprimir, otro ejemplo, si un proceso quiere leer de un fichero debe decirle al proceso del disco que fichero es).

Normalmente lo que ocurre es que los procesos que deben comunicarse comparten un área de memoria. Por ejemplo, cuando un proceso quiere imprimir escribe en un directorio especial, llamado directorio de spooler, el nombre del fichero a imprimir. Más tarde, el proceso de impresora, si lo hay, lo imprime y borra el nombre del fichero del directorio. Así pues, el directorio spooler será una tabla con los ficheros a imprimir.

Suponiendo que hay dos procesos (PA y PB) que quieren imprimir. Habrá dos variables:

- sal: Posición del primer fichero que hay que imprimir (en el dibujo sal=4).
- ent: Primera posición libre (en el dibujo ent =7).

Los pasos son:

- Vamos a suponer que PA está ejecutándose, y quiere imprimir algo. Entonces lee la variable ent.
- En ese instante imaginemos que llega una interrupción de reloj y el planificador quiere que sea PB quien deba ejecutarse.
- PB, da la casualidad, que también quiere imprimir, por lo que lee ent, y escribe FB en la casilla 7 del directorio spooler. Luego, incrementa ent en una unidad.
- Cuando pasara a ejecutarse PA, éste pondría FA en la posición 7, que ya había leído antes, machacando FB. El demonio impresor no se daría cuenta, y FB se perdería.

Esto puede pasar si hay variables compartidas por varios programas, y se llaman condiciones de carrera. Son estas situaciones en que dos o más procesos comparten una zona común (bien de memoria o de disco) y el

resultado final depende de los momentos en que se hace la ejecución. Habrá que buscar alguna forma de solucionarlo.

Para evitar las condiciones de carrera lo que se hace es impedir que más de un proceso acceda simultáneamente (para leer o escribir) a una zona común. A esto se le llama exclusión mutua.

A la parte del programa donde se accede a la zona compartida se le llama sección o región crítica. Hay varias soluciones a las condiciones de carrera, pero todas ellas deben cumplir estas cuatro condiciones:

- Dos procesos no pueden estar nunca dentro de sus regiones críticas al mismo tiempo.
- No puede haber suposiciones de tiempo (ni de velocidad) de los procesos.
- Nunca un proceso que no esté en su región crítica va a poder bloquear a otro proceso (si se está en la región crítica sí se deberán bloquear los otros procesos).
- Ningún proceso debe tener que esperar un tiempo arbitrariamente largo (puesto que estamos en un sistema de tiempo compartido no vale esperar a que termine un proceso para seguir con el otro).

A lo largo de la historia han surgido varias soluciones, que veremos en el siguiente apartado. Algunas se han visto que no funcionan.

- ***Programación concurrente***

3.1. Exclusión mutua con espera activa

Prohibición de las interrupciones.

Es la primera solución de las condiciones de carrera: cuando un proceso quiere acceder a su sección crítica debe prohibir todas las interrupciones. Así nadie lo interrumpirá hasta que no termine con la región crítica; en dicho momento se habilitan de nuevo las interrupciones. El planificador ya podrá pasar a otro proceso.

Se asegura la exclusión mutua pero esto no es conveniente en los procesos de usuario, pues así los usuarios acapararían todo el procesador si pueden inhabilitar las interrupciones a su antojo.

Además, si hay varios procesadores en el sistema, prohibir las interrupciones sería para un solo procesador, y los demás podrían acceder a las variables compartidas.

Este método sólo es bueno en monoprocesadores dentro del núcleo del sistema operativo y en realidad no tiene espera activa.

Variables cerrojo.

Se tiene una variable cerrojo compartida con valor inicial cero. Si un proceso quiere entrar en su región crítica mira el cerrojo: si es cero entra, si es uno espera a que se ponga a cero. Cuando entra, pone el cerrojo a uno.

Cuando el proceso sale de su región crítica, pondrá el cerrojo de nuevo a cero. El problema es que el cerrojo es una variable compartida y, por tanto, puede haber condiciones de carrera con ella (por ejemplo, dos procesos pueden mirar el cerrojo a la vez, y si lo ven a cero, entran ambos en su región crítica simultáneamente).

Alternancia estricta.

Suponemos que hay dos procesos A y B. El proceso A sigue el siguiente bucle infinito:

```
while (TRUE)

{

while (turno!=0);

sección_crítica();

turno=1;

sección_no_crítica();

}
```

Aquí se ejecutan alternativamente la sección crítica y la no crítica.

La variable turno no la pueden modificar A y B al mismo tiempo (siempre se pregunta antes de entrar y le contesta el otro). Algo similar ocurre para el proceso B:

```
while (TRUE)

{

while (turno!=1);

sección_crítica();

turno=0;

sección_no_crítica();

}
```

Aquí tenemos la variable turno. Si turno es cero entonces el proceso A entra en su sección crítica; al salir pone turno a uno y entonces le toca al proceso B.

Cuando se hace el while (turno!=1); o while (turno!=0); se produce una espera activa pues se produce una comprobación continua sin hacer nada. La espera activa tiene el problema de que se pierde mucho tiempo de CPU.

Sin embargo, supongamos que inicialmente turno es cero. Entonces A entra en su sección crítica, y al salir pone turno a uno, con lo que entra B en su sección crítica. Cuando sale B pone turno a cero y empieza a ejecutar su sección no crítica. Supongamos ahora que esta sección no crítica es mucho más larga que el proceso A. Entonces A encuentra turno a cero, ejecuta su sección crítica, lo pone a uno, acaba también su sección no crítica y quiere de nuevo ejecutar la sección crítica. Pero entonces turno es uno y ni siquiera B ha entrado en su sección crítica.

Esto no cumple una de las condiciones, pues al ser el proceso A mucho más corto que el B, A tiene que esperar constantemente a B. El planificador se sigue ejecutando. Con este algoritmo sólo estamos imponiendo la alternancia al entrar en la sección crítica, no en la ejecución.

Solución de Peterson.

En ella ya no hay alternancia estricta.

Es como sigue:

```
#define FALSE 0

#define TRUE 1

#define N 2

int turno;

int interesado[N];

entrar_en_región (int proceso)

{

int el_otro;

el_otro=1−proceso;

interesado [proceso]=TRUE;

turno=proceso;

while (turno==proceso && interesado[el_otro]==TRUE);

}

salir_de_región (int proceso)

{

interesado[proceso]=FALSE;

}
```

N es el número de procesos que quieren acceder a su sección crítica.

turno es la variable que indica a quién le toca el turno.

interesado es un vector de enteros (en realidad de variables booleanas, 0 ó 1) de N elementos.

Antes de entrar en la región crítica se entra en la función entrar_en_región. Si el proceso que quiere entrar es cero, el otro se pone a uno y viceversa. Previamente a todo, el vector interesado se inicializa a FALSE, pero aquí se pone a TRUE interesado[proceso] (proceso es 0 ó 1 indicando si es el primer o segundo proceso).

Si el while no se cumple se entra en la región crítica. Si se cumple, se queda bloqueado en dicho while.

Al salir de la región crítica se ejecuta la función salir_de_región.

¿Por qué se hace la pregunta turno==proceso en el while si se acaba de asignar? Se hace por si justo al hacer turno=proceso el planificador cambia de proceso y el otro también hace turno=proceso (turno es una variable compartida). Si no se comprueba si turno==proceso ambos procesos se quedarían bloqueados ya que ambos interesado[el_otro]==TRUE. O sea, el primer proceso en hacer turno==proceso entra y el segundo en hacerlo se queda bloqueado.

Aquí no se fuerza a la alternancia estricta pues el mismo proceso puede entrar varias veces seguidas.

La instrucción TSL (Test and Set Lock).

Esta instrucción va a hacer una comprobación y una asignación a un cerrojo. Lee el contenido de una dirección de memoria y carga un valor distinto de cero. El hardware va a asegurar que estas dos cosas van a ser una operación indivisible, para que ningún otro proceso a procesador pueda acceder a la memoria entre ambas instrucciones que, por tanto, se ejecutarán de una sola vez.

Es como sigue:

entrar_en_región();

tsl registro, indicador

cmp registro, #0

jnz entrar_en_región

ret

salir_de_región;

mv indicador, #0

ret

Cualquier proceso que quiera entrar en su región crítica ejecuta la función entrar_en_región. En ella, la instrucción tsl guarda el indicador en registro, y carga un valor de uno, todo seguido. Ahora se compara con cero y si no es igual no se bloquea. Pero si el indicador es uno, el uno se almacena en el registro y se queda en un bucle hasta que el proceso que sale de la región crítica ejecute salir_de_región y ponga cero en el indicador.

Aquí no se da el problema de la variable cerrojo compartida, ya que al ser tsl una instrucción indivisible no puede haber dos procesos que vean el cerrojo a cero a la vez.

Hasta aquí los métodos de exclusión mutua con espera activa, los cuáles tienen el problema de que se desaprovecha tiempo de CPU sin hacer nada. Pero hay otro problema adicional. Supongamos que hay dos procesos H y L, siendo H de prioridad más alta que la del L. Además, supongamos que H está bloqueado en una E/S, por lo que podrá estar ejecutándose L, y de hecho vamos a suponer que lo está, y además que está en su región crítica. Si en ese momento termina la E/S de H, H pasa a estar listo, por lo que el planificador con gran probabilidad decidirá que se ejecute H ya que es de mayor prioridad. Si ahora H quiere entrar en su región crítica, debe esperar a que L termine de ejecutar la suya, pero L no va a terminar ya que se está ejecutando H, y L es de menor prioridad. Es decir, se ha producido un interbloqueo de procesos.

Por todo esto a partir de ahora vamos a ver soluciones para obtener la exclusión mutua pero sin espera activa.

3.2 Exclusión mutua sin espera activa

Son métodos en los que si un proceso no puede entrar en su región crítica se queda bloqueado, no en un bucle.

Dormir y despertar (sleep & wakeup).

Dormir va a ser una llamada al sistema que lo que hace es bloquear al que llama hasta que otro proceso lo haga despertar. Despertar es otra llamada al sistema que despierta al proceso que se le pase como parámetro. De esta manera el proceso bloqueado no gasta tiempo de CPU. Como ejemplo vamos a ver el problema del productor−consumidor.

Vamos a hacer las suposiciones de que existe un buffer finito que es compartido por los dos procesos, que el productor coloca los elementos en el buffer y que el consumidor quita elementos del buffer.

Productor y consumidor deben sincronizarse: si el buffer está lleno, el productor no debe colocar más elementos, sino bloquearse; y si el buffer está vacío será el consumidor el que deba bloquearse.

Lo que se hace es:

```
#define N 100

int cuenta=0:

productor()

{

while (TRUE)

{

producir_elemento();

if (cuenta==N) dormir();

dejar_elemento();

cuenta=cuenta+1;

if (cuenta==1) despertar(consumidor);

}

}
```

Esto es para el productor, para el consumidor:

```
consumidor();

{

while (TRUE)
```

```
{

if (cuenta==0) dormir();

retirar_elemento();

cuenta=cuenta−1

if (cuenta==N−1) despertar (productor);

consumir_elemento;

}

}
```

Se define N=100 como el tamaño del buffer.

cuenta es una variable entera que indica el número de elementos que hay en el buffer.

El productor está en un bucle infinito donde produce elementos y los deja en el buffer si hay sitio; si no lo hay, se duerme. Cuando puede dejar el elemento, lo hace incrementando además cuenta y pregunta si cuenta es uno, pues en ese caso anteriormente cuenta era cero, con lo cual el consumidor puede estar dormido, por lo que lo despierta (no estaría dormido si no ha intentado coger nada).

Para el consumidor, si el buffer está vacío, se duerme. Si no, coge un elemento, decrementa cuenta y mira si cuenta es N−1, pues en ese caso probablemente el productor estaba dormido, ya que antes cuenta era igual a N.

Sin embargo, hay un problema en este algoritmo: supongamos que el consumidor mira la variable cuenta y ésta es cero. En ese momento, el planificador decide cambiar al productor, el cuál produce un elemento, incrementa cuenta y manda un despertar al consumidor, pero éste aún no estaba dormido por lo que el despertar se pierde. Entonces, el consumidor se dormiría al volver a él, y el productor seguiría hasta llenar el buffer, durmiéndose también él. O sea, ambos se interbloquean.

Esto sucede ya que cuenta es una variable global, por lo que se produce una condición de carrera.

Una posible solución es guardar las señales de despertar, que éstas no se pierdan. Eso es lo que hace la siguiente solución.

Semáforos.

Un semáforo es una variable entera sobre la que tenemos dos operaciones diferentes:

- Bajar: (en inglés, wait). Lo que hace es comprobar el valor del semáforo. Si es mayor que cero, decrementa el valor y continúa. Si es cero, se pone a dormir. Esta es una operación atómica: comprobar el valor del semáforo, y decrementar o dormir, van a hacerse en un solo paso (debe garantizarlo el sistema operativo).
- Subir: (en inglés, signal). Puede que haya procesos esperando en el semáforo. Si hay procesos esperando, despierta a uno de ellos. El planificador decide cuál de ellos se va a despertar y el semáforo conserva su valor. Si no existen procesos esperando se incrementa el valor del semáforo. Si hay un proceso que encontró el semáforo a cero y se puso a dormir, entonces subir no incrementa el

semáforo. También ha de ser una operación atómica.

La operación bajar puede bloquear un proceso, pero la operación subir nunca bloquea a nadie.

Para conseguir que haya operaciones atómicas, el sistema operativo puede, por ejemplo, prohibir las interrupciones en ellas. Esto no era bueno en procesos de usuario, pero sí para el sistema operativo. Sin embargo, prohibir las interrupciones no sirve si existen varios procesadores en paralelo en el sistema, en ese caso, deberá haber un variable cerrojo, para cuyo acceso deberá usarse la instrucción tsl. La instrucción tsl implica espera activa, pero en este caso no hace perder mucho tiempo, pues incrementar o decrementar el semáforo tarda muy poco.

El programa correspondiente a la solución con semáforos sería, entonces:

```
#define N 100

typedef int semaforo;

semaforo mutex=1;

semaforo vacio=N;

semaforo lleno=0;

productor()

{

while(TRUE)

{

producir_elemento (&elemento);

bajar (&vacio);

bajar (&mutex);

dejar_elemento (elemento);

subir (&mutex);

subir (&lleno);

}

}

consumidor()

{

int elemento;
```

```
while (TRUE)

{

bajar (&lleno);

bajar (&mutex);

retirar_elemento (&elemento);

subir (&mutex);

subir (&vacio);

consumidor_elemento (elemento);

}

}
```

Tenemos tres semáforos: lleno indica el número de elementos que hay en el buffer, vacio indica el número de posiciones vacías que quedan en el buffer y mutex es un semáforo binario que sólo vale cero o uno.

En el productor, si vacio es cero es que no hay huecos vacíos. Entonces, el productor produce elementos (de tipo entero) en un bucle infinito, y llama a bajar(&vacio). Hemos supuesto que bajar y subir se llaman con la dirección del argumento. Si vacio es cero, entonces el productor se duerme.

Pero si vacio es distinto de cero, se llama a bajar(%mutex), que sirve para proteger la sección crítica. Si no hay ningún proceso en la región crítica, mutex valdría uno, se dejaría el elemento y se subirían mutex y lleno.

En el consumidor, si lleno es cero, el buffer está vacío, por lo que se comprueba con bajar(&lleno). Si no, se bloquea aquí, se hace bajar(&mutex). Si también se pasa, se quita un elemento, se suben mutex y vacio y se consume el elemento.

Supongamos por ejemplo que el productor se bloquea al bajar vacio. Seguirá así hasta que el consumidor no ejecute subir(&vacio).

Con mutex no se crean condiciones de carrera pues bajar y subir son operaciones indivisibles.

Hemos visto hasta aquí dos usos de los semáforos: el de exclusión mutua, del que se encarga el semáforo mutex y la sincronización. Lo semáforos vacio y lleno se usan para sincronizar a productor y consumidor.

Se les llama semáforos binarios a los que sólo pueden valer cero o uno. Dichos semáforos binarios se suelen usar muy a menudo para ocultar las interrupciones del siguiente modo:

- Vamos a tener un semáforo iniciado a cero para cada dispositivo de E/S.
- Cada vez que arranca una operación de E/S, se baja el semáforo, con lo cual se bloquea el proceso.
- El manejador de interrupciones ejecuta un subir para desbloquear el proceso (cuando llega la interrupción).

Vemos que esto no deja de ser un proceso de sincronización.

Los semáforos se usan mucho pero son difíciles de programar. Por ejemplo, supongamos que, en el productor, intercambiamos de orden bajar(&vacio) y bajar(&mutex); al llegar a bajar(&mutex), que es ahora la primera, mutex, que es uno, lo pone a cero. Pero si ahora el buffer está lleno, al hacer bajar(&vacio) el productor se bloquea, y el consumidor se bloquearía al hacer él bajar(&mutex). O sea, habría un interbloqueo.

Contadores de eventos.

Sobre un contador de eventos vamos a tener tres operaciones diferentes (al contador de eventos lo llamamos E):

- Leer: Devuelve el valor de E.
- Avanzar: Incrementa E. Se realiza en una operación atómica.
- Esperar: Se espera que E valga V o mayor que V. Si E es menor que V, se bloquea el proceso; si es mayor o igual que V, se pasa.

Se implementan como llamadas al sistema. Los contadores de eventos nunca se decrementan. Ejemplo del problema productor−consumidor:

```
#define N 100

typedef int contador_de_eventos;

contador_de_eventos ent =0;

contador_de_eventos sal=0;

productor();

{

int elemento, secuencia=0;

while (TRUE)

{

producir_elemento (&elemento);

secuencia=secuencia+1;

esperar (sal, secuencia−N);

dejar_elemento (elemento);

avanzar (&ent);

}

}

consumidor ()
```

```
{

int elemento, secuencia=0;

while (TRUE)

{

secuencia=secuencia+1;

esperar (ent,secuencia);

retirar_elemento(&elemento);

avanzar (&sal);

consumir_elemento (elemento);

}

}
```

Hay que tener en cuenta que debe cumplirse que el número de entradas ha de ser mayor o igual al número de salidas y que, además, la diferencia entre el número de entradas y el de salidas ha de ser menor o igual que el tamaño (ya que entradas es el número de elementos que hemos metido en el buffer y que salidas es el número de elementos que hemos sacado; tamaño es el ídem del buffer).

En el programa ent es un contador de eventos que cuenta los elementos metidos en el buffer y sal es un contador de eventos que cuenta los elementos sacados del buffer.

En el productor secuencia es una variable auxiliar local para comprobar si el elemento producido puede ser dejado, elemento es el elemento creado. Cuando se crea un elemento se incrementa secuencia. Con la instrucción esperar(sal, secuencia−N) se comprueba la condición segunda arriba explicada. Inicialmente sal es cero, secuencia es uno y N es cien, el productor debería esperar a que el consumidor sacara algo e incrementaría salida.

En el consumidor se incrementa secuencia y luego hace esperar(ent,secuencia), que no es más que comprobar la primera condición arriba explicada. Inicialmente, pone secuencia a uno, para retirar el primer elemento, pero deberá esperar a que ent se ponga a uno.

Como secuencia es una variable local tanto para el productor como para el consumidor no se producirán condiciones de carrera.

El valor máximo de ent, sal y secuencia estará limitado por el rango máximo permitido para los enteros en nuestro sistema.

Monitores.

Un monitor es un conjunto de procedimientos, variables y estructuras de datos agrupados en dicho monitor, que es, pues, un tipo especial de módulo o paquete.

Los procesos pueden acceder a los procedimientos que están dentro del monitor pero no a las estructuras

internas del monitor. Además, sólo un proceso puede estar a la vez activo dentro del monitor; no habrá dos procesos que ejecuten a la vez procedimientos del monitor. Suelen ser mecanismos de lenguajes de alto nivel por lo que el propio lenguaje debe permitirnos el uso de los monitores.

Es el propio compilador el que debe traducir el monitor, de forma que si al traducir hay un proceso activo no dejará que entre otro. Para ello se pueden usar semáforos binarios: cuando un proceso entre, que baje el semáforo.

Todas las secciones críticas de los procesos se tendrán así implementadas en procedimientos dentro del monitor de modo que será el compilador quien asegure la exclusión mutua entre las regiones críticas. Un ejemplo:

monitor ejemplo

integer i

condition c;

procedure_*();

_

_

end;

procedure_*();

_

_

end;

end monitor

La variable condition sirve para bloquear los distintos procesos caso de que sea necesario (por ejemplo en el caso del problema del productor−consumidor). Se les llama variables condición. Sobre ellas se pueden ejecutar dos instrucciones:

- esperar: Bloquea al proceso que llama y deja entrar a otro proceso que estuviera esperando.
- darpaso: Desbloquea al proceso que se había quedado bloqueado a la espera de una verdadera condición.

Ahora hay que solucionar que no haya dos procesos activos a la vez dentro del monitor. Para ello, la operación darpaso debe ser la última instrucción dentro del monitor que ejecuten los procesos, pues si no, habría dos procesos activos a la vez, o sea, inmediatamente después de dar paso a un proceso debe salir del monitor.

Esto es parecido a dormir y despertar, sólo que se asegura que darpaso será posterior a esperar.

El monitor agrupa todas las secciones críticas, y le encarga al compilador que evite que dos procesos accedan

a la vez a sus secciones críticas. O sea, se trata de una solución a más alto nivel que los semáforos.

Si, por ejemplo, el productor está dentro del monitor y ejecuta esperar, ya no está activo, por lo que podrá entrar el consumidor. Pero cuando éste ejecute darpaso, tendrá que salir del monitor.

Ejemplo: problema del productor−consumidor.

No vemos lenguaje C ahora, sino algo parecido a PASCAL, y es que el C no lleva implementados los monitores.

```
monitor ProductorConsumidor

condition lleno,vacio;

integer cuenta;

procedure depositar;

begin

if cuenta=N then esperar (lleno);

depositar_elemento;

cuenta:=cuenta+1;

if cuenta=1 then darpaso (vacio);

end

procedure retirar;

begin

if cuenta=0 then esperar (vacio);

retirar_elemento;

cuenta:=cuenta+1;

if cuenta=N−1 then darpaso (lleno);

end

begin

cuenta:=0;

end

end monitor
```

```
procedure productor

begin

while true do

begin

producir_elemento;

ProductorConsumidor.depositar;

end

end

procedure consumidor

begin

while true do

begin

ProductorConsumidor.retirar;

consumir_elemento;

end

end
```

Las variables condición son lleno y vacio, para esperar a que el buffer esté lleno y vacio respectivamente.

Dentro del bucle infinito del productor se llama al procedimiento depositar dentro del monitor. Algo similar ocurre en el consumidor con el procedimiento retirar.

cuenta es una variable del monitor para contar el número de elementos del buffer.

El buffer debe ser una estructura dentro del monitor, para asegurar la exclusión mutua.

El problema de los monitores es que pocos lenguajes los traen implementados.

Paso de mensajes.

Las soluciones vistas hasta ahora sirven para asegurar la exclusión mutua en sistemas donde la memoria se comparte entre varios procesadores. Pero no son válidas en sistemas distribuidos donde no hay memoria compartida. En este caso se puede usar el paso de mensajes.

En esta solución vamos a tener dos primitivas: enviar (send) y recibir (receive). Ambas van a ser funciones de biblioteca, de forma que en la librería tendremos procedimientos para ambas:

enviar (destino, &mensaje)

recibir (origen, &mensaje)

El paso de mensajes tiene, sin embargo, algunos problemas:

Se pueden perder mensajes en la red (por eso el transmisor debe esperar un asentimiento de que ha llegado bien el mensaje).

Si se pierde el asentimiento se reenviaría el mensaje, por lo que habría que seguir una secuencia (numeración) en los mensajes, para que se supiera que ha llegado el mismo repetido.

Hay, además, que identificar los procesos. Para renombrar procesos dentro de una misma máquina se puede hacer: proceso@maquina.dominios

El dominio sirve para aliviar la nomenclatura, de modo que en distintos dominios puede haber máquinas con el mismo nombre.

También hay que tratar la autentificación para que otro que no sea el destinatario no pueda leer el mensaje (para ello se pueden cifrar los mensajes).

Ejemplo: problema del productor−consumidor:

```
#define N 100

productor()

{

int elemento;

mensaje m;

while (TRUE)

{

producir_elemento (&elemento);

recibir (consumidor,&m);

formar_mensaje (&m,elemento);

enviar (consumidor,&m);

}

}

consumidor()

{
```

```
int elemento i;

mensaje m;

for (i=0 ; i<N ; i++)

enviar (productor,&m);

while (TRUE)

{

recibir (productor, &m);

extraer_elemento (&m, &elemento);

enviar (productor, &m);

consumir_elemento (elemento);

}

}
```

Lo primero que hace el consumidor es enviar N mensajes vacíos (o inicializados aleatoriamente) al productor. El sistema operativo, a la llegada, almacena los mensajes en un buffer hasta que el productor ejecute recibir. Si el sistema operativo no tiene ningún mensaje, al hacer recibir el productor se bloquea.

El productor, al recibir un mensaje, crea otro nuevo y lo manda al consumidor. Éste se bloquea hasta que no reciba el mensaje, caso de que ejecutara recibir antes de que el productor lo enviara; si fuera al revés, el sistema operativo en la máquina del consumidor, guardaría el mensaje hasta que el consumidor lo pidiera.

El propio sistema operativo gestiona los envíos, por lo que no se producirán condiciones de carrera.

Esto sería también una forma de sincronizar dos procesos que están en distintas máquinas. La sincronización se hace mediante el bucle for del consumidor y mediante enviar y recibir (entre el consumidor y el productor habrá siempre un flujo de N mensajes).

Al empezar, el buffer temporal del productor tiene N mensajes. El productor enviará un mensaje, pero si el consumidor no lo recibe (porque no quiere, no porque se pierda) enviará otro, y otro, hasta N; en ese momento se acabarán los mensajes. Cuando el consumidor recibe un mensaje, envía otro de forma que el productor a su vez podrá enviar su siguiente mensaje ya que el consumidor ya le ha informado de que ha consumido un mensaje. De este modo se garantiza que el productor se bloquea cada vez que manda N mensajes consecutivos sin que sean consumidos.

Hay otra forma de comunicar mediante paso de mensajes en la cual el sistema operativo no tiene buffer donde meter los mensajes mientras sean pedidos por los procesos (a estos buffers se le llama buzones). Los veremos más adelante.

Suponiendo que productor y consumidor están en dos máquinas distintas el paso se hace a través de un buzón. De esta forma el método es sin espera activa, pues si el buffer esta vacío y se ejecuta recibir, el proceso se bloquea. Se garantiza la exclusión mutua (aunque, por ejemplo, haya más de un productor) pues es el propio

sistema operativo quien se encarga de ellos.

Este método es parecido a los tubos (pipes), con la diferencia de que en el paso de mensajes se preserva la configuración de los mensajes. Por ejemplo, en un pipe si mando 100 caracteres, por el otro lado puedo leer dos veces 50 caracteres. En el paso de mensajes debo leer 100.

Rendezvous (encuentro o cita).

Es un caso de paso de mensajes pero sin buzón.

Lo que se hace es obligar a sincronizarse en cada paso al proceso que envía y al que recibe: el proceso que envía se queda a la espera de que el otro lo reciba (se queda bloqueado).

Equivalencia entre primitivas.

Vamos a ver algunos ejemplos de implementar algunas primitivas a partir de otras.

Semáforos para implementar monitores.

Supongamos que tenemos un compilador que puede soportar monitores pero en el sistema operativo sólo hay semáforos. ¿Cómo usamos los semáforos para implementar los monitores? Lo que hay que hacer es:

- Tener un semáforo por cada monitor iniciado a uno que controla la entrada al monitor.
- Para entrar en el monitor se ejecuta bajar.
- Para salir del monitor se ejecuta subir.
- Habrá que tener también un semáforo iniciado a cero por cada variable condición.

Inplementaríamos entonces las primitivas esperar y darpaso.

esperar:

subir mutex

bajar c

bajar mutex

darpaso:

subir c

mutex es el semáforo asociado al monitor. Al esperar, se sube mutex para que otro proceso pueda entrar en el monitor. c es el semáforo asociado a la variable condición.

Monitores para implementar semáforos.

Es el caso opuesto al anterior. Deberemos tener:

- Un contador para almacenar el valor del semáforo.
- Una lista enlazada para poner los procesos que estén esperando en el semáforo.
- Una variable condición por proceso.

Y cada primitiva:

bajar:

decrementa el contador si es mayor que cero.

si es cero, se hace esperar en la variable condición asociada al proceso (si es cero antes de decrementar, no después). Antes de hacer esperar, hay que añadir el proceso a la lista.

subir:

se comprueba si hay algún proceso en la lista. Si lo hay, se hace un darpaso y se actualiza la lista.

si no hay nadie en la lista, se incrementa el contador.

La exclusión mutua en el semáforo queda garantizada por la naturaleza del monitor.

Mensajes para implementar semáforos.

Si el sistema operativo dispone de llamadas al sistema enviar y recibir, se puede utilizar un sincronizador que recibe un mensaje con lo que se quiere hacer (subir o bajar) y el semáforo sobre el que quiere actuar. El sincronizador va a tener un contador y una lista de procesos bloqueados por cada semáforo. Si se puede realizar la operación se envía un mensaje al proceso que solicitó la llamada.

Si no se puede, no envía mensaje de vuelta al proceso, que se queda bloqueado y se almacena en la lista de procesos bloqueados por el semáforo. El proceso primero llamaría a enviar y luego a recibir.

bajar:

si el contador es mayor que cero: decrementar, enviar mensaje al proceso.

si el contador es igual a cero: añadir proceso a lista.

subir:

enviar mensaje al proceso.

si hay proceso bloqueado, desbloquear el proceso de la lista eliminada.

enviar mensaje al proceso eliminado.

Problemas clásicos de comunicación entre procesos.

Para ver que la solución a un programa concurrente es acertada se suele probar su funcionamiento con los problemas siguientes.

Problema de la comida de los filósofos.

Fue un problema planteado por Dijkstra.

Supongamos que en una mesa redonda están sentados cinco filósofos. Existen cinco tenedores, pero la comida es china por lo que cada filósofo precisa dos tenedores para comer.

La vida de los filósofos consiste sólo en pensar y comer.

Si un filósofo quiere comer debe tomar el tenedor que está a su izquierda y a su derecha pero sólo podrá hacerlo si los dos filósofos que estén a su lado no están comiendo.

Se trata, por tanto, de un problema de recursos compartidos.

Una solución posible es:

```
#define N 5

filosofo (i)

{

while (TRUE)

{

meditar ();

coger_tenedor (i);

coger_tenedor ((i+1)%N);

comer();

dejar_tenedor (i);

dejar_tenedor ((i+1)%N);

}

}
```

Esta solución no es buena, pues si todos los filósofos cogen a la vez el tenedor a su izquierda, todos se bloquearían a la vez.

Otra posibilidad sería, antes de bloquearse intentando coger el de la derecha, mirar si el filósofo de la derecha ya lo ha cogido, en cuyo caso soltaría el de la izquierda, esperaría un tiempo y volvería a intentarlo. Esto tampoco valdría si todos empiezan a la vez a coger el tenedor de la izquierda, y cuando esperan, esperan al mismo tiempo.

A este problema se le llama inanición: Cuando al intentar usar un recurso compartido, todos se bloquean y nunca lo usan.

Se podría pensar en que el tiempo que espera cada uno sea aleatorio. El interbloqueo se daría entonces muy raramente, pero en ciertos sistemas no debe ocurrir nunca (por ejemplo, en una central nuclear).

Una solución que sí es válida es con semáforos, cuando un filósofo empieza a comer bajaría un semáforo de forma que otro no podría comer hasta que ese semáforo no se subiera. O sea, coger los tenedores estaría dentro de la región crítica. Sin embargo, esta solución tiene un problema: que sólo un filósofo podría estar

comiendo a la vez, mientras que con cinco tenedores podrían estar comiendo hasta dos filósofos a la vez.

Veamos una solución que permite el máximo paralelismo posible:

```
#define N 5
#define izquierda (i+N−1)%N
#define derecha (i+1)%N
#define meditando 0
#define hambriento 1
#define comiendo 2
typedef int semaforo;
int estado[n];
semaforo mutex=1;
semaforo s[N];
filosofo(int i)
{
while(TRUE)
{
meditando();
coger_tenedores(i);
comer();
dejar_tenedores(i);
}
}
coger_tenedores (int i)
{
bajar(&mutex);
estado[i]=hambriento;
```

```
comprobar(i);

subir(&mutex);

bajar(&s[i]);

}

dejar_tenedores (int i);

{

bajar(&mutex);

estado(i)=meditando;

comprobar(izquierda);

comprobar(derecha);

subir(&mutex);

}

comprobar (int i)

{

if(estado[i]==hambriento&&estado[izquierda]!=comiendo&&

estado[derecha]!=comiendo)

{

estado[i]=comiendo;

subir(&s[i]);

}

}
```

En coger_tenedores se pone en el vector de entradas que el filósofo correspondiente está hambriento. Pero ese vector de estados está en la región crítica, así que primero se intenta bajar mutex. Ahora se comprueba si el filósofo i puede comer, lo cual se cumple si no están comiendo ni el de su izquierda ni el de su derecha; en ese caso se pone comiendo en el estado y se sube el vector de semáforos (el cual estaría inicializado a cero). Tras la comprobación se sube mutex porque ya se ha salido de la región crítica, y se baja s[i]; si el filósofo tiene estado comiendo, no se bloqueará, pero si no, s[i]=0 y se bloquea.

En dejar_tenedores, tras bajar mutex para entrar en la región crítica y poner el estado del filósofo en meditando, se comprueba para el filósofo de la izquierda y el de la derecha, por ejemplo, si el filósofo uno quiere dejar los tenedores, comprueba si el cero y el dos pueden comer. Así, para el cero, si el cero está

hambriento y el uno y el cuatro no están comiendo (el uno se sabe que no lo está), se sube el semáforo del cero, desbloqueando al filósofo cero, que posiblemente se quedó en el bajar(&s[i]); de coger_tenedores. Lo mismo se hace para el filósofo dos.

Problema de los lectores y escritores.

Tenemos una base de datos a la que pueden acceder muchos procesos. No nos importará que varios procesos lean a la vez en la base de datos, pero si un proceso está escribiendo en la base de datos no dejaremos que haya más procesos en ella, ni leyendo ni escribiendo.

Por tanto, tendremos un semáforo que será bajado por el escritor al entrar en la base de datos y subido al salir. También los lectores tendrán que bajar el semáforo para ver si había un escritor dentro.

La solución será entonces:

```
typedef int semaforo;

semaforo mutex=1;

semaforo bd=1;

int nl=0;

lector()

{

while(TRUE)

{

bajar(&mutex);

nl=nl+1;

if (nl==1) bajar (&bd);

subir (&mutex);

leer_de_base_de_datos();

bajar(&mutex);

nl=nl−1;

if (nl==0) subir (%bd);

subir (&mutex);

utilizar_datos_leidos();

}
```

```
}

escritor()

{

while (TRUE)

{

generar_datos ();

bajar (&bd);

escribir_en_base_de_datos();

subir(&bd);

}

}
```

Tenemos dos semáforos, inicializados a uno. mutex me asegura la exclusión mutua en nl, que es una variable compartida por los lectores. bd asegura la exclusión mutua en el acceso a la base de datos.

Cuando entra el primer lector, se pone bd a cero. El resto de lectores, al entrar, no bajan bd pues entonces se quedarían bloqueados. De igual forma, al salir es sólo el último lector el que sube bd, que se pondrá a uno cuando ya no haya lectores.

El escritor sólo podrá entrar cuando bd sea uno. Mientras el escritor está en la base de datos bd es cero y no puede entrar ningún lector, pues el primero debe bajar bd.

Redes de Petri.

Es una técnica de especificación formal de sistemas. Es una herramienta gráfica. La utilidad de una técnica de especificación o descripción formal (FDT) es la siguiente:

En un programa secuencial la forma de depurar es mediante prueba y error. Pero en programación concurrente es mucho más difícil escribirlos, depurarlos y corregirlos; es incluso complicado reproducir el caso específico en que un proceso puede fallar. Además, la fase de depuración de un proyecto es muy costosa. Por todo esto se intenta que la depuración sea fácil, pues además mientras más avanzados estemos en el proyecto más difícil será corregir un error de diseño inicial.

Las FDT intentan disminuir los costes de depuración, dando unas especificaciones muy claras de forma que un sistema será correcto si las cumple (se intenta probar que un sistema es correcto incluso antes de implementarlo).

Hay multitud de FDT's: Z, VDM, SOL, ESTELLE, LOTOS, redes de Petri, etc.

Las redes de Petri se utilizan para modelar sistemas siendo sus características las siguientes:

- Hacen posible modelar comportamientos que comprenden concurrencia, sincronización y recursos

compartidos.
- Con ellas se puede comprobar si un sistema cumple ciertas características (de esta forma se pueden detectar ciertos errores en el sistema incluso antes de su realización.
- Si el análisis del sistema no es satisfactorio no podríamos abordar la implementación; en caso contrario sí podríamos hacerlo con bastantes garantías.

Las aplicaciones más utilizadas son:

- Sistemas de fabricación (en industrias).
- Sistemas informáticos (sistemas operativos).
- Protocolos de comunicación.

Una red de Petri es un grafo orientado compuesto por dos clases diferentes de nodos:

- Los lugares: representados por una circunferencia o una elipse. Se llama P al conjunto de los lugares. P será un conjunto finito y no vacío.
- Las transiciones: representadas por un trazo rectilíneo vertical. El conjunto de transiciones es T y también es finito y no vacío.

Lugares y transiciones se unen alternativamente con arcos. O sea, un arco une una transición con un lugar o viceversa pero no entre dos lugares o dos transiciones.

Formalemente una red de Petri es una cuádrupla que se compone de P, T, Pre, Post.

Pre está incluido en PxT. Post está incluido en TxP. Si el par (p,t) está incluido en Pre, habrá un arco de p a t, y se dice que p es un lugar de entrada para t. Si el par (t,p) está incluido en Post, habrá una flecha de t a p, y se dice que p es un lugar de salida para la transición t.

Normalmente a las transiciones se les asocian acciones y a los lugares condiciones.

O sea, Pre marca las precondiciones de una transición. Asimismo, Post da las postcondiciones de una transición.

Marcado de las redes de Petri.

En una red de Petri, los lugares pueden contener un número positivo o nulo de marcas (tokens) las cuales se representan por puntos. Se habla de M(p) como el marcado del lugar p.

El número de marcas en una red de Petri no tiene por que ser constante a lo largo del tiempo. Se denomina M0 al marcado inicial, y se dice que es la distribución de marcas en cada uno de los lugares en el instante inicial.

Un marcado se puede representar como un vector, donde cada elemento se corresponde con un lugar.

Se dice que una transición está sensibilizada o validada, o que es franqueable, si todos sus lugares de entrada están marcados.

Una transición es franqueada o disparada si se verifica el acontecimiento (acción) a que está asociada, en ese caso se dispara o franquea. El disparo o franqueo de una transición consiste en :

- Se quita una marca de cada lugar de entrada de esa transición.
- Se añade una marca a cada lugar de salida de esa transición.

La primera transición se denomina transición fuente y no tiene ningún lugar de entrada. La última es una transición sumidero y no tiene ningún lugar de salida.

Por ejemplo:

Suponiendo que el ejemplo anterior se refiere a una barbería:

- p1: un cliente espera.
- p2: el barbero espera.
- p3: un cliente está siendo atendido y el barbero trabaja.
- t1: el barbero empieza un afeitado.

Si la situación inicial es la anterior es que un cliente y el barbero están esperando. Entonces t1 está validada y se disparará cuando el barbero comience. En ese instante, se quita una marca de p1 y de p2 y se pone otra a p3: se cumple así la postcondición.

Ahora vamos a suponer que tenemos varios clientes:

- p1: cliente espera.
- p2: barbero inactivo.
- p3: cliente está siendo servido.
- p4: cliente atendido.
- t1: entra cliente.

- t2: barbero empieza.
- t3: barbero termina.
- t4: sale cliente.

Cuando entra un cliente, espera. Si hay un cliente esperando y el barbero está inactivo, el barbero empieza. Cuando acaba, el cliente está atendido y puede salir.

Comportamientos modelables con las redes de Petri.

Con una red de Petri podemos modelar:

- Acciones secuenciales:

- Acciones concurrentes: t1 y t2 son transiciones que se pueden disparar de forma concurrente:

- Exclusión mutua o conflicto: t1 y t2 comparten un lugar de entrada común. Si dicho lugar de entrada estuviera marcado, t1 y t2 serían ambas franqueables, pero sólo podemos disparar una. Se produce un conflicto. Además, existe exclusión mutua entre ambas, se dispara una o la otra: