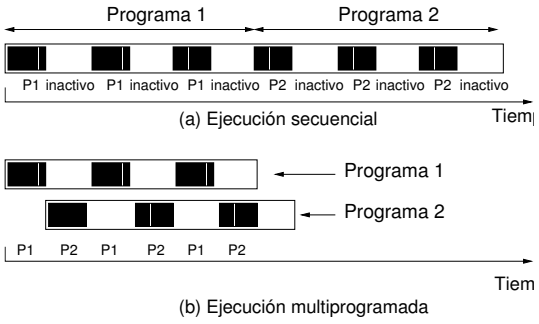
libro abierto / serie apuntes

Pablo Ruiz Múzquiz

Sistemas Operativos

0.5.0



▷ Un libro libre de Alqua

SSOO	Sistemas Operativos	004.451 ALQ

† lomo para ediciones impresas

Dedicado

A mi hermana Inés

http://alqua.org/libredoc/SSOO

Pablo Ruiz Múzquiz pablo@alqua.org http://alqua.org/people/pablo

Sistemas Operativos

versión 0.5.0
15 de abril de 2004

alqua, madeincommunity

copyleft

Serie apuntes

Área Sistemas operativos

CDU 004.451

Editores

Pablo Ruiz Múzquiz pablo@alqua.org

Notas de producción

Plantilla `latex-book-es-b.tex, v. 0.1` (C) Álvaro Tejero Cantero.

▷compuesto con software libre◁

Índice general

Portada I

Copyleft VI

Índice general VII

1. Introducción a los Sistemas Operativos 1
- 1.1. Definición de Sistema Operativo . . . 1
- 1.2. Relación con la máquina subyacente . . . 2
  - 1.2.1. Componentes básicos de la arquitectura Von Neuman . . . 2
  - 1.2.2. Registros del procesador . . . 3
  - 1.2.3. Ejecución de instrucciones . . . 4
  - 1.2.4. Interrupciones . . . 5
- 1.3. Funciones y objetivos de los Sistemas Operativos . . . 7
  - 1.3.1. El Sistema Operativo como Interfaz Usuario/Computadora . . . 7
  - 1.3.2. El Sistema OPerativo como administrador de recursos . . . 9
  - 1.3.3. Facilidad de evolución del Sistema Operativo . . . 9
- 1.4. Evolución histórica de los Sistemas Operativos . . . 10
  - 1.4.1. Proceso en serie. Primera generación (1945-1955) . . . 10
  - 1.4.2. Sistemas sencillos de proceso por lotes. Segunda generación (1955-1965) . . . 11
  - 1.4.3. Multiprogramación. Tercera Generación (1965-1980) . . . 11
  - 1.4.4. Computadoras personales. Cuarta Generación (1980-1990) . . . 13

2. Gestión de procesos 15
- 2.1. Procesos y tareas . . . 15
  - 2.1.1. División implícita y explícita de tareas . . . 15
  - 2.1.2. Tipos de procesos y relación entre procesos concurrentes . . . 16
  - 2.1.3. El sistema operativo y los procesos . . . 17
- 2.2. Creación y terminación de procesos . . . 18
- 2.3. Estados de un proceso . . . 19
  - 2.3.1. Modelo de dos estados . . . 19
  - 2.3.2. Modelo de 5 estados . . . 21
  - 2.3.3. Procesos suspendidos . . . 23
- 2.4. Estructuras de control del sistema operativo . . . 27

2.4.1. Tablas de memoria, de E/S, de archivos y de procesos 27
2.4.2. Bloque de control de procesos (BCP) 28
2.4.3. Estados del sistema y listas de procesos 30
2.4.4. Conmutación de procesos 31
2.4.5. Servicios del sistema operativo para la gestión de procesos 32

3. Planificación de procesos **35**
3.1. Concepto y criterios de planificación 35
3.1.1. Utilización del procesador: 35
3.1.2. Productividad 35
3.1.3. Tiempo de retorno 36
3.1.4. Tiempo de espera 36
3.1.5. Tiempo de respuesta 36
3.2. Tipos de planificadores 36
3.2.1. Planificador a largo plazo (PLP) 36
3.2.2. Planificador a corto plazo (PCP) 37
3.2.3. Planificador a medio plazo (PMP) 38
3.3. Algoritmos de planificación 38
3.3.1. Algoritmo First Come First Serve (FCFS) 39
3.3.2. Algoritmo por reparto circular de tiempo (RR, Round-Robin) . . . 39
3.3.3. Planificación con expropiación basada en prioridades (ED, Event-Driven) 42
3.3.4. Planificación MLQ (Multiple level queues) 43

4. Programación Concurrente **45**
4.1. Multitarea, multiprogramación y multiproceso 45
4.2. Principios de concurrencia 45
4.3. Comunicación y sincronización de procesos 46
4.3.1. Posibilidades de interacción de procesos 46
4.3.2. Necesidad de sincronización de los procesos: región crítica y exclusión mutua 47
4.4. Soluciones software para la exclusión mutua 49
4.4.1. Algoritmo de Dekker 49
4.4.2. Algoritmo de Peterson 50
4.4.3. Semáforos 56
4.4.4. Monitores 58
4.4.5. Paso de mensajes 60
4.4.6. Soluciones hardware para la exclusión mutua 63

5. Interbloqueos **73**
5.1. Principios de interbloqueo 73
5.1.1. Recursos reutilizables 73
5.1.2. Recursos consumibles 73
5.1.3. Condiciones de interbloqueo 74

5.2. Prevención de interbloqueos 74
5.3. Detección de interbloqueos 75
5.4. Predicción de interbloqueo. Algoritmo del banquero 76
5.4.1. Negativa de iniciación de procesos 77
5.4.2. Negativa de asignación de recursos 78

6. Gestión de memoria **81**
6.1. Reubicación 81
6.2. Asignación de memoria con particiones fijas 82
6.3. Asignación de memoria con particiones dinámicas 84
6.4. Asignación de memoria con paginación simple 84
6.5. Asignación de memoria con segmentación simple 87
6.6. Memoria virtual 88
6.6.1. Estructuras Hardware y de control 88
6.6.2. Hiperpaginación y cercanía de referencias 89
6.6.3. Memoria virtual con paginación y *buffer* de traducción adelantada (TLB) 90
6.6.4. Software del SO para la gestión de memoria virtual 90

Índice alfabético **95**

Historia **97**

Creative Commons Deed **99**

Manifiesto de Alqua **101**

El proyecto libros abiertos de Alqua **105**

Otros documentos libres **109**

1 Introducción a los Sistemas Operativos

1.1. Definición de Sistema Operativo

Un sistema operativo puede ser contemplado como una colección organizada de extensiones software del hardware, consistentes en rutinas de control que hacen funcionar al computador y proporcionan un entorno para la ejecución de programas. Además, estos programas utilizan las facilidades proporcionadas por el sistema operativo para obtener acceso a recursos del sistema informático como el procesador, archivos y dispositivos de entrada/salida (E/S). De esta forma, el SO constituye la base sobre la cual pueden escribirse los programas de aplicación, los cuales invocarán sus servicios por medio de *llamadas al sistema*. Por otro lado, los usuarios pueden interactuar directamente con el SO a través de órdenes concretas. En cualquier caso, el SO actúa como *interfaz* entre los usuarios/aplicaciones y el hardware de un sistema informático.

El rango y la extensión de los servicios proporcionados por un SO dependen de varios factores. Así, las funciones visibles al usuario están en gran medida determinadas por la necesidades y características del entorno objetivo que el SO está destinado a soportar. Por ejemplo, un SO destinado al desarrollo de programas en un entorno interactivo puede tener un conjunto bastante diferente de llamadas y órdenes que un sistema operativo diseñado para soporte en tiempo de ejecución a una aplicación de tiempo real dedicada, tal como el control del motor de un coche, el control del reactor de una central nuclear o el sistema de actualizaciones derivado de las operaciones de un cajero automático de una entidad bancaria.

Internamente, un SO actúa como gestor de los recursos del sistema informático tales como el procesador, la memoria, los archivos y los dispositivos de E/S. En esa función, el SO mantiene actualizada la información relativa al estado de sistemas que soportan la ejecución concurrente de programas, el SO resuelve las peticiones conflictivas de recursos de manera que preserve la integridad del sistema y al hacerlo intenta optimizar el rendimiento final.

En general, el SO presenta al usuario el equivalente de una máquina virtual que sea más fácil de utilizar que la máquina subyacente y uno de sus objetivos primarios es incrementar la productividad de los recursos que ofrece al sistema mediante una planificación lo más óptima posible de ellos.

1.2. Relación con la máquina subyacente

1.2.1. Componentes básicos de la arquitectura Von Neuman

A un nivel muy alto, un sistema informático que implemente la arquitectura Von Neuman clásica consta de 3 componentes básicos: memoria principal, unidad central de proceso y dispositivos de entrada/salida. La unidad central de proceso, a su ves, está constituida por la unidad aritmético-lógica, la unidad de control y un conjunto de registros. Los componentes básicos mencionados se encuentran interconectados para llevar a cabo la función principal del computador, que consiste en la ejecución de las sentencias que forman los procesos. Así pues, se tienen cuatro elementos estructurales principales.

- Memoria principal: Comúnmente conocida como memoria RAM. En ella se encontrará el programa en código máquina a ejecutar, los datos de entrada y los resultados. La memoria está formada por un conjunto de celdas idénticas que pueden ser accedidas de forma aleatoria mediante los distintos registros de direccionamiento. Esta memoria es normalmente volátil y también se conoce como *memoria real*.
- La unidad aritmético-lógica permite efectuar un conjunto de opoeraciones aritméticas y lógicas de los datos. Estos datos, que pueden proceder de memoria principal o ser el resultado de operaciones previas, se almacenarán en los registros de entrada de que esta unidad dispone. El resultado de la operación, así como la información relativa al estado de terminación de la misma, quedarán almacenados en los correspondientes registros.
- La unidad de control es la que se encarga de hacer funcionar al conjunto, para lo cual lleva a cabo las siguientes funciones:
  - Lee de memoria las instrucciones máquina que forman el programa.
  - Interpreta cada instrucción leída.
  - Lee los datos de memoria referenciados por la instrucción.
  - Ejecuta la instrucción.
  - Almacena el resultado de cada instrucción.
- Finalmente, la unidad de Entrada/Salida se encarga de realizar la transferencia de información entre la memoria (o los registros) y los periféricos. La E/S se puede efectuar bajo el gobierno de la unidad de control (E/S programada) o de forma independiente (DMA). El transporte de datos se realiza, pues, entre el computador y su entorno exterior. Dicho entorno consta de una gran variedad de dispositivos externos que incluye a los dispositivos de memoria secundaria, los equipos de comunicación y los terminales.

Además de los descritos anteriormente, los sistemas informáticos disponene de un conjunto de elementos de interconexión (*buses de datos*). Estos elementos están constituidos

por mecanismos que permiten la comunicación entre los componentes que conforman el sistema informático, es decir, su función es la de interconectar procesadores, memoria principal y módulos de E/S.

1.2.2. Registros del procesador

Dentro del procesador hay un conjunto de registros que ofrecen un nivel de memoria que es más rápido y pequeño que la memoria principal. Los registros del procesador pueden clasificarse de la siguiente forma:

- Registros visibles de usuario: Un registro visible al usuario es aquél que puede ser referenciado por medio de lenguaje máquina que ejecuta el procesador, siendo, por lo general, accesible a todos los programas, tanto de aplicación como de sistema. Un programador de lenguaje de máquina o ensamblador puede minimizar las referencias a memoria principal mediante un uso óptimo de estos registros. Los tipos de registro normalmente disponibles son: registros de datos, registros de dirección y registros de códigos de condición.

  - Los registros de datos pueden ser asignados por el programador a diversas funciones. En muchos casos, son de propósito general y pueden ser empleados por instrucciones que lleven a cabo cualquier tipo de operación sobre un dato determinado. Sin embargo, suelen establecerse ciertas restricciones como dedicar algunos registros para operaciones en coma flotante.
  - Los registros de dirección guardan direcciones de memoria principal que pueden contener datos, instrucciones o parte de una dirección efectiva utilizada para calcularla. Como ejemplo de registros de dirección podemos incluir los siguientes:
    - Registro índice: Se utiliza en el direccionamiento indexado que consiste en sumar un índice a un valor base para obtener la dirección efectiva.
    - Puntero de segmento: Cuando se utiliza direccionamiento segmentado la memoria se divide en segmentos, que son bloques de palabras de tamaño variable. Una referencia a memoria consta de un segmento particular y un desplazamiento dentro del segmento.
    - Puntero de pila: Si existe un direccionamiento de pila visible para los usuarios, la pila estará, por lo general, en memoria principal, existiendo un registro dedicado a señalar la cima de la pila, para poder sacar (pop) e introducir (push) elementos en ella.
  - Una última categoría de registros que son, al menos, parcialmente visibles para los usuarios, son aquéllos que contienen **códigos de condición** (también denominados *indicadores* o *flags*). Los códigos de condición son bits activados por el hardware como resultado de determinadas operaciones. Por ejemplo, una operación aritmética puede producir un resultado positivo, negativo, cero, con acarreo o con desbordamiento, lo que se reflejará en el correspondiente bit

o flag de control y que puede consultarse posteriormente, por ejemplo, como parte de una operación de salto condicional. Es importante tener en cuenta que los códigos de condición pueden ser consultados por las aplicaciones de usuario pero no modificados por éstas. Los bits de código de condición se sgrupan en uno o más registros que forman parte de los determinados registros o palabras de control que veremos a continuación.

En algunas máquinas, una llamadaa un procedimiento o subrutina provocará que los registros visibles de usuario se salven automáticamente, para luego restaurarlos y continuar con la ejecución normal. Este proceso de salvar y restaurar lo lleva a cabo el procesador como parte de las instrucciones de llamada y retorno. Esto permite que cada procedimiento pueda utilizar los registros de forma independiente. En otras máquinas, sin embargo, es responsabilidad del programador salvar los registros que considere relevantes antes de efectuar una llamada a un procedimiento.

- Los registros de control y estado son utilizados por el procesador para el control de las operaciones o por rutinas privilegiadas del sistema operativo para controlar la ejecución de los programas. En la mayoría de las máquinas, la mayor parte de estos registros no son visibles para los usuarios. Además de los registros MAR, MBR, IOAR, IOBR, que se utilizan en funciones de direccionamiento e intercambio de datos entre la unidad procesadora y la memoria principal y los dispositivos de E/S, la unidad de control tiene asociados una serie de registros, entre los que cabe destacar el contador de programa (PC, *program counter*), que indica la dirección de la siguiente instrucción máquina a ejecutar, el puntero de pila (SP, *stack pointer*), que sirve para manejar la pila del sistema en memoria principal, el registro de instrucción (RI) que permite almacenar la instrucción máquina a ejecutar, y el registro de estado (SR) o palabra de estado del programa (PSW, *program status word*) que almacena junto con el contador de programa (PC) diversa información producida por la última instrucción del programa ejecutada (bits de estado aritmético) e información sobre la forma en que ha de comportarse la computadora (bits de nivel de ejecución que determinan el modo en que la computadora ejecuta las instrucciones, usuario o sistema o supervisor, y bits de control de interrupciones que establecen las instrucciones que se pueden aceptar).

1.2.3. Ejecución de instrucciones

La tarea básica que realiza un computador es la ejecución de instrucciones. El punto de vista más sencillo es considerar que el procesamiento de instrucciones consiste en una sencuencia sencilla que se repite a alta velocidad (cientos de millones de veces por segundo). Esta secuencia consiste en 3 pasos: lectura de memoria de la instrucción máquina apuntada por el PC, incremento del contador del programa - para que apunte a la siguiente instrucción máquina - y ejecución de la instrucción.

Esta secuencia tiene 2 prioridades fundamentales: es lineal, es decir, ejecuta de forma consecutiva las instrucciones que están en direcciones consecutivas, y forma un bucle

infinito. Esto significa que la unidad de control de la computadora está continua e ininterrumpidamente realizando esta secuencia.

Podemos decir, por tanto, que lo único que sabe hacer la computadora es repetir a gran velocidad esta secuencia. Esto quiere decir, que para que realice algo útil, se ha de tener cargados en memoria un programa máquina con sus datos y hemos de conseguir que el contador de program apunte a la instrucción máquina inicial del programa.

El esquema de ejecución lineal es muy limitado, por lo que se añaden unos mecanismos que permiten alterar esta ejecución lineal. En esencia, todos ellos se basan en algo muy simple; modifican el contenido del programa, con lo que se consigue que se salte o bifurque a otro segmento del programa o a otro programa (que, lógicamente, también ha de residir en memoria). Los tres mecanismos básicos de ruptura de secuencia son los siguientes.

- Las instrucciones máquina de salto o bifurcación, que permiten que el programa rompa su secuencia lineal de ejecución pasando a otro fragmento de sí mismo.
- Las interrupciones externas o internas, que hacen que la unidad de control modifique el valor del contador de programa saltando a otro programa.
- La instrucción de máquina "TRAP", que produce un efecto similar a la interrupción, haciendo que se salte a otro programa.

Si desde el punto de vista de la programación son especialmente interesantes las instrucciones de salto, desde el punto de vista de los SSOO son mucho más importantes las interrupciones y las interrupciones de TRAP. Por tanto, centraremos nuestro interés en resaltar los aspectos fundamentales de estos dos mecanismos.

1.2.4. Interrupciones

Casi todos los computadores tienen un mecanismo mediante el cual otros módulos (E/S, memoria) pueden interrumpir la ejecución normal del procesador. Las interrupciones aparecen, principalmente, como una vía para mejorar la eficiencia del procesamiento debido a que la mayoría de los dispositivos externos son mucho más lentos que el procesador.

Con las interrupciones, el procesador se puede dedicar a la ejecución de otras instrucciones mientras una operación de E/S está en proceso. Cuando el dispositivo de E/S esté disponible, es decir, cuando esté preparado para aceptar más datos del procesador, el módulo de E/S de dicho dispositivo enviará una señal de *solicitud de interrupción* al procesador. El procesador responde suspendiendo la operación del programa en curso y saltando a un programa que da servicio al dispositivo de E/S en particular, conocido como *rutina de tratamiento de interrupciones (Interrupt handler)*, reanudando la ejecución original después de haber atendido al dispositivo.

Desde el punto de vista del programa de usuario una interrupción es solamente eso: una interrupción de la secuencia normal de ejecución. Cuando el tratamiento de la interrupción termina, la ejecución continúa. El programa no tiene que disponer de ningún código especial para dar cabida a las interrupciones; el procesador y el sistema operativo

son los responsables de suspender el pograma de usuario y de reanudarlo después en el mismo punto.

Para dar cabida a las interrupciones, se añade un ciclo de interrupción al ciclo de instrucción. En el ciclo de interrupción el procesador comprueba si ha ocurrido alguna interrupción, lo que se indicará con la presencia de alguna señal de interrupción. Si no hay interrupciones pendientes, el procesador sigue con el ciclo de lectura y trae la próxima instrucción del programa en curso. Si hay una interrupción pendiente, el programador suspende la ejecución del programa en curso y ejecuta una rutina de tratamiento de interrupción. Esta rutina, generalmente, forma parte del sistema operativo, determina la naturaleza de la interrupción y realiza cuantas acciones sean necesarias. Una interrupción desencadena una serie de sucesos tanto en el hardware del procesador como en el software. Estos sucesos pueden secuenciarse de la siguiente forma:

1. El dispositivo emite una señal de interrupción al procesador.

2. El procesador finaliza la ejecución de la instrucción en curso antes de responder a la interrupción.

3. El procesador pregunta por la interrupción comprueba que hay una y envía una señal de reconocimiento al dispositivo que generó la interrupción, de forma que éste suspende su señal de interrupción.

4. El procesador necesita ahora prepararse para transferor el control a la rutina de interrupción. Para empezar, hace falta salvar la información necesaria para reanudar la ejecución del programa en curso en el punto de la la interrupción. La mínima información es la palabra de estado del programa (PSW) y la ubicación de la próxima instrucción a ejecutar.

5. El procesador carga ahora el contador de programa con la ubicación de entrada del programa de tratamiento de interrupción.

6. La rutina de interrupción suele comenzar salvando el contenido de los registros del procesador porque la rutina puede utilizarlos y se perdería la información que la ejecución del proceso interrumpido hibiera dejado en ellos.

7. La rutina de tratamiento de la interrupción procesa la interrupción realizando las acciones requeridas para atenderla.

8. Tras completar el tratamiento de la interrupción, se recuperan de la pila los valores de los registros que se salvaron y se restauran en los correspondientes registros.

9. El paso final es restaurar los valores del la PSW y del contador de programa también a partir de la pila y continuar con la ejecución del programa interrumpido. Estas acciones se llevan a cabo como resultado de ejecutar la instrucción máquina para retorno de interrupción, RETI, que incluyen la mayoría de las computadoras en su lenguaje máquina.

Las computadoras incluyen varias señales de solicitud de interrupción, cada una de las cuales tiene una determinada prioridad. En caso de activarse al tiempo varias de estas señales, se tratará la de mayor prioridad, quedando las demás a la espera de ser atendidas. Además la computadora incluye un mencanismo de **inhibición** selectiva que permite detener todas o determinadas señales de interrupción. Las señales inhibidas no son atendidas hasta que pasen a estar desinhibidas. La información de inhibición de las interrupciones suele incluirse en la parte del registro de estado que solamente es modificable en nivel de núcleo, por lo que su modificación queda restringida al sistema operativo.

Las interrupciones se pueden generar por diversas causas, que se pueden clasificar de la siguiente forma:

- Excepciones de programa. Hay determinadas causas que hacen que un programa presente un problema en su ejecución, por lo que deberá generarse una interrupción, de forma que el SO trate dicha causa. Ejemplos de errores de este tipo son el desbordamiento de operaciones matemáticas, al división entre cero, el intento de acceso a una zona de memoria no permitida, etc.
- Interrupciones de reloj.
- Interrupciones de E/S. Los controladores de los dispositivos de E/S necesitan interrumpir para indicar que han terminado una operación o un conjunto de ellas.
- Excepciones del hardware como la detección de un error de paridad en la memoria.
- Instrucciones de TRAP. Estas instrucciones permiten que un programa genere una interrupción y se utilizan fundamentalmente para solicitar los servicios del SO.

1.3. Funciones y objetivos de los Sistemas Operativos

Como ya se ha visto, un sistema operativo actúa como interfaz entre la máquina desnuda y los programas de aplicaciones o el propio usuario. por otro lado, el sistema operativo también se encarga de gestionar los recursos del sistema informático para obtener un uso lo más óptimo posible de éstos. A continuación, trataremos las funciones del sistema operativo desde estos dos puntos de vista, así como las características que debe presentar para mantener una capacidad de evolución adecuada.

1.3.1. El Sistema Operativo como Interfaz Usuario/Computadora

Un sistema operativo debe hacer que la interacción del usuario o de los programas de aplicación con el computador resulte sencilla y fácil y debe construirse de modo que permita el desarrollo efectivo, la verificación y la introducción de nuevas funciones en el sistema y, a la vez, no interferir en los servicios ya proporcionados.

El Hardware y el Software que se utilizan para proveer al usuario de aplicaciones puede contemplarse de forma estratificada o jerárquica. Este usuario final no tiene que preocuparse de la arquitectura del computador y contempla el sistema informático en términos

de aplicaciones. Estas aplicaciones pueden construirse con un lenguaje de programación y son desarrolladas por los programadores de aplicaciones.

Si se tuviera que desarrollar un programa de aplicación con un conjunto de instrucciones totalmente responsables del control del hardware, dicho programa tendría una tarea abrumadora y compleja. Para facilitar esta tarea, se ofrecen una serie de programas de sistema. Algunos de estos programas implementan funciones muy utilizadas que ayudan a la creación de aplicaciones de usuario, la gestión de archivos y el control de los dispositivos de E/S. El programa de sistema más importante es el sistema operativo.

El sistema operativo oculta al programador los detalles del hardware y le proporciona una interfaz cómoda para utilizar el sistema y actúa como mediador, ofreciendo al programador y a los programas de aplicación un conjunto de servicios y utilidades que faciliten su tarea.

De forma resumida el sistema operativo ofrece *servicios* en las siguientes áreas:

- Creación de programas: El sistema operativo ofrece una gran variedad de servicios como los editores y depuradores (*debuggers*), para ayudar al programador en la creación de programas. Normalmente, estos servicios están en forma de programas de utilidad que no forman realmente parte del sistema operativo, pero que son accesibles a través de él.

- Ejecución de programas: Para ejecutar un programa es necesario realizar un cierto número de tareas. Las instrucciones y los datos deben cargarse en memoria principal, los archivos y los dispositivos de E/S deben inicializarse y deben prepararse otros recursos. El sistema operativo administra todas estas tareas por el usuario.

- Acceso a los dispositivos de E/S: Cada dispositivo de E/S requiere un conjunto propio y peculiar de instrucciones o señales de control para su funcionamiento. El sistema operativo, ayudado por los manejadores o *drivers* de dispositivo tiene en cuenta estos detalles de forma que el programador pueda pensar en forma de lecturas y escrituras simples desde o hacia el dispositivo.

- Acceso controlado a los archivos: El sistema operativo se ocupa del formato de los archivos y del medio de almacenamiento. En el caso de sistemas de varios usuarios trabajando simultáneamente, es el sistema operativo el que brinda los mecanismos para controlar que el acceso a los archivos se lleve a cabo de una forma correcta.

- Acceso al sistema: En el caso de un sistema compartido o público, el sistema operativo controla el acceso al sistema como un todo y a los recursos específicos del sistema. Las funciones de acceso deben brindar proteción a los recursos y a los datos ante usuarios no autorizados y debe resolver conflictos en la propiedad de los recursos.

- Detección y respuesta a errores: Cuando un sistema informático está en funcionamiento pueden producirse varios errores. El sistema operativo debe dar una respuesta que elimine la condición de error con el menor impacto posible sobre las aplicaciones que están en ejecución.

- Contabilidad: Un sistema operativo debe recoger estadísticas de utilización de los diversos recursos y supervisar parámetros de rendimiento tales como el tiempo de respuesta.

1.3.2. El Sistema OPerativo como administrador de recursos

Un SO debe perseguir una utilización lo más óptima y equilibrada posible de los recursos que administra. De esta forma se obtendrá un alto rendimiento del sistema informático gobernado.

El SO es el responsable de la gestión de los recursos de la máquina y mediante su administración tiene el control sobre las funciones básicas de la misma. El SO no es nada más que un programa pero la diferencia clave es su propósito. El SO dirige al procesador en el empleo de otros recursos del sistema y en el control del tiempo de ejecución de los programas de usuario.

Una parte del SO reside en memoria principal. En esta parte está el núcleo (*kernel*) que incluye funciones del SO utilizdas con más frecuencia aunque, en un momento dado, puede incluir otras partes en uso. El resto de la memoria, que contiene datos y programas de usuario, es administrada conjuntamente por el SO y por el hardware de control de memoria. El SO decide cuándo puede utilizarse un dispositivo de E/S por parte de un programa en ejecución y controla el acceso y la utilización de los archivos. El procesador es, en sí mismo, un recurso y es el SO el que debe determinar cuánto tiempo de procesador debe dedicarse a la ejecución de un programa usuario en particular. En el caso de sistemas multiprocesador, la decisión debe tomarse entre todos los procesadores.

1.3.3. Facilidad de evolución del Sistema Operativo

Un SO importante evolucionará en el tiempo por una serie de razones:

- Actualizaciones del hardware y nuevos tipos de hardware: Las mejoras introducidas en los componentes hardware del computador deben tener soporte en el sistema operativo. Un ejemplo es el aprovechamiento del sistema operativo del hardware de paginación que acompaña a la memoria de algunos sistemas informáticos.

- Nuevos servicios: Como respuesta a nuevas necesidades, el sistema operativo ampliará su oferta de servicios para añadir nuevas medidas y herramientas de control.

- Correcciones: El sistema operativo tiene fallos que se descubrirán con el curso del tiempo y que es necesario corregir.

La necesidad de hacer cambios en un SO de forma regular introduce ciertos requisitos en el diseño. Una afirmación obvia es que el sistema debe tener una construcción modular, con interfaces bien definidas entre los módulos y debe estar bien documentado.

1.4. Evolución histórica de los Sistemas Operativos

Para intentar comprender los requisitos básicos de un SO y el significado de las características principales de un sistema operativo contemporáneo, resulta útil considerar cómo han evolucionado los sistemas operativos a los largo de los años.

1.4.1. Proceso en serie. Primera generación (1945-1955)

En los primeros computadores, de finales de los 40 hasta mediados de los 50, el programa interactuaba directamente con el hardware: no había sistema operativo. La operación con estas máquinas se efectuaba desde una consola dotada con indicadores luminosos y conmutadores o a través de un teclado hexadecimal. Los programas se arrancan cargando el registro contador de programas con la dirección de la primera instrucción. Si se detenía el programa por un error, la condición de error se indicaba mediante los indicadores luminosos. El programador podía examinar los registros relevantes y la memoria principal para comprobar el resultado de la ejecución o para determinar la causa del error.

El siguiente paso significativo de la evolución en el uso de sistemas informáticos vino con la llegada de dispositivos de E/S tales como tarjetas perforadas y cintas de papel y con los traductores de lenguajes. Los programas. codificados ahora en un lenguaje de programación, se traducen a un forato ejecutable mediante un programa como un compilador o un intérprete. Otro programa, llamado *cargador*, automatiza el proceso de cargar en memoria estos programas en código ejecutable. El usuario coloca un programa y sus datos de entrada en un dispositivo de entrada y el cargador transfiere la información desde el dispositivo a la memoria. Después de transferir el control al programa cargado por medios manuales o automáticos, comienza la ejecución del mismo. El programa en ejecución lee sus datos desde el dispositivo de entrada asignado y puede producir ciertos resultados en un dispositivo de salida tal como una impresora o la pantalla.

Estos primeros sistemas presentaban dos problemas principales:

- Planificación: La mayoría de las instalaciones empleaban un formulario de reserva de tiempo de máquina. Normalmente, un usuario podía reservar bloques de tiempo, múltiplos, por ejemplo, de media hora. Si la ejecución del programa terminaba antes del plazo asignado, el tiempo restante se desperdiciaba. También podía suceder que el programa no terminara dentro del plazo asignado, con lo que el programadir no podía saber si el programa había terminado satisfactoriamente o no.
- Tiempo de preparación: Un programa aun siendo sencillo requería un tiempo de preparación bastante grande ya que en primer lugar se cargaba un compilador y un programa en lenguaje de alto nivel (programa fuente) en la memoria. A continuación, se salvaba el programa ya compilado (programa objeto) y, por último, se montaba y cargaba este programa objeto junto con las funciones comunes.

Este modo de trabajo podía denominarse *proceso en serie* porque refleja el hecho de los que usuarios tenían que acceder al computador en serie.

1.4.2. Sistemas sencillos de proceso por lotes. Segunda generación (1955-1965)

Las primeras máquinas eran muy caras y, por tanto, era importante maximizar la utilización de las mismas. El tiempo desperdiciado en la planificación y la preparación era inaceptable.

Para mejorar el uso, se desarrolló el concepto de sistema operativo por lotes (*batch*). El primer sistema operativo por lotes fue desarrollado a mediados de los 50 por la General Motoros para usar en un IBM 701.

La idea central que está detrás del esquema sencillo de proceso por lotes es el uso de un elemento de software conocido como **monitor**. Con el uso de esta clase de sistema operativo, los usuarios ya no tenían acceso directo a la máquina. En su lugar, el usuario debía entregar los trabajos en tarjetas o en cinta al operador del computador, quien agrupaba secuencialmente los trabajos por lotes y ubicaba los lotes enteros en un dispositivo de entrada para su empleo por parte del monitor. Cada programa se construía de modo tal que volviera al monitor al terminar el procesamiento y, en ese momento, el monitor comenzaba a cargar automáticamente el siguiente programa.

Para obtener provecho del potencial de utilización de recursos, un lote de trabajos debe ejecutarse automáticamente sin intervención humana. Para este fin, deben proporcionarse algunos medios que instruyan al sistema operativo sobre cómo debe tratar cada trabajo individual. Estas intrucciones son suministradas generalmente por medio de órdenes del sistema operativo incorporadas al flujo de lotes. Las órdenes del sistema operativo son sentencias escritas en un Lenguaje de Control de Trabajos (*JCL, Job Control Language*). Entre las órdenes típicas de un JCL se incluyen las marcas de comienzo y finalización de un trabajo, las órdenes para cargar y ejecutar programas y las órdenes que anuncian necesidades de recursos tales como el tiempo esperado de ejecución y los requisitos de memoria. Estas órdenes se hallan incoporadas al flujo de los trabajos, junto a los programas y a los datos del usuario.

Una parte residente en memoria del SO por lotes, a veces llamado *monitor de lotes*, lee, interpreta y ejecuta estas órdenes. En respuesta a ellas, los trabajos del lote se ejecutan de uno en uno.

1.4.3. Multiprogramación. Tercera Generación (1965-1980)

Incluso con las mejoras anteriores, el proceso por lotes dedica los recursos del sistema informático a una única tarea a la vez.

En el curso de su ejecucicón, la mayoría de los programas oscilan entre fases intensivas de cálculo y fases intensivas de operaciones de E/S. Esto queda indicado en la figura 1.1 donde los periodos de cálculo intensivo se indican mediante cuadros sombreados y las operaciones de E/S mediante zonas en blanco. El problema es que los dispositivos de E/S son muy lentos comparados con el procesador. El procesador gasta parte del tiempo ejecutando hasta que encuentra una instrucción de E/S. Entonces debe esperar a que concluya la instrucción de E/S antes de continuar.

Esta ineficiencia no es necesaria. Se sabe que hay memoria suficiente para almacenar

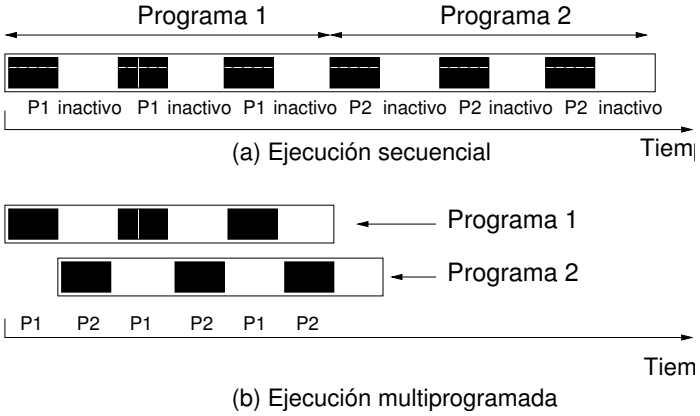


Figura 1.1: Ejecución secuencial vs multiprogramación

en memoria el sistema operativo (monitor residente) y un programa usuario. Supongamos que hay espacio suficiente para almacenar el sistema operativo y dos programas de usuario. Ahora, cuando un trabajo necesite esperar por una operación de E/S, el procesador puede cambiar a otro trabajo que esté listo para ser ejecutado. Si ampliamos la memoria para almacenar varios programas, podremos conmutar entre todos de forma que el procesador permanezca ocupado el mayor tiempo posible, evitando así el desperdicio de tiempo que suponen las esperas hasta que se completen las operaciones de E/S. Este concepto es conocido como **multiprogramación** o **multitarea** y es el punto central de los sistemas operativos modernos.

Como sugiere la figura 1.1 se pueden lograr ganancias significativas de rendimiento intercalando la ejecución de los programas, o, *multiprigramando*, que es como se le denomina a este modo de operación. Con un solo procesador no es posible la ejecución paralela de programas, y como máximo, sólo un programa puede tener el control del procesador en un instante determinado. El ejemplo presentado en la figura 1.1 (b) consigue un 100 % de utilización del procesador sólo con dos programas activos. Aunque conveniente para ilustrar la idea básica en la multiprogramación, no se deben esperar resultados tan espectaculares en programas reales, ya que las distribuciones de las fases de computación y E/S tienden a ser más variables. Para aumentar la utilización de recursos, los sistemas de multiprogramación reales permiten generalmente que más de dos programas compitan por los recursos del sistema al mismo tiempo. El número de programas en competencia activa por los recursos de un sistema informático se denomina *grado de multiprogramación*. En principio, mayores grados de multiprogramación deben dar lugar a una mayor utilización de recursos.

La multiprogramación ha sido tradicionalmente empleada para aumentar la utilización de los recursos de un sistema informático y para soportar múltiples usuarios simultáneamente activos.

1.4.4. Computadoras personales. Cuarta Generación (1980-1990)

Con el desarrollo de la tecnología LCI (Large Scale Integration) de construcción de circuitos, que permitía fabricar chips con miles de transistores en un centímetro cuadrado de silicio, se inicio la era de la computadora personal. En términos de arquitectura, las computadoras personales no eran muy distintas de las minicomputadoras del tipo PDP-11, pero en términos de precio sí eran bastante distintas. Las computadoras personales más poderosas reciben el nombre genérico de *estaciones de trabajo*, pero en realidad sólo son computadoras personales grandes.

La amplia disponibilidad de poder de cómputo condujo, junto con un nivel gráfico bastante adecuado, al crecimiento de la industria de producción de software para las computadoras personales. Gran parte de este software tenía, además, la ventaja de presentar una gran amiganilidad con el usuario.

Dos sistemas operativos han dominado la escena de las computadoras personales y las estaciones de trabajo: MS-DOS de Microsoft y UNIX de AT&T. MS-DOS tuvo un amplio uso en el IBM PC y otras máquinas que incorporaban el microprocesador 8088 de Intel o alguno de sus sucesores. UNIX, su contendiente principal, dominó las computadoras que no utilizaban microprocesadores de Intel, así como las estaciones de trabajo, en particular las que poseen chips RISC de altas prestaciones.

Un interesante desarrollo que comenzó a llevarse a cabo a mediados de la década de los 80 ha sido el crecimiento de las redes de computadoras personales con sistemas operativos en red y sistemas operativos distribuidos. En un sistema operativo en red, los usuarios son conscientes de la existencia de varias computadoras y pueden conectarse con máquinas remotas. Cada máquina ejecuta su propio sistema operativo local y tiene su propio usuario. Un sistema operativo distribuido, por el contrario, presenta al usuario un conjunto de computadores independientes como si se tratara de un solo sistema. En un sistema operativo distribuido los usuarios no deben ser conscientes del lugar donde su programa va a ejecutarse o la ubicación de los archivos a los que desea acceder, esas cuestiones deben ser manejadas automáticamente y de forma eficiente por el sistema operativo.

2 Gestión de procesos

2.1. Procesos y tareas

Proceso una definición tradicional de proceso es la de instancia de un programa en ejecución. La ejecución de tal programa es indicada al SO mediante una acción u orden especializada.

El SO responde en ese punto creando un nuevo proceso. En general, esta actividad consiste en la creación e iniciliazación de estructuras de datos en el SO para monitorizar y controlar el progreso de proceso en cuestión. Una vez creado, el proceso pasará a estar activo y competirá por la utilización de recursos del sistema como el procesador y los dispositivos I/O.

Un proceso evoluciona cíclicamente entre períodos de ejecución activa y de espera por la terminación de actividades de I/O. Cuando un proceso queda inactivo por especificar una operación de I/O y quedar a la espera de que ésta se complete, el SO puede planificar la ejecución de otro proceso.

Desde este punto de vista, un proceso es una entidad individualmente planificable, que puede ser asignada al procesador y ejecutada por éste. El SO controla, pues, dinámicamente la evolución de los procesos registrando la información correspondiente a sus cambios cuando éstos se produzcan. Esta información es utilizada por el SO para sus labores de planificación y gestión sobre el conjunto de procesos que en un determinado momento pueden coexistir en el sistema informático.

De esta forma además de la plantilla estática constituida por el programa ejecutable en que se basa, un proceso posee ciertos atributos que ayudan al SO en su gestión. Los atributos de un proceso incluyen su estado actual, unidad de planificación, derechos de acceso, nivel de prioridad entre otros datos. Desde el punto de vista del usuario, un proceso no es más que la ejecución de un conjunto de instrucciones que llevan a cabo una determinada tarea, mientras que para el SO es una entidad que atraviesa dinámicamente un conjunto de estados y que solicita los recursos del sistema que le son necesarios. De esta forma, el acceso a tales recursos debe ser planificado de forma que se consiga un rendimiento en la utilización de los mismos lo más óptimo posible.

2.1.1. División implícita y explícita de tareas

Dependiendo del SO y del entorno objetivo de ejecución de programas, la división de un trabajo en tareas que serán ejecutadas como procesos independientes así como la asignación inicial de los atributos de esos procesos pueden ser efectuadas o bien por el SO o bien por el desarrollador de la aplicación. En otras palabras, lo que constituirá un proceso independiente puede provenir de:

1. División implícita de tareas definida por el sistema.

2. División explícita de tareas definida por el desarrollador.

En general, la división implícita de tareas se aplica en sistemas operativos multitarea para multiplexar la ejecución de una serie de programas y explotar los beneficios de la concurrencia entre una serie de aplicaciones. La división explícita en tareas permite mejoras adicionales en el rendimiento al explotar la concurrencia inherente o propia de una determinada aplicación o programa. La división implícita en tareas significa que los procesos son definidos por el sistema, esta división aparece comúnmente en sistemas de multiprogramación de propósito general tales como los sistemas de tiempo compartido. En este enfoque cada programa remitido para su ejecución es tratado por el SO como un proceso independiente.

El SO asignará valores iniciales a lo atributos del proceso tales como la prioridad de planificación y los derechos de acceso en el momento de la creación del proceso basándose en el perfil del usuario y en valores predeterminados del sistema.

La división explícita significa que los desarrolladores definen explícitamente cada proceso y alguno de sus atributos, típicamente una única aplicación queda dividida en varios procesos relacionados con objeto de mejorar su rendimiento. La división explícita se utiliza en situaciones donde se desea elevar la eficiencia o controlar explícitamente las actividades del sistema.

Entre las razones más comunes para aplicar la división explícita de tareas se incluyen

1. Ganancia de velocidad: algunas de las tareas independientes en que quede dividida la aplicación podrán ejecutarse de forma concurrente con la consiguiente mejora en el tiempo total de ejecución de la aplicación.

2. Mejora en el rendimiento de la utilización de dispositivos de I/O con latencia: dedicando tareas explícitas a la realización de operaciones I/O, éstas podrán ser planificadas por el SO de forma concurrente con tareas de computación intensiva con la consiguiente ganancia en el rendimiento.

3. Multiprocesamiento: los procesos independientes que constituyen una aplicación pueden ser perfectamente portados para su ejecución en un entorno multiprocesador con lo que se conseguiría paralelismo real.

4. Computación distribuida: de igual forma, cada tarea independiente puede ser asignada a un procesador que forme parte de un sistema distribuido, siendo necesaria una sincronización con el resto de procesadores que se ocupan de sus respectivas tareas.

2.1.2. Tipos de procesos y relación entre procesos concurrentes

En principio, podemos realizar una clasificación muy general de los procesos entre procesos de usuario y procesos de sistema. Un proceso de usuario es aquel creado por

el SO como respuesta a una de ejecución del usuario o de una aplicación que ejecuta a instancias de éste.

Un proceso de sistema es un proceso que forma parte del propio SO y que desempeña alguna de sus labores características, como por ejemplo, la elección del siguiente proceso a ejecutar o bien la prestación de un servicio determinado como el acceso a un recurso de I/O del sistema. Cualquiera que sea la naturaleza de los procesos, éstos pueden mantener fundamentalmente dos tipos de relaciones: Competición y/o Colaboración.

En virtud de la compartición de recursos de un solo sistema, todos los procesos concurrentes compiten unos con otros por la asignación de los recursos del sistema necesarios para sus operaciones respectivas. Además, una colección de procesos relacionados que representen colectivamente una sola aplicación lógica, suelen cooperar entre sí. La cooperación es habitual entre los procesos creados como resultado de una división explícita de tareas. Los procesos cooperativos intercambian datos y señales de sincronización necesarios para estructurar su progreso colectivo. Tanto la competición como la colaboración de procesos requiere una cuidada asignación y protección de los recursos en términos de aislamiento de los diferentes espacios de direcciones. La cooperación depende de la existencia de mecanismos para la utilización controlada de los datos compartidos y el intercambio de señales de sincronización.

Los procesos cooperativos comparten típicamente algunos recursos y atributos además de interactuar unos con otros. Por esta razón, con frecuencia se agrupan en lo que se denomina una familia de procesos. Aunque dentro de una familia son posibles relaciones complejas entre sus procesos miembro, la relación más comúnmente soportada por los sistemas operativos es la relación padre-hijo. Esta relación se establece cuando el SO crea un nuevo proceso a instancias de otro ya existente. Los procesos hijos heredan generalmente los atributos de sus procesos padres en el momento de su creación y también pueden compartir recursos con sus procesos hermanos.

2.1.3. El sistema operativo y los procesos

Todos los SSOO de multiprogramación están construidos en torno al concepto de proceso. Los requisitos principales que debe cumplir un SO para con los procesos son los siguientes:

1. El SO debe intercalar la ejecución de procesos para optimizar la utilización del procesador ofreciendo a la vez un tiempo de respuesta razonable.

2. El SO debe asignar los recursos del sistema a los procesos en conformidad con una política específica que evite situaciones de interbloqueo.

3. El SO podría tener que dar soporte a la comunicación entre procesos y ofrecer mecanismos para su creación, labores que pueden ser de ayuda en la estructuración de aplicaciones.

2.2. Creación y terminación de procesos

Cuando se añade un proceso a los que ya está admisnitrando el SO, hay que construir las estructuras de datos que se utilizan para gestionar y controlar el proceso y asignar el espacio de direcciones que va a utilizar dicho proceso. Estas acciones constituyen la creación de un nuevo proceso.

Son cuatro los sucesos comunes que llevan a la creación de un proceso.

1. Nueva tarea en un sistema de proceso por lotes.
2. Nueva conexión interactiva.
3. Nuevo proceso creado por el SO para dar un servicio.
4. Un proceso generado por otro ya existente.

Por otro lado, en cualquier sistema informático debe existir alguna forma de que un proceso indique su terminación. Un trabajo por lotes debe incluir una instrucción de detención **halt** o una llamada explícita a un servicio del sistema operativo para indicar su terminación, mientras que en una aplicación interactiva, será la acción del usuario la que indique cuándo termine el proceso.

Todas estas acciones provocan al final una petición de servicio al SO para terminar el proceso demandante. Además, una serie de errores pueden llevarnos a la terminación de un proceso. A continuación se enumeran algunas de las condiciones más habituales de terminación de procesos:

1. Terminación normal: Un proceso termina de ejecutar su conjunto de instrucciones y finaliza.
2. Tiempo límite excedido: El proceso requiere más tiempo para completar su ejecución del que el sistema establece como máximo.
3. No disponibilidad de memoria: Tiene lugar cuando un proceso necesita más memoria de la que el sistema puede proporcionar.
4. Violación de límites: Ocurre cuando un proceso trata de acceder a una posición de memoria a la que no puede hacerlo.
5. Error de protección: Se produce si un proceso intenta utilizar un recurso o un archivo para el que no tiene permiso o trata de utilizarlo de forma incorrecta.
6. Error aritmético: Aparece si el proceso intenta hacer un cálculo prohibido como la división por cero o trata de almacenar un número mayor del que el hardware acepta.
7. Superación del tiempo máximo de espera por un recurso: En este caso, el proceso se encuentra a la espera de obtener un recurso o de que tenga lugar un determinado evento durante un tiempo que alcanza el límite establecido.

8. Fallo de dispositivo I/O: Se produce por un error en la entrada o la salida tal como la incapacidad de encontrar un archivo o la ocurrencia de un fallo de lectura o escritura después de un número máximo de intentos.

9. Instrucción no válida: Se produce si un proceso intenta ejecutar una instrucción inexistente.

10. Inento de acceso a una instrucción privilegiada: Se presenta si un proceso intenta utilizar una instrucción reservada para el SO.

11. Mal uso de los datos: Un elemento de dato no está inicializado o es de un tipo equivocado para la operación que se pretende realizar.

12. Intervención del operador o del SO: Por alguna razón, el operador o el SO termina con el proceso. Por ejemplo, si se considera comprometido el rendimiento del sistema.

13. Finalización del proceso padre: Cuando un proceso padre finaliza el SO puede diseñarse para terminar automáticamente con todos sus descendientes.

14. Solicitud del proceso padre: Un proceso padre tiene normalmente autoridad para terminar con cualquiera de sus hijos.

2.3. Estados de un proceso

En cualquier sistema operativo, es básico conocer el comportamiento que exhibirán los distintos procesos y el conjunto de estados que pueden atravesar.

2.3.1. Modelo de dos estados

El modelo más sencillo que puede construirse tiene en cuenta que un momento dado un proceso puede estar ejecutándose en el procesador o no. Así pues, un proceso puede estar en uno de dos estados: Ejecución o No ejecución (Véase la figura 2.1).

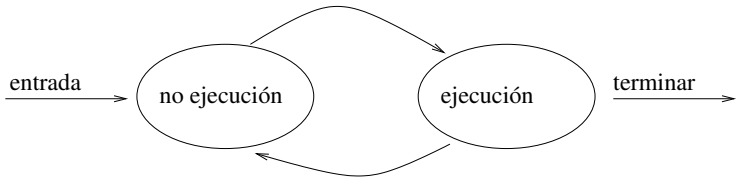


Figura 2.1: Esquema de un diagrama de dos estados

Cuando el SO crea un nuevo proceso, éste entra en el sistema en el estado de *No ejecución*. De este modo, el proceso existe, es conocido por el SO y está esperando la oportunidad de ejecutarse. En un momento dado, el sistema operativo decide otorgar

el procesador a un proceso determinado con lo que dicho proceso pasará de estado *No ejecución* a *Ejecución*.

Cada cierto tiempo, el proceso en ejecución es interrumpido y el sistema operativo seleccionará un nuevo proceso para que tome el control del procesador. El proceso interrumpido pasa del estado de *Ejecución* al de *No ejecución* mientras que el proceso elegido realiza la transición inversa.

Incluso en este modelo tan simple, se aprecian ya algunos de los elementos importantes en el diseño de SSOO. Cada proceso debe representarse de forma que el sistema operativo tenga conocimiento de su estado actual y de su posición en memoria.

Aquellos procesos que no estén en estado de ejecución deberán almacenarse en algún tipo de estructura de datos mientras esperan que el sistema operativo les otorgue el control sobre el procesador. La siguiente figura 2.2 propone una estructura basada en una cola de procesos.

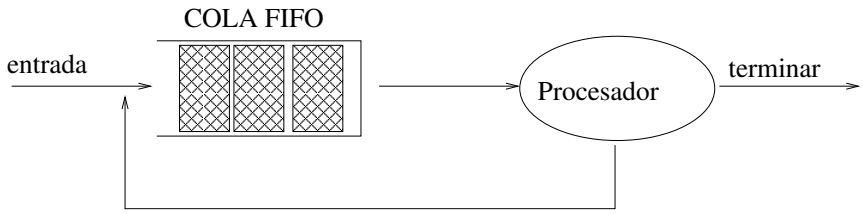


Figura 2.2: Esquema de un sistema de Cola FIFO

Dicha cola consiste en una lista enlazada de bloques en la que cada uno de estos bloques representa a un proceso. Cada bloque consistirá en un puntero a la estrcutura de datos donde el SO guarda toda la información relativa al proceso. El comportamiento del SO en este caso es similar al de un gestor de colas. Así, cada vez que el SO cree un nuevo proceso se introducirá el correspondiente bloque al final de la cola, acción que también se llevará a cabo cuando un proceso sea expropiado del procesador en favor de otro o cuando se bloquee en espera de que se complete una operación de E/S. Cuando un proceso termine su ejecución, será descartado del sistema. Si todos los procesos estuvieran siempre listos para ejecutar, la disciplina de comportamiento de cola presentada sería eficaz. El modelo de cola sigue un comportamiento FIFO y el procesador opera siguiendo un turno rotatorio con los procesos disponibles. De esta forma, a cada proceso de la cola se le otorga una cierta cantidad de tiempo para ejecutar. Si no se presentan bloqueos y transcurrido éste volverá a la cola para optar de nuevo a tener control sobre el procesador. Sin embargo, esta implementación no es adecuada debido a que algunos procesos en estado de no ejecución estarán listos para ejecutar mientras que otros se encontrarán a la espera de obtener algún recurso solicitado o a que se complete una operación de E/S. Así pues, el SO puede no entregar el procesador al proceso que se encuentre al frente de la cola. Si éste está bloqueado, tendrá que recorrer la cola buscando el primer proceso que no lo esté y que lleve más tiempo en espera.

Una forma más natural de afrontar esta situación es dividir el estado de *no ejecucución* en dos; los estados *listo* y *bloqueado*. Además se añadirán dos nuevos estados al sistema.

Estos estados son: *nuevo* y *terminado*, que resultan de utilidad para las labores de gestión de procesos. Así se dará lugar al modelo de 5 estados.

2.3.2. Modelo de 5 estados

En este modelo un proceso puede encontrarse en cualquiera de los siguiente 5 estados.

1. Estado **Nuevo**: esta estado corresponderá a procesos *que acaban de ser definidos pero que aún no han sido admitidos por el sistema operativo como procesos ejecutables*. Para estos procesos se habrán realizado ciertas tareas de gestión interna como la asignación de un identificador y la creación de algunas estructturas de control. La principal motivación para la existencia de este estado es la limitación por parte del SO del número total de procesos activos por razones de rendimiento[1] o por las restricciones impuestas por la capacidad de la memoria.

2. Estado **Listo** o **Preparado**: En este estado se encontrarán aquellos procesos que dispongan de todos los recursos necesarios para comenzar o proseguir su ejecución y se encuentran a la espera de que se les concedael control del procesador.

3. Estado de **Ejecución**: En este estado se encuentra el proceso que tiene el control del procesador. Dado que se considerarán arquitecturas que disponen de un único procesador, en un instante determinado sólo un proceso puede encontrarse en este estado.

4. Estado **Bloqueado**: En este estado se encuentran aquellos procesos que carecen de algún recurso necesario para su ejecución siendo este recurso distinto del procesador o bien se encuentran a la espera de que tenga lugar un determinado evento.

5. Estado **Terminado**: A este estado pertenecen aquellos procesos excluidos por el SO del grupo de procesos ejecutables. Un proceso alcanza este estado cuando llega al punto normal de terminación, cuando se abandona debido a un error irrecuperable o cuando un proceso con la debida autoridad hace que termine su ejecución. En este punto, el proceso ya no es susceptible de ser elegido para ejecutarse. Sin embargo, el SO conserva cierta información asociada con él para su posible utilización, bien por otras aplicaciones como programas de utilidad para el análisis de la historia y rendimiento del proceso o bien por parte del SO con fines estadísticos. Una vez extraída esta información, el SO ya no necesita mantener más datos relativos al proceso y éstos se borran del sistema.

En la figura 2.3 presentamos el diagrama de transiciones entre estados

Transición a Nuevo: Se crea un nuevo proceso para ejecutar un programa

[1] Se puede llegar a perder más tiempo en la gestión del cambio de procesos que en el proceso mismo.

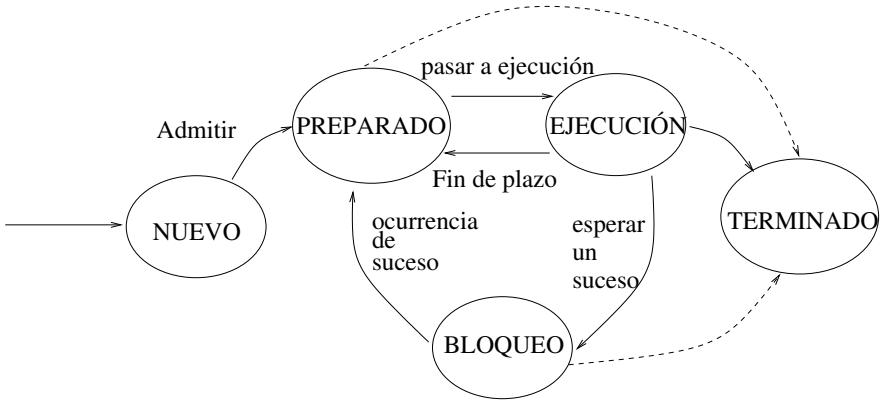


Figura 2.3: Diagrama de transiciones entre estados. La línea punteada indica situación expcecional.

Transición Nuevo-Preparado: Esta transición tiene lugar cuando el SO está preparado para aceptar o admitir un proceso más. Se tendrán en cuenta las restricciones derivadas de la capacidad de la memoria y que no haya tantos procesos activos como para degradar el rendimiento.

Transición Preparado-Ejecución: Esta transición se produce cuando el SO selecciona un nuevo proceso para ejecutar en función de su política de planificación.

Transición Ejecución-Preparado: La razón más común para esta transición es que el proceso que está en ejecución ha alcanzado el tiempo máximo permitido de ejecución ininterrumpida. Hay otras causas alternativas que no están implementadas en todos los SSOO como la expropiación de un proceso en favor de otro más prioritario. Otra situación, muy extraordinaria, que origina esta transición es que un proceso ceda voluntariamente el control del procesador.

Transición Ejecución-Bloqueo: Un proceso realiza esta transición cuando queda a la espera por la concesión de un determinado recurso o por la ocurrencia de un determinado suceso.

Transición Bloqueado-Preparado: Tiene lugar si a un proceso bloqueado se le concede el recurso solicitado u ocurre el suceso por el que estaba esperando.

Transición Preparado-Terminado: Puede ocurrir si, por ejemplo, un proceso padre decide en un momento determinado finalizar la ejecución de sus procesos hijos. Si alguno de dichos procesos se encontraba en estado **preparado** realizará esta transición. Otra razón puede ser debida a un requisito de memoria que es denegado.

Transición Bloqueado-Terminado: Un proceso hijo puede realizar esta transición por la misma razón que la anterior. Otra causa puede ser que el proceso supere el tiempo máximo de espera por un recurso y el sistema operativo decida entonces terminarlo (es la razón más habitual).

Este modelo de 5 estados puede implementarse igualmente mediante estructuras de tipo cola siguiendo un esquema como el se muestra en la figura 2.4.

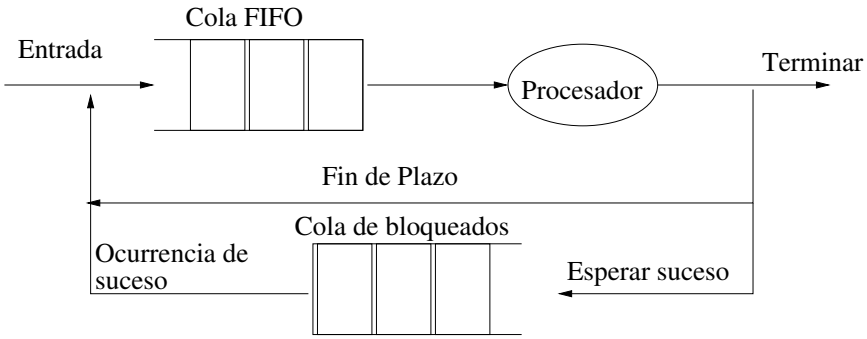


Figura 2.4: Diagrama de transiciones entre estados implementada con una cola.

Ahora se dispone de dos colas, una para los procesos en situación de **preparado** y otra para los **bloqueados**. A medida que se admiten procesos nuevos en el sistema, éstos se sitúan en la cola de **preparados**. Cuando el SO tiene que escoger un proceso para ejecutar, lo hace sacanado uno de dicha cola. En ausencia de prioridades, la referida cola puede gestionarse mediante un algoritmo FIFO. Cuando un proceso es expropiado del procesador, puede ser porque ha terminado su ejecución, porque ha excedido el tiempo máximo de posesión del procesador y entonces es devuelto a la cola de **preparados** o porque ha quedado bloqueado a la espera de un determinado suceso con lo que se introducirá en la cola de **bloquedados**. Cuando tiene lugar un determinado suceso, todos los procesos que esperaban por él son pasados desde la cola de **bloquedados** a la de **preparados**.

Esta última medida significa que cuando se produce un suceso, el SO debe recorrer toda la cola de bloqueados buscando aquellos procesos que esperen por el suceso. En un SO grande puede haber una gran cantidad de procesos en la cola de bloqueados, por tanto, resultará más eficiente disponer de un conjunto de colas, una para cada suceso. En tal caso, cuando se produzca un evento, la lista entera de procesos en la cola correspondiente a ese suceso podrá pasarse a estado **preparado**. Véase la figura 2.5.

Si la planificación de procesos se realiza mediante un esquema basado en prioridades, entonces es conveniente tener un cierto número de colas de procesos listos, una para cada prioridad.

2.3.3. Procesos suspendidos

Debido a que el procesador es mucho más rápido que los dispositivos de E/S puede ocurrir que en un momento dado todos los procesos del sistema se encuentren bloqueados a la espera de que se complete alguna operación de E/S. Para solucionar este problema existen dos opciones

1. Amplir la memoria del sistema de forma que sea posible albergar en ella más

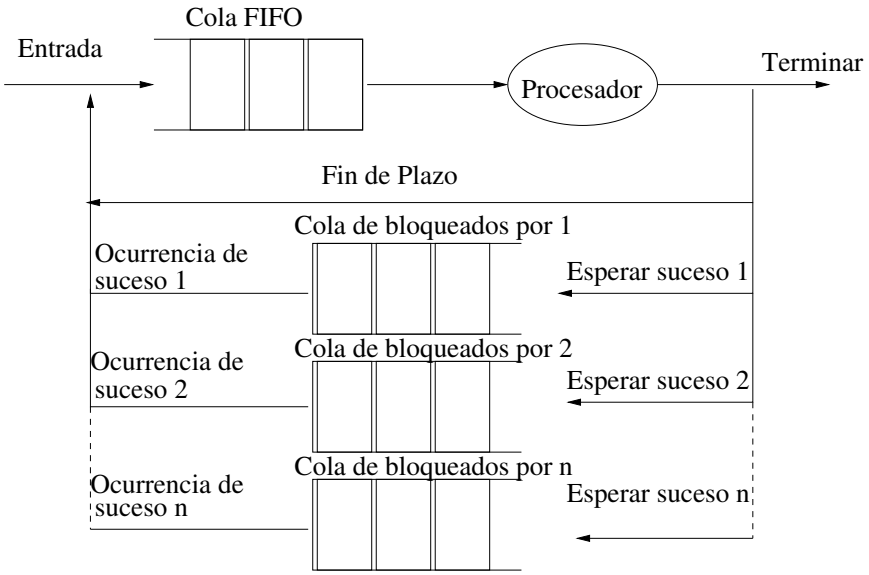


Figura 2.5: Diagrama de transiciones entre estados con varias colas, una para cada proceso.

procesos e incrementar así la posibilidad de que alguno de ellos haga uso efectivo del procesador.

2. La otra solución consiste en aplicar una técnica conocida como *intercambio* o *swaping*. Esta técnica consiste en que cuando todos los procesos que se encuentran en memoria principal están bloqueados, el SO puede sacar a uno de ellos de su correspondiente cola y transferirlo a memoria secundaria. El proceso transferido se dice entonces que queda en estado suspendido. Una vez realizada esta operación, el SO está en condiciones de traer de nuevo a memoria a un proceso previamente suspendido o bien dar entrada al sistema a un nuevo proceso.

En general, se considera suspendido a un proceso que presenta las características siguientes:

1. Un proceso suspendido no está disponible de inmediato para su ejecución.

2. Un proceso puede estar esperando o no un suceso. Si lo está, la condición de bloqueado es independiente de la condición de suspendido y el acontecimiento del suceso bloqueante no lo habilita para la ejecución.

3. El proceso fue situado en estado suspendido por un agente (el SO o el proceso padre) con el fin de impedir su ejecución.

4. El proceso no puede apartarse de estado hasta que llegue la orden expresa para ello.

Si añadimos este nuevo estado a nuestro diagrama de 5 estados, obtendremos la figura 2.6.

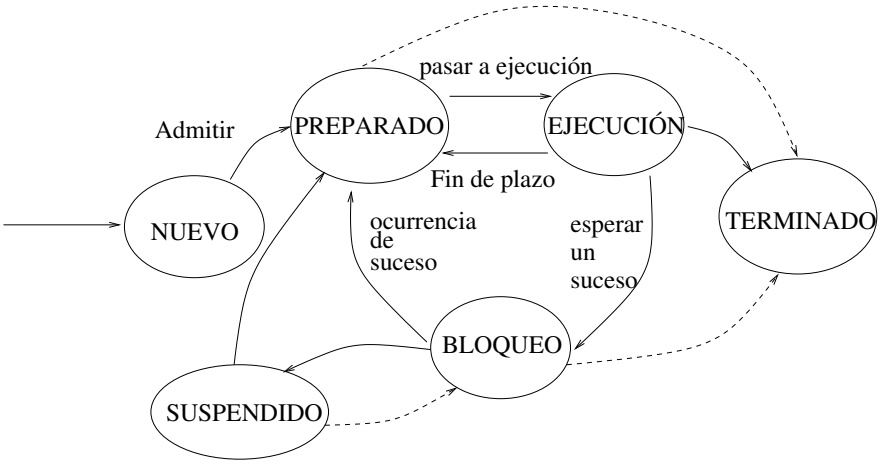


Figura 2.6: Diagrama de 5 estados + suspendido

Teniendo en cuenta que un proceso suspendido se encontraba bloqueado a la espera de que ocurriera un cierto suceso y que dicho suceso puede ocurrir mientras el proceso permanece en memoria secundaria, sería más eficiente desdoblar el estado suspendido en dos, uno para las procesos suspendidos que aún esperan el suceso que les bloqueó (estado bloqueado y suspendido) y otro para los procesos suspendidos que por haber tenido lugar se encuentran en situación de proseguir su ejecución (estado listo y suspendido). Para verlo mejor, consúletese la figura 2.7.

Las transiciones que involucran a los nuevos estados son las siguientes:

- Transición Bloqueado y Suspendido-Preparado y Suspendido: Esta transición tiene lugar si se ha producido un suceso por el que había sido bloqueado el proceso suspendido. Es importante tener en cuenta que esto requiere que esté accesible para el SO la información relativa a los procesos suspendidos.
- Transición Preparado y Suspendido-Preparado: Cuando no hay procesos preparados en memoria principal el sistema operativo tendrá que traer de memoria secundaria un proceso que pueda continuar su ejecución. Además, puede darse el caso de que el proceso en estado Preparado y Suspendido tenga una nueva prioridad mayor que la de los procesos en estado Preparado. En este caso se deberá decidir entre ejecutar el proceso de mayor prioridad con el coste consiguiente de la operación de intercambio si no hay espacio en memoria principal para todos los procesos, o bien esperar a que haya espacio suficiente en memoria principal para albergar al proceso suspendido.
- Transición Preparado-Preparado y Suspendido: Como veíamos en la transición anterior, se puede producir un intercambio entre un proceso en estado Preparado

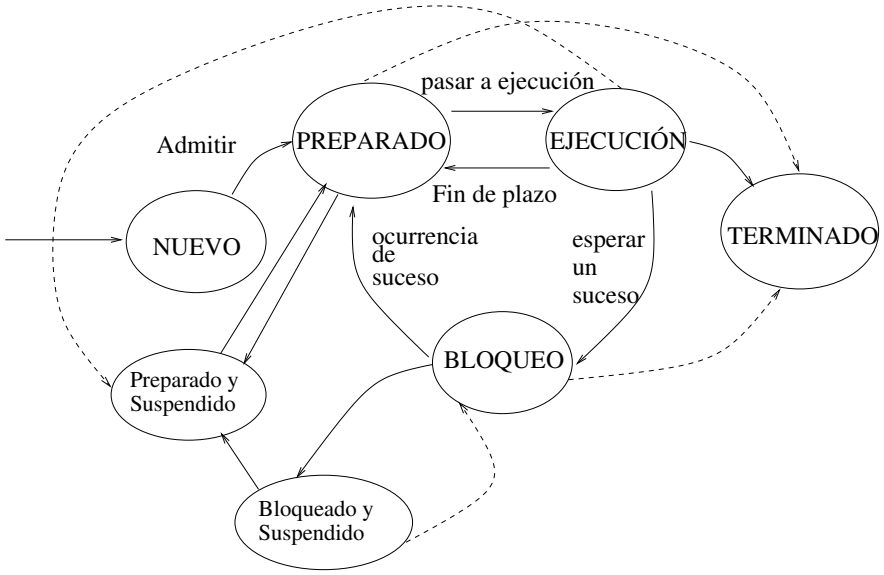


Figura 2.7: Diagrama de 5 estados + 2 suspendidos (preparado y suspendido, bloqueado y suspendido)

y Suspendido y otro en estado de Preparado si no hay memoria suficiente para ambos. Generalmente, el SO prefiere suspender a un proceso bloqueado en vez de a uno en estado Preparado. Sin embargo, puede ser necesario suspender a un proceso Preparado si ésta es la única forma de liberar un bloque lo suficientemente grande de memoria principal. Además, el SO puede escoger suspender un proceso Preparado de más baja prioridad en lugar de uno bloqueado de prioridad más alta si se estima que el proceso bloqueado pronto pasará a estado de Preparado.

- Transición Bloqueado y Suspendido-Bloqueado: Si un proceso termina y libera memoria principal y existe además algún proceso en la cola de procesos Bloqueados y Suspendidos con mayor prioridad de la de todos los proceso que se encuentran en la cola de Preparados y Suspendidos, el SO puede entonces traer el proceso a memoria si tiene razones para suponer que va a ocurrir pronto el suceso que bloqueó al proceso.

- Transición Ejecución-Preparado y Suspendido: Generalmente, un proceso en ejecución pasa al estado Preparado cuando expira su fracción de tiempo de procesador, sin embargo, si se está expulsando al proceso porque hay otro de prioridad mayor en la lista de Bloqueados y Suspendidos que acaba de desbloquearse, entonces el SO podría pasar directamente el proceso en ejecución a la cola de Preparados y Suspendidos, liberando así espacio en la memoria principal.

ejemplo 1 proceso de prioridad 1 en estado Bloqueado, 3 procesos de prioridad 2 en Preparado, 1 proceso en Ejecución de prioridad 1 y 1 Preparado de prioridad 3. 1

bloqueado y suspendido de prioridad 1. los bloqueados lo dejarán de estar pronto. ¿qué transición tendría lugar?

Entre las razones más habituales para la suspensión de procesos podemos citar las siguientes:

1. Intercambio un proceso por otro(s): El SO necesita liberar memoria principal para cargar un proceso que está listo para ejecutarse.
2. Suspensión de un proceso por el SO por *sospechar* que está causando algún tipo de problema.
3. Solicitud expresa del usuario.
4. Un proceso puede ejecutarse periódicamente y puede ser suspendido mientras espera el intervalo de tiempo antes de una nueva ejecución.
5. Por una petición del proceso padre.

2.4. Estructuras de control del sistema operativo

El SO es el controlador de los sucesos que se producen en un sistema informático y es el responsable de planificar y expedir a los procesos para su ejecución en el procesador. El SO es quien asigna los recursos a los procesos y el que responde a las solicitudes de servicios básicos realizadas por los programas de usuario, esencialemte se puede considerar al SO como una entidad que administra el uso que hacen los procesos de los recursos del sistema.

A continuación se tratarán los elementos que necesita el SO para llevar a cabo sus labores de control de procesos y de administración de recursos.

2.4.1. Tablas de memoria, de E/S, de archivos y de procesos

Si el SO va a administrar procesos y recursos, entonces tiene que disponer de información sobre el estado actual de cada proceso y de cada recurso. El método universal para obtener esta información es sencillo. El sistema operativo construye y mantiene tablas de información sobre cada entidad que está administrando. Por ejemplo, las tablas de memoria se utilizan para mantener el control sobre la memoria principal o *real* y la secundaria o *virtual*.

Las tablas de memoria deberán incluir la siguiente información:

1. Asignación de memoria principal a los procesos.
2. Asignación de memoria secundaria a los procesos.
3. Atributos de protección de segmentos de memoria principal o secundaria.
4. Información necesaria para la gestión de la memoria secundaria.

Las tablas de E/S son utilizadas por el SO para administrar los dispositivos y los canales de E/S del sistema informático. En un momento dado, un dispositivo de E/S puede estar disponible o estar asignado a un proceso particular. Si hay una operación de E/S en marcha el SO necesita conocer el estado de dicha operación y la posición de memoria principal que se está utilizando como origen o destino de la transferencia de E/S.

El SO también mantiene un conjunto de tablas de archivos, las cuales ofrecen información sobre las propiedades de éstos. Sobre su posición y distribución en la memoria secundaria, su estado actual y otros atributos. Gran parte de esta información, sino toda, puede ser mantenida y utilizada por un sistema de gestión de archivos. Éste consisitirá en un módulo del SO bien diferenciado y su labor se ocupará de todas las operaciones necesarias para la gestión y tratamiento de los archivos.

Un ejemplo de estructura para la ubicación de archivos es la conocida como FAT (File Allocation Table)[2]. Un ejemplo de sistema de ficheros sería el utilizado por el SO Windows NT y conocido NTFS (NT Filesystem).

Las tablas de procesos almacenan información relativa al conjunto de procesos activos presentes en un instante determinado en el sistema. La información típicamente almacenada para cada proceso y conocida como imagen del proceso en memoria consiste en:

1. Datos de usuario: Almacena los datos con que trabaja el proceso así como la pila utilizada por éste. [espacio de direcciones del proceso]
2. Programa de usuario: Contiene el código objeto[3] del programa que se va a ejecutar. [espacio de direcciones del proceso]
3. Pila de sistema: Se utiliza para almacenar parámetros y direcciones de retorno. [estructuras del sistema operativo]
4. Bloque de control de proceso: Contiene la información necesaria para que un proceso pueda ser gestionado y controlado por el SO. [estructuras del sistema operativo]

2.4.2. Bloque de control de procesos (BCP)

El SO agrupa toda la información que necesita conocer respecto a un proceso particular en una estructura de datos denominada *descriptor de proceso* o *bloque de control de proceso* (BCP). Cada vez que se crea un proceso, el SO crea uno de estos bloques para que sirva como descripción en tiempo de ejecución durante toda la vida del proceso (Véase figura 2.8). Cuando el proceso termina, su BCP es *liberado* y devuelto al depósito de celdas libres del cual se extraen nuevos BCPs.

[2] Consiste en una tabla enlazada que hace referencia a los sectores del disco duro asignados a un mismo fichero. FAT da problemas al fragmentarse frecuentemente lo que provoca un retardo al acceder a esos datos. Para solucionar este problema existen herramientas de desfragmentación. En sistemas UNIX se trata desde el principio más eficientemente esta asignación y los datos almacenados apenas sufren fragmentación.

[3] código que puede ser ejecutado por una determinada arquitectura.

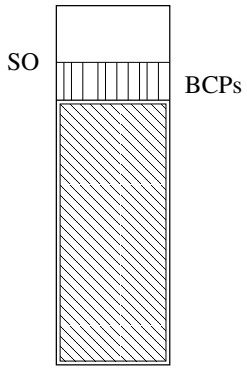


Figura 2.8: Dentro de la memoria asignada al SO tenemos unos bloques reservados para los BCPs de los procesos.

Un proceso resultará conocido para el SO y, por tanto, susceptible de ser elegido para competir por los recursos del sistema sólo cuando existe un BCP activo asociado a él[4]. El BCP es una estructura de datos[5] con campos para registrar los diferentes aspectos de ejecución del proceso así como de la utilización de los recursos. La información del BCP se agrupa generalmente en las siguientes categorías:

1. Identificación del proceso

   La información correspondiente a la identificación del proceso consiste en un conjunto de identificadores que incluyen:

   a) El identificador del proceso (PID): Consiste en un número entero asignado por el sistema.

   b) El identificador del proceso padre

   c) La identificación del usuario: Es una cadena de caracteres.

2. Información del estado del procesador: La información relativa al estado del microprocesador consta de:

   a) Registros visibles para el usuario: Son los registros utilizados por el proceso para almacenar datos de entrada y resultados.

   b) Registros de control y estado, entre los cuales de incluyen el *contador de programa* (PC), los *regitros de códigos de condición*[6], los *registros con indicadores de habilitación o inhabilitación de interrupciones y el modo de ejecución.*

[4] en el modelo de 5 estados, había un estado llamado Nuevo en donde el SO sabía que existía un nuevo proceso pero sin BCP ya que aún no era candidato para asignarle recursos.
[5] El SO tiene bloques de memoria libres preparados para almacenar BCPs.
[6] bits que reflejan el resultado de una operación aritmética (bit de overflow, bit de acarreo, bit de cero, etc)

c) Puntero a la pila del proceso: El proceso utiliza una estructura de pila para almacenar parámetros y direcciones de retorno de funciones y procedimientos

3. Información de control y gestión del proceso: La información de control y gestión del proceso incluye:

   a) Información de planificación y estado: esta información es necesaria para que el SO lleve a cabo sus funciones de planificación. Los elementos típicos de esta información son los siguientes
      1) Estado del proceso (ejecución, preparado, etc).
      2) Prioridad de planificación (se utilizarán algoritmos de planificación que usarán esta información).
      3) Información para la planificación: ésta depende del algoritmo de planificación utilizado.
      4) Suceso por el que se encuentre esperando el suceso para reanudar su ejecución

   b) Estructuación de datos: Un proceso puede estar enlazado con otros procesos formando una cola, un anillo o alguna otra estructura. Por ejemplo; todos los procesos que se encuentran en estado preparado con un determinado nivel de prioridad pueden estar enlazado en una cola. El BCP podrá contener entonces punteros a otros BCPs para dar soporte a esas estructuras.

   c) Comunicación entre procesos: en el BCP pueden ubicarse indicadores, señales y mensajes asociados con la comunicación entre los procesos independientes.

   d) Privilegios de los procesos: A los procesos se les otorgan privilegios en términos de la memoria a la que pueden acceder y los tipos de instrucciones que pueden ejecutar. Además, también se pueden aplicar privilegios al uso de servicios y utilidades del sistema.

   e) Gestión de memoria: Esta sección incluye punteros a las tablas de página y/o segmentos que describen la memoria asignada al proceso.

   f) Recursos en propiedad y utilización de los procesos: Se incluyen los recursos controlados por el proceso tales como los ficheros abiertos por éste. También se suele incluir un histórico de la utilización del procesador o de otro recurso. Esta información puede ser necesaria para el planificador.

2.4.3. Estados del sistema y listas de procesos

Un estado de un proceso es sólo un componente del estado global del sistema que incluye a todos los procesos y recursos. Para controlar la evolución de todos los procesos, el SO mantiene listas de BCPs clasificadas por el estado actual de los procesos aceptados. En general, existirá una lista con los BCPs de todos los procesos en situación de *preparado* y una lista con todos los BCPs en situación de *suspendido*. Mediante estas listas el SO forma colecciones de procesos en estados análogos y serán examinadas por las rutinas de asignación de recursos del SO.

ejemplo El planificador buscará el siguiente proceso a ejecutar en la lista de los BCPs de procesos preparados.

El rendimiento del SO puede mejorar ordenando y actualizando estas listas de la manera más conveniente para las rutinas del SO que operan con ellas. Las transiciones de estado de un proceso quedarán reflejadas en el cambio de su BCP de una lista a otra.

2.4.4. Conmutación de procesos

Una transición entre dos procesos residentes en memoria en un sistema multitarea se denomina *conmutación de procesos* o *conmutación de tareas.* Las principales operaciones implicadas en una conmutación de procesos están resumidas en la figura 2.9.

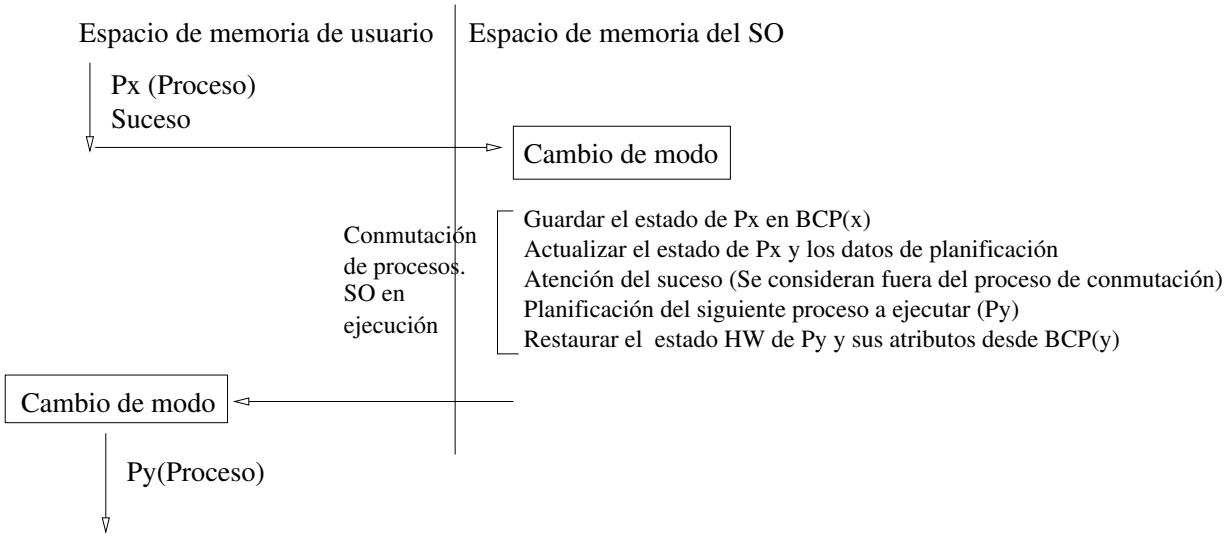


Figura 2.9: Esquema de conmutación de recursos

La figura se interpreta de la siguiente manera

- Estamos en modo usuario con un proceso P_x ejecutándose. Ocurre un suceso.
- Pasamos a modo supervisor (ejecutamos SO).
  - Guarda el estado del proceso interrumpido en su BCP correspondiente
  - Actualiza el estado de P_x (en función del suceso que haya tenido lugar, el estado será uno u otro) y los datos de planificación (Algunos ssoo tienen ciertas consideraciones para recalcular estos datos de planificación para un proceso. Por ejemplo, restar prioridad a procesos largos.
  - Atendemos al suceso dependiendo de cada caso (no incluimos esta operación en lo que consideramos *proceso de conmutación*)

  - Planificamos cuál va a ser el siguiente proceso a ejecutar (que podría ser el mismo de inicio).
  - Una vez elegido el nuevo proceso a ejecutar, llamémosle P_y, hay que restaurar su estado HW y recuperamos sus atributos gracias a su BCP.

- Una vez elegido el nuevo proceso, realizamos un cambio de modo.

La conmutación de procesos es una operación considerablemente más compleja y costosa que la conmutación del contexto de interrupción y puede ser bastante complicada en ssoo grandes que disponen de un mantenimiento preciso de recursos y de sofisticados esquemas de planificación. Dada su complejidad y su relativamente alta frecuencia de ocurrencia, la implementación de la conmutación de procesos puede afectar significativamente al rendimiento de un SO de multiprogramación.

Es especialmente importante en sistemas donde el tiempo es un factor crítico tales comos los *sistemas en tiempo real*. La eficiencia de la conmutación de procesos puede ser mejorada con ayuda del hardware y una estructura de software especial conocida como **hebra** o **hilo**.

Un esquema hardware habitualmente empleado para la acelerar la conmutación de procesos es disponer de múltiples conjuntos estructuralmente idénticos de registros del procesador. Un conjunto para el SO y otro para los procesos de usuario. Un bit dedicado en la única copia de la palabra del estado del procesador indica el estado actual de operación supervisor o usuario y el conjunto de registros activos. Este método reduce el gasto software por conmutación de modo y la mayor parte del proceso de almacenamiento del estado hardware a la ejecución de unas pocas instrucciones de manipulación de bits. Si se dispone de más de dos conjuntos de registros, se pueden aplicar ahorros similares a los procesos de usuario hasta alcanzar el número de conjuntos de registros disponibles.

Pregunta ¿Por qué la conmutación de procesos es más costosa que la conmutación del contexto de interrupción?

2.4.5. Servicios del sistema operativo para la gestión de procesos

Aunque los ssoo suelen diferir en su filosofía y objetivos de diseño sus capas del núcleo más internas muestran una gran similitud en cuanto al tipo y rango de primitivas de gestión de procesos que ofrecen. Los detalles y parámetros varían inevitablemente de un sistema a otro, pero las funciones proporcionadas por la colección total de llamadas al SO son muy parecidas. Esto es así porque el concepto de proceso es común a todos los ssoo pero cada uno los gestiona de forma distinta.

- El servicio de creación de procesos `crear(Id_proceso, atributos)`: En respuesta a esta llamada el SO crea un proceso con el identificador y los atributos especificados o predeterminados por el sistema. *El SO obtiene un nuevo BCP del conjunto de memoria libre, rellena sus campos con los parámetros proporcionados y/o predeterminados e inserta el BCP en la cola de procesos preparados*[7]. De esta manera,

[7] cuando un proceso tiene BCP ya puede ejecutarse por lo que ya estará en el nivel de Preparado.

el proceso especificado podrá ser elegido por el SO para su ejecución. Algunos de los parámetros o atributos que pueden definirse en el momento de creación de un proceso son los siguientes:

1. Nivel de privilegios.
2. Nivel de prioridad.
3. Tamaño y requisitos de memoria.
4. Información sobre acceso a memoria y derechos de acceso a dispositivos de E/S.
5. Tamaño máximo del área de datos y/o de la pila.

- El servicio de terminación de procesos `terminar(Id_proceso)`: La invocación de esta llamada hace que el SO destruya el proceso designado y lo suprima del sistema. El SO reacciona reclamando todos los recursos asignados al proceso especificado, cerrando los archivos abiertos por o para el proceso y efectuando otras operaciones de que pueden considerarse necesarias y que dependerán de la naturaleza del proceso. A continuación el BCP es eliminado de la lista en que resida y devuelto al conjunto de posiciones libres. Un proceso puede eliminarse a sí mismo pero no puede crearse a sí mismo. Un proceso podrá eliminar a otro siempre y cuando tenga privilegios para ello.

- El servicio para abortar un proceso `abortar(Id_proceso)`: Esta orden supone la terminación forzosa de un proceso. El SO efectúa generalmente muchas de las acciones que conlleva la orden `terminar`. Habitualmente se proporciona un volcado de registros y memoria junto con la información relativa a la identidad del proceso que se aborta y la razón de la acción. El uso más frecuente de esta orden es para terminaciones involuntarias.

  ejemplo en MS Windows NT se produce una operación inválida y el SO lo aborta. Aparece una ventana *detalles* en donde se ofrece un volcado del registro y memoria.

- El servicio dividir/unir `fork()/join()`: La operación dividir `fork()` se utiliza para dividir una sencuencia de instrucciones en dos secuencias que se ejecutan concurrentemente. Se crea así un nuevo proceso (proceso hijo) que ejecuta una rama del código dividido mientras el proceso padre continúa ejecutando la otra. Esta llamada proporciona al proceso padre el identificador del proceso hijo y lo utiliza para asignarle una rama de código. La operación unir `join()` se utiliza para reunir las dos secuencias de códigos divididos y puede ser empleada por un proceso padre para sincronizarse con un proceso hijo.

  ejemplo Esto se utilizó mucho en servidores de ficheros. Existe un proceso llamado *listener* que se mantiene a la escucha. Cuando recibe la petición de un cliente (identificado por una dirección IP, etc). *Listener* realiza un `fork()` y al nuevo

proceso le proporciona la dirección IP del cliente mientras el proceso Padre sigue a la escucha.

- El servicio para bloquear un proceso `bloquear(Id_proceso)`: Como respuesta a esta llamada, el proceso designado queda bloqueado idefinidamente y pasa a este estado. Un proceso puede bloquearse a sí mismo o puede bloquear a otro proceso cuando está autorizado para ello en virtud de su nivel de privilegio, prioridad o pertenencia a una familia.
- Servicio para reanudar un proceso `reanudar(Id_proceso)`: Esta llamada al sistema saca un proceso del estado de bloqueo impuesto por la llamada anterior, el BCP del proceso pasa a la lista de preparados. Un proceso no puede reanudarse a sí mismo si está en estado de bloqueo indefinido, por lo que si se encuentra en
- esta situación no podrá continuar hasta que sea reanudado.
- Servicio para retardar un proceso `retardar(Id_proceso, tiempo)`: Esta orden es conocida en bastantes ssoo como **sleep** y bloquea a un proceso durante un tiempo igual al especificado.
- Servicio para leer los atributos de un proceso `leer_Atributos(Id_proceso,atributos)`: Esta llamada vuelca en la estructura suministrada el conjunto de atributos del proceso.
- Servicio para modificar la prioridad de un proceso `modificar_Prioridad(Id_proceso, nuevo_valor_prioridad)`: Esta llamada al sistema establece como nueva prioridad del proceso la especificada por parámetros.

3 Planificación de procesos

3.1. Concepto y criterios de planificación

La planificación hace referencia a un conjunto de políticas y mecanismos incorporados al SO que gobiernan el orden en que se ejecutan los trabajos que deben ser completados por el sistema informático. Un planificador es un módulo del SO que selecciona el siguiente trabajo a admitir en el sistema y el siguiente proceso que tomará el control sobre el procesador. El objetivo primario de la planificación es optimizar el rendimiento del sistema de acuerdo con los criterios considerados más importantes por los diseñadores del mismo.

Entre las medidas de rendimiento y los criterios de optimización más habituales que los planificadores utilizan para llevar a cabo su labor se encuentran los siguientes:

3.1.1. Utilización del procesador:

La utilización del procesador es la fracción de tiempo promedio durante la cual el procesador está ocupado, es decir, la fracción de tiempo durante la cual el procesador se encuentra activo ejecutando algún proceso, bien de usuario, bien del propio SO. Con esta interpretación, la utilización del procesador puede ser medida con relativa facilidad, por ejemplo mediante un *proceso nulo* especial[1] que se ejecute cuando ningún otro proceso pueda hacerlo. Una alternativa es considerar únicamente la operación en modo usuario y, por tanto, excluir el tiempo empleado para el SO[2].

En cualquier caso, el objetivo es mantener al procesador ocupado tanto tiempo como sea posible. De esta forma, se conseguirá que los factores de utilización de los restantes componentes también sean elevados obteniéndose con ello buenas medidas de rendimiento.

3.1.2. Productividad

La produtividad se refiere a la cantidad de trabajo completada por unidad de tiempo. Un modo de expresarla es definiéndola como el número de trabajos de usuario ejecutados por una unidad de tiempo. Cuanto mayor sea este número, más trabajo aparentemente está siendo ejecutado por el sistema.

[1] en MINIX existe un proceso llamado idle:

```
idle(){
     for(;;);
}
```

[2] Podríamos decir que el tiempo de uso real del procesador es el tiempo total de uso menos el tiempo que se estuvo ejecutando la instrucción idle().

3.1.3. Tiempo de retorno

El tiempo de retorno T_R se define como el tiempo que transcurre desde el momento en que un trabajo o programa es remitido al sistema hasta que es totalmente completado por el mismo. Es decir, el tiempo de retorno T_R es el tiempo consumido por el proceso dentro del sistema y puede ser expresado como la suma del *tiempo de servicio* o *tiempo de ejecución + el tiempo de espera*. $T_R = T_S + T_E$.

3.1.4. Tiempo de espera

El tiempo de espera T_E es el tiempo que un proceso o trabajo consume a la espera de la asignación de algún recurso o de que tenga lugar algún evento. En este tiempo también se incluyen el periodo de espera por la obtención del propio procesador[3] debido a la competencia con otros procesos en un sistema con multiprogramación. Este tiempo es la penalización impuesta por compartir recursos con otros procesos y puede expresarse como el tiempo de retorno - el tiempo de ejecución efectivo. El tiempo de espera T_E elimina la variabilidad debida a las diferencias en tiempos de ejecución del trabajo.

3.1.5. Tiempo de respuesta

El tiempo de respuesta en sistemas interactivos se define como el tiempo que transcurre desde el momento en que se introduce el último carácter de una orden que desencadena la ejecución de un programa o transacción hasta que aparece el primer resultado en el terminal. Generalmente también se le denomina tiempo de respuesta de terminal.

En sistemas en tiempo real, el tiempo de respuesta es esencialmente una latencia y se define como el tiempo que transcurre desde el momento en que un suceso interno o externo es señalado hasta que se ejecuta la primera instrucción de su correspondiente rutina de servicio. A este tiempo suele denominársele tiempo de respuesta al proceso.

3.2. Tipos de planificadores

En un SO complejo pueden coexistir tres tipos de planificadores: A corto, a medio y a largo plazo.

3.2.1. Planificador a largo plazo (PLP)

Su misión consiste en controlar la admisión de procesos nuevos al sistema. Cuando está presente este tipo de planificador, su objetivo principal es proporcionar una mezcla equilibrada de trabajos. El PLP decide cuándo se da entrada al sistema a un nuevo proceso para que éste sea ejecutado. Este proceso puede proceder de la respuesta al envío de un trabajo por lotes o bien a la orden de ejecución realizada por el usuario. En cierto modo, el PLP actúa como una válvula de admisión de primer nivel para mantener

[3] no confundir estos tiempos. El tiempo de espera por el procesador está incluido en el tiempo de espera total T_E.

la utilización de recursos al nivel deseado. Es importante conseguir una administración equilibrada para saber cómo conjugar procesos interactivos que tienen retardos especiales con procesos por lotes que son una simple de cola de cálculo.

Por ejemplo, cuando la utilización del microprocesador puede admitir más trabajos, el planificador puede dar entrada al sistema a nuevos procesos y aumentar con ello la probabilidad de asignación de alguno de estos procesos al procesador. Por el contrario, cuando el tiempo para la utilización del procesador resulte alto y así se refleje en el deterioro en el tiempo de espera, el PLP puede optar por reducir la frecuencia de admisión de procesos a situación de preparado. El PLP es invocado generalmente cada vez que un trabajo completado abandona el sistema.

La frecuencia de llamada al PLP es, por tanto, dependiente del sistema y de la carga de trabajo pero generalmente es mucho más baja que para los otros dos tipos de planificadores[4].

Como resultado de esta no demasiada frecuente ejecución, el PLP puede incorporar algoritmos relativamente complejos y computacionalmente intensivos para admitir trabajos al sistema. En términos del diagrama de transición de estados de procesos, el PLP quedará a cargo de las transiciones del estado nuevo al estado preparado o listo.

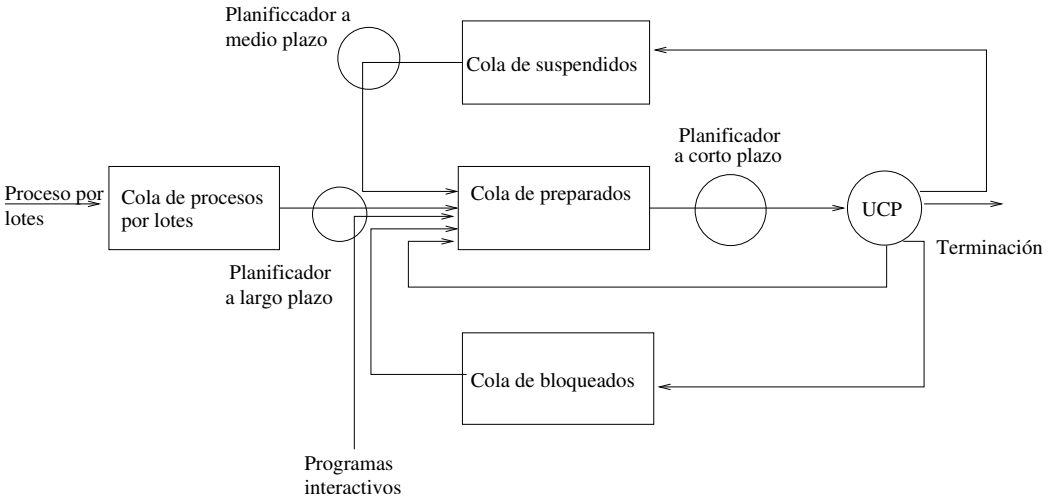


Figura 3.1: Esquema de un SO con planificadores

3.2.2. Planificador a corto plazo (PCP)

Este planificador decide qué procesos toman el control de la CPU. El PCP asigna el procesador entre el conjunto de procesos preparados residentes en memoria. Su principal objetivo es maximizar el rendimiento del sistema de acuerdo a con el conjunto de criterios elegidos. Al estar a cargo de la transición de estado preparado a ejecución, el PCP deberá ser invocado cuando se realice una operación de conmutación de procesos para seleccionar

[4] en una unidad de tiempo se utilizará menos veces y ello hará posible que su estructura sea más compleja.

el siguiente proceso a ejecutar. En la práctica el PCP es llamado cada vez que un suceso interno o externo hace que se modifique alguna de las condiciones que definen el estado actual del sistema. Algunos de los sucesos que provocan una replanificación en virtud de su capacidad de modificar el estado del sistema son:

1. Tics de reloj, es decir, interrupciones basadas en el tiempo.
2. Interrupciones y terminaciones de operaciones de E/S.
3. Llamadas de operación al sistema operativo frente a llamadas de consulta.
4. Envío y recepción de señales.
5. Activación de programas interactivos.

En general, cada vez que ocurre uno de estos sucesos, el SO llama al PCP para determinar si debería planificarse otro proceso para su ejecución.

3.2.3. Planificador a medio plazo (PMP)

El PMP tiene por misión traer procesos suspendidos a la memoria principal. Este planificador controla la transición de procesos en situación de suspendidos a situación de preparados. El PMP permanecerá inactivo mientras se mantenga la condición que dio lugar a la suspensión del proceso, sin embargo, una vez desaparecida dicha condición el PMP intenta asignar al proceso la cantidad de memoria principal que requiera y volver a dejarlo en situación de preparado. Para funcionar adecuadamente, el PMP debe disponer de información respecto a las necesidades de memoria de los procesos suspendidos, lo cual no es complicado de llevar a la práctica ya que el tamaño real del proceso puede ser calculado en el momento de suspenderlo almacenándose en el BCP.

Este planificador será invocado cuando quede espacio libre en memoria por la terminación de un proceso o cuando el suministro de procesos preparados quede por debajo de un límite especificado.

3.3. Algoritmos de planificación

Antes de comenzar a estudiar los distintos tipos de algoritmos de planificación es importante tener en cuenta que hay dos categorías generales de éstos.

La planificación no apropiativa[5] Se basa en que una vez que el proceso pasa a estado de ejecución no abandona el procesador hasta que termina o hasta que se bloquea en espera de una operación de E/S o al solicitar algún servicio del sistema.

La planificación apropiativa Un proceso que está ejecutando puede ser interrumpido por el sistema operativo para otorgar el procesador a un proceso distinto en función de los criterios de planificación utilizados; prioridad, número de usos del procesador, etc.

3.3.1. Algoritmo First Come First Serve (FCFS)

La disciplina de planificación más sencilla es el algoritmo FCFS. La carga de trabajo se procesa simplemente en un orden de llegada. Por no tener en consideración el estado del sistema ni las necesidades de recursos de los procesos individuales, la planificación FCFS puede dar lugar a pobres rendimientos. Este algoritmo exhibe un alto tiempo de respuesta a sucesos debido a la falta de expropiación y caracterización con las propiedades de los procesos. La planificaciñon FCFS elimina la noción e importancia de las prioridades de los procesos.

ejercicio Sean dos procesos P_1 y P_2 con tiempos de servicios de 20 y 2 unidades de tiempo, respectivamente. Si el primero en llegar es el proceso P_1, calcular los tiempos de retorno de ambos procesos y el tiempo de retorno medio. Realizar los mismos cálculos si el primero en llegar es el proceso P_2.

- $P_2P_1 \longrightarrow$

$$\left.\begin{array}{llll} P_1: & T_E = 0 & T_S = 20 & T_R = 20 \\ P_2: & T_E = 20 & T_S = 2 & T_R = 22 \end{array}\right\} \begin{array}{l} \bar{T_R} = 21 \\ \bar{T_E} = 10 \end{array}$$

- $P_1P_2 \longrightarrow$

$$\left.\begin{array}{llll} P_2: & T_E = 0 & T_S = 2 & T_R = 2 \\ P_1: & T_E = 2 & T_S = 20 & T_R = 22 \end{array}\right\} \begin{array}{l} \bar{T_R} = 12 \\ \bar{T_E} = 1 \end{array}$$

3.3.2. Algoritmo por reparto circular de tiempo (RR, Round-Robin)

En entornos interactivos tales como sistemas de tiempo compartido, el requisito principal es proporcionar tiempos de espera razonablemente buenos y, en general, compartir los recursos del sistema equitativamente entre todos los usuarios. Solamente las disciplinas de planificación que permiten la expropiación del procesador pueden ser consideradas en tales entornos y una de las más utilizadas es la de *Reparto circular de tiempos o por turnos*. Básicamente, el tiempo del procesador se divide en cuotas o cuantos que son asignados a los procesos solicitantes. Ningún proceso puede ejecutarse durante más tiempo que el establecido por ese cuanto si hay más procesos esperando en la cola de preparados. Si un proceso necesita más tiempo para completarse después de agotar su cuota de tiempo, volverá de nuevo a la cola de procesos preparados. Si el proceso termina antes de que expire esta cuota de tiempo, el planificador dará inmediatamente el procesador a otro proceso en situación de preparado. Con esta planificación y en un sistema con n procesos activos, cada proceso recibe aproximadamente $\frac{1}{n}$ del tiempo del procesador.

Con este algoritmo de planificación, los procesos cortos pueden ser ejecutados dentro de una única cuota de tiempo y presentarán por tanto buenos tiempos de respuesta. En el caso de procesos más largos, éstos pueden circular unas cuantas veces a través de la cola de preparados antes de terminar. El tiempo de respuesta a estos procesos más largos será siempre proporcional a sus necesidades de recursos.

La planificación por reparto de tiempo requiere el soporte de un temporizador de intervalos que se programa generalmente para que interrumpa al SO cada vez que expire una cuota o cuanto de tiempo, forzando así la ejecución del planificador. El rendimiento de este tipo de planificación es muy sensible a la elección de la cuota de tiempo, que suele oscilar entre 1 y 100 milisegundos dependiendo del sistema. Una cuota demasiado corta puede dar lugar a retrasos significativos debido a las frecuentes interrupciones del temporizados y consiguientes conmutaciones de procesos. En el otro extremo, una cuota demasiado larga transformaría a un planificador RR en un planificador FCFS.

tiempo de retorno normalizado se define como el tiempo de retorno/tiempo de servicio.

ejercicio Sea una sistema con 5 procesos activos, los tiempos de activación y de servicio de cada uno de ellos son los siguiente

		$P_A P_B$	P_C	P_D	P_E	
tiempo de llegada	T_{ll}	0	2	4	6	8
tiempo de servicio	T_S	3	6	4	5	2

1. Obtener el datagrama de ejecución si el algoritmo de planificación utilizado es FCFS.

2. Cronograma de ejecución si la planificación es Round-Robin con una cuota de una unidad de tiempo.

3. Cronograma de ejecución si la planificación es Round-Robin con una cuota de cuatro unidades de tiempo.

	P_A	P_B	P_C	P_D	P_E
T_{ll}	0	2	4	6	8
T_S	3	6	4	5	2
T_R	3	7	9	12	12
T_E	0	1	5	7	10
T_{RN}	1	1.17	2.25	2.48	6.0
T_R	4	17	13	14	7
T_E	1	11	9	9	5
T_{RN}	1.33	2.83	3.27	2.80	3.5
T_R	3	17	7	14	9
T_E	0	11	3	9	7
T_{RN}	1	2.83	1.75	2.80	4.5

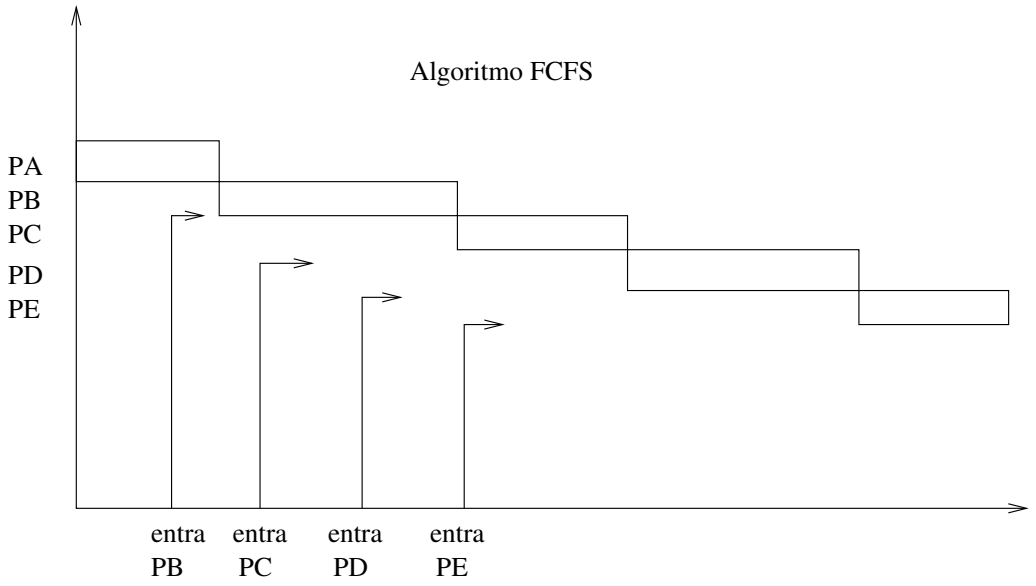


Figura 3.2: Solución gráfica al ejercicio propuesto: FCFS

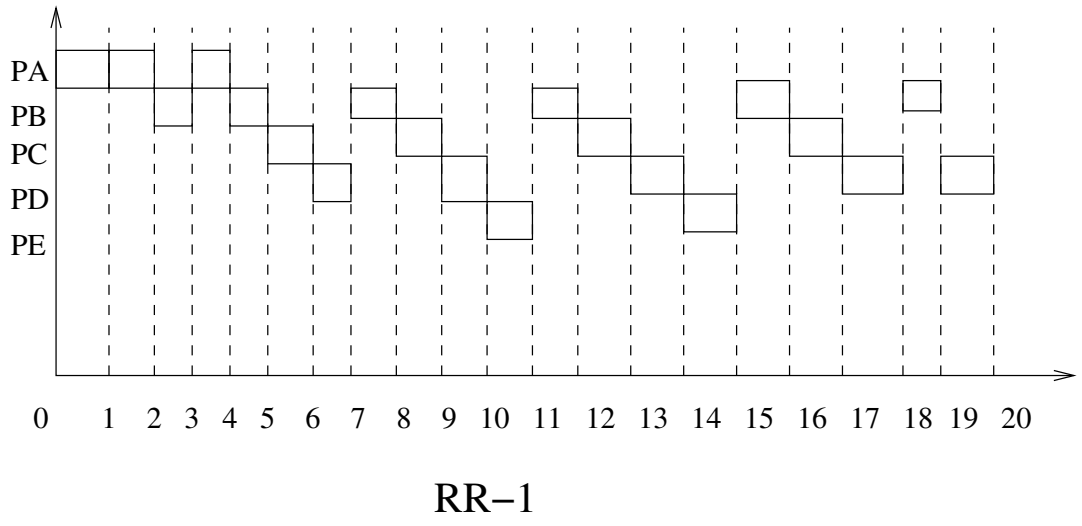


Figura 3.3: Solución gráfica al ejercicio propuesto: RR-1

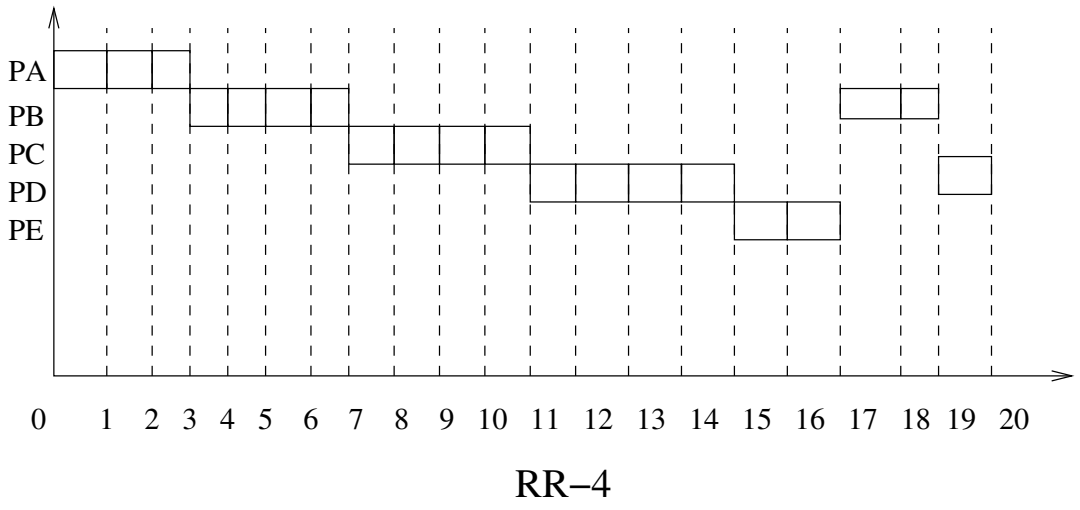


Figura 3.4: Solución gráfica al ejercicio propuesto: RR-4

3.3.3. Planificación con expropiación basada en prioridades (ED, Event-Driven)

Cada proceso del sistema está asignado a un nivel de prioridad y el planificador siempre elige al proceso preparado con prioridad más alta. Estas prioridades pueden ser estáticas o dinámicas. La prioridad estática no variará a lo largo del ciclo de vida del proceso mientras que la prioridad dinámica sí puede hacerlo. En cualquier caso, los valores iniciales de estas prioridades son asignados por el usuario o el SO en el momento de la creación del proceso. Un problema habitual en este tipo de planificación es la posibilidad de que los procesos de prioridad más baja queden siempre relegados en favor de los de prioridad más alta. Así, con este tipo de planificación no puede garantizarse la terminación de un proceso dentro de un tiempo finito. Hay SSOO donde esto no se puede consentir, por ejemplo; los sistemas operativos en tiempo real.

En sistemas donde tal situación no puede ser tolerada, el remedio habitual lo proporciona la utilización de una prioridad por envejecimiento, según la cual, la prioridad de un proceso aumenta gradualmente en función de su tiempo de permanencia en el sistema. Los procesos más antiguos conseguirán así una prioridad tal que asegura su terminación en tiempo finito. En un SO de tiempo real estricto, donde cada proceso debe tener garantizada su ejecución antes de la expiración de un plazo, se utiliza una disciplina de planificación por plazo más inmediato. Este tipo de planificación daría entrada al procesador al proceso cuyo plazo esté más próximo a cumplirse.

Otro tipo de planificación en este tipo de sistemas es la planificación por mínima laxitud, según la cual se selecciona el proceso con menor diferencia entre el tiempo que tarda en cumplirse el plazo y el tiempo restante de computación. Véase la figura 3.5.

La prioridad puede subir porque ha transcurrido un tiempo determinado o porque al proceso se le ha expropiado x veces del procesador o por las razones que estén definidas

pero siempre premiando de alguna forma el tiempo de vida del proceso en el sistema.

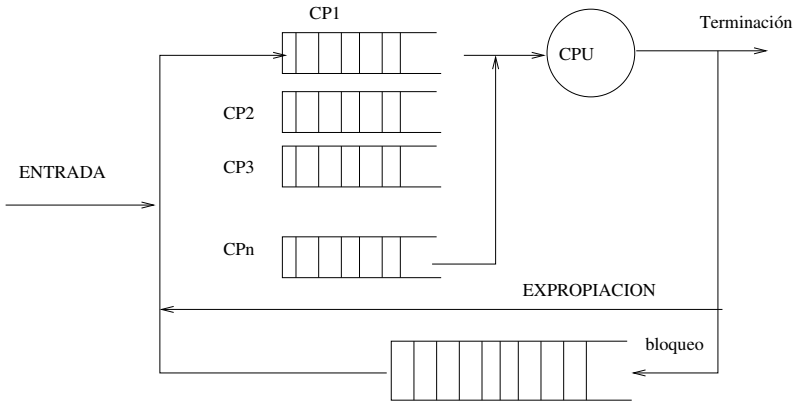


Figura 3.5: Esquema de un sistema con planificación ED

3.3.4. Planificación MLQ (Multiple level queues)

En sistemas mixtos donde coexisten procesos interactivos y procesos por lotes, resulta más conveniente adoptar una planificación compleja que combine a varias disciplinas. Un modo de implementar esta planificación es clasificar la carga de trabajo de acuerdo con sus características y mantener colas de procesos separados servidas por diferentes planificadores. A este método se le denomina Planificación por colas MultiNivel. Véase la figura 3.6.

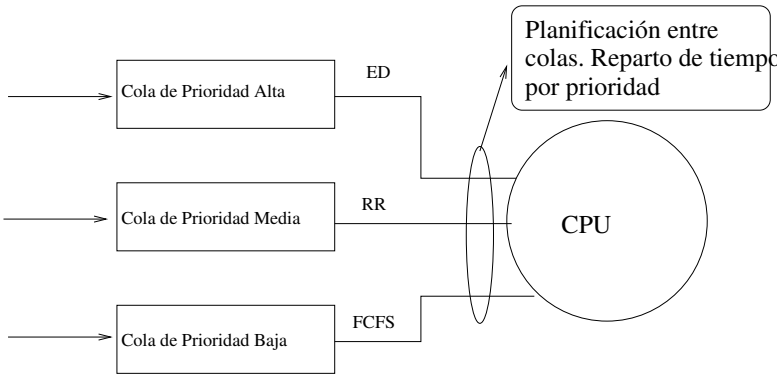


Figura 3.6: Esquema de un sistema complejo.

4 Programación Concurrente

4.1. Multitarea, multiprogramación y multiproceso

Un sistema multitarea es aquel que permite la ejecución de varios procesos sobre un procesador mediante la multiplexación de este entre los procesos.

La multitarea se implementa generalmente manteniendo el código y los datos de varios procesos simultáneamente en memoria y multiplexando el procesador y los dipositivos E/S entre ellos.

La multitarea suele asociarse con soporte software y hardware para la protección de memoria con el fin de evitar que los procesos corrompan el espacio de direcciones y el comportamiento de otros procesos residentes en memoria, un sistema multitarea, sin embargo, no tiene necesariamente que soportar formas elaboradas de gestión de memoria y archivos. En este sentido multitatarea es sencillamente sinónimo de concurrencia.

El término multiprogramación designa a un SO que además de soportar multitarea proporciona formas sofisticadas de protección de memoria y fuerza el control de la concurrencia cuando los procesos acceden a dispositivos E/S y a archivos compartidos, en general la multiprogramación implica multitarea pero no viceversa.

Los SSOO operativos de multiprogramación soportan generalmente varios usuarios en cuyo caso también se les denomina sistemas multiusuario o multiacceso. Los SSOO para multiproceso gestionan la operación de sistemas informáticos que incorporan varios procesadores conocidos habitualmente como *sistemas multiprocesadores*. Los SSOO para multiprocesadores son multitarea por definición ya que soportan laejecución simultánea de varias tareas o procesossobre diferentes procesadores y serán multiprogramados si disponen de los mecanismos de control de concurrencia y protección de memoria adecuados. En general todos los SSOO de multiprogramación se caracterizan por mantener un conjunto de procesos activos simultáneamente que compiten por los recursos del sistema, incluidos el procesador, la memoria y los dipositivos E/S.

Un SO de multiprogramación vigila el estado de todos los procesos activos y de todos los recursos del sistema, cuando se producen cambios importantes de estado, ocuando es invocado explícitamente el SO se activa para asignar recursos y proporcionar ciertos servicios de su repertorio.

4.2. Principios de concurrencia

La concurrencia es el punto clave en los conceptos de multitarea, multiprogramación y multiproceso y es fundamental para el diseño de SSOO, la concurrencia comprende un gran número de cuestiones de diseño incluyendo la comunicación entre procesos, la

compartición y competencia por los recursos, la sincronización de la ejecución de varios procesos y la asignación del procesador a los procesos, la cocncurrencia puede presentarse en tres contextos diferentes:

- Varias aplicaciones: en este caso el tiempo de procesador de una máquina es compartido dinámicamente entre varios trabajos o aplicaciones activas.
- Aplicaciones estructuradas: como consecuencia del diseño modular de una aplicación y la división de la misma en tareas explícitas estas pueden ser ejecutadas de forma concurrente.
- Estrucutra del sistema operativo: como resultado de la aplicación de la estructuración en el diseño del propio SO, de forma que este se implemente como un conjunto de procesos.

Como soporte a la actividad concurrente el SO debe ser capaz de realizar un estrecho seguimiento de los procesos activos, asignando y desasignando recursos entre ellos, el SO debe proteger los datos y recursos de cada proceso contra ingerencias o intrusiones intencionadas o no, de otros procesos.

El resultado de un proceso debe ser absolutamente independiente de la velocidad relativa a la que se realice su ejecución con respecto al resto de procesos, y por supuesto dicho resultado debe ser similar al obtenido si la ejecución del proceso se realizara de forma individual.

4.3. Comunicación y sincronización de procesos

4.3.1. Posibilidades de interacción de procesos

Las posibilidadades de interacción de los procesos pueden clasificarse en función del nivel de conocimiento que cada proceso tiene de la existencia de los demás.

1. Un proceso no tiene en absoluto conocimiento de la existencia de los demás. Se trata de procesos independientes que no están preparados para trabajar conjuntamente y mantienen entre sí una relación exclusivamente de competencia
2. Que los procesos tengan un conocimiento indirecto de los otros procesos. Esta situación tiene lugar cuando los procesos no tienen un conocimiento explícito entre ellos, pero comparten el acceso a algunos dipositivos o zonas de memoria del sistema. Entre estos procesos se establece una relación de cooperación por compartir objetos comunes.
3. Tiene lugar cuando los procesos tienen conocimiento directo unos de otros por haber sido diseñados para trabajar conjuntamente el alguna actividad. Esta situación muestra una relación claramente de cooperación.

En cualquiera de estas tres situaciones hay que dar solución a tres problemas de control:

1. **Necesidad de exclusión mutua**. Es decir, los procesos deberán acceder de forma exclusiva a ciertos recursos o zonas de memoria considerados como críticos.
2. **Interbloqueos**: tienen lugar cuando ninguno de los procesos en competencia puede continuar su ejecución normal por carecer de alguno de los recursos que necesita.
3. **Inhanición**: este problema tiene lugarcuando la ejecución de un proceso queda siempre pospuesta a favor de algún otro de los procesos en competencia.

Por supuesto, el control de la concurrencia involucra inevitablemente al SO ya que es el encargado de asignar y arrebatar los recursos del sistema a los procesos.

4.3.2. Necesidad de sincronización de los procesos: región crítica y exclusión mutua

Independientemente del tipo de interacción existente entre los distintos procesos activos, en un sistema con multiprogramación éstos comparten un conjunto de elementos que deben ser accedidos de forma controlada para evitar situaciones de inconsistencia. Estos elementos compartidos ya sean dispositivos de E/S o zonas de memoria comunes son considerados como críticos y la parte del programa que los utiliza se conoce como *región o sección crítica*. Es muy importante que sólo un programa pueda acceder a su sección crítica en un momento determinado. Por esta razón, el SO debe ofrecer mecanismos que hagan posible una correcta sincronización de los distintos procesos activos en los accesos a los recursos que comparten.

El uso de variables compartidas es una forma sencilla y habitual de comunicación entre procesos interactivos. Cuando un conjunto de procesos tiene acceso a un espacio común de direcciones, se pueden utilizar variables compartidas para una serie de cometidos como, por ejemplo, indicadores de señalización o contadores. Sin embargo, la actualización de variables compartidas puede conducir a inconsistencias; por esta razón, cuando se utilicen hay que asegurarse de que los procesos acceden a ellas debidamente ordenados.

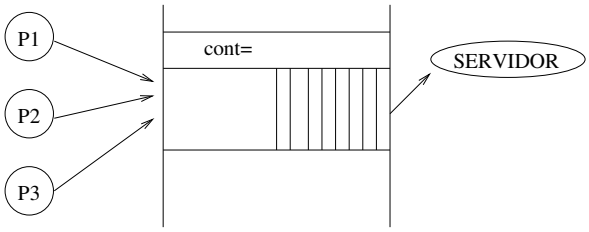


Figura 4.1: Gestión de procesos

Una posible ejecución sería:

1. Llega el proceso 1 y se ejecuta hasta actual (actual=3)
2. El flujo de ejecución se pone en proceso 2 por la razón que sea. Y toma actual y lo pone a 4, lo que hará también proceso 1 cuando le vuelva.

Algoritmo 1 Productor y Servidor

```
Productor(TipoElemento e){
    actual=cont;
    ponerElementoEnCola(e);
    cont=actual+1;
}
TipoElemento Servidor(){
    actual=cont;
    cont=actual-1;
    TipoElemento e = obtenerElementoCola();
    devolver(e);
}
```

Pregunta ¿Se permite cambio de contexto en esta situación?

Respuesta Sí, mientras que las variable de la región crítica no se vean amenzadas.

La actualización de una variable compartida puede ser considerada como una sección crítica. Cuando se permita la entrada de un proceso en una de estas secciones críticas, dicho proceso deberá completar todas las instrucciones que consituyen su región crítica antes de que se permita la entrada a otro proceso a la suya. De este manera, sólo el proceso que ejecuta la sección crítica tiene permitido el acceso a la variable compartida. Los restantes proceso tendrán prohibido el acceso a dicha variable quedando en situación de bloqueado si intentan acceder a su región crítica. Aesta forma de acceso se la denomina *acceso en exclusión mutua*. El acceso en exclusión mutua es una forma de acceso en la que un proceso excluye temporalmente a todos los demás de utilizar un recurso compartido con el fin de asegurar la integridad del sistema. Si el recurso compartido es una variable, la exclusión mutua asegura que, como máximo, un proceso tendrá acceso a ella durante las actualizaciones críticas. En el caso de compartir dispositivos, la exclusión mutua es mucho más obvia si se consideran los problemas que pueden derivarse de su uso incontrolado. Una solución para la exclusión mutua deberá garantizar que se cumplen los siguientes requisitos:

1. Asegurar la exclusión mutua entre los procesos al acceder al recurso compartido.

2. No establecer suposiciones con respecto a las velocidades y prioridades relativas de los procesos en conflicto.

3. Garantizar que la terminación de cualquier proceso fuera de su región crítica no afecta a la capacidad del resto de procesos contendientes para acceder a los recursos compartidos.

4. Cuando más de un proceso desee entrar en su región crítica, se deberá conceder la entrada a uno de ellos en tiempo finito. Evitar interbloqueos.

4.4. Soluciones software para la exclusión mutua

4.4.1. Algoritmo de Dekker

Primer intento

Algoritmo 2 Algoritmo de Dekker

```
Program EMADS1

var turno:0..1;

process 0
begin
   <bucle infinito en donde el proceso 0 comprueba la variable turno>
   <es decir, si no es su turno de ejecución, no realiza nada.>
   <En caso contrario ejecutará la sección crítica y da el paso al>
   <siguiente proceso.>
   while true
   begin
while turno<>0 do {nada};
   <sección crítica>
   turno:=1;
   <resto de código>
   end;
end;

process 1
begin
   while true
   begin
while turno<>1 do {nada};
   <sección crítica>
   turno:=0;
   <resto de código>
   end;
end;

begin
   turno:=0;
   parbegin
      process0;
      process1;
   parend;
end;
```

Este algoritmo garantiza la exclusión mutua. Si dos procesos con distinto ritmo de ejecución, compiten por la región crítica se adoptará el ritmo del proceso más lento. El otro problema viene dado por que si un proceso falla tanto dentro como fuera de su sección crítica, el otro proceso queda bloqueado indefinidamente.

Preguntas asociadas al algoritmo 2:

- ¿un proceso puede ejecutar dos veces seguidas su sección crítica?. No, puesto que en cuanto acaba una ejecución, le pasa el turno al otro proceso.
- ¿si un proceso se cae fuera de su región crítica, afecta al funcionamiento del otro proceso?. Sí, ya que la variable turno se queda a 1 y el otro proceso nunca tendría la condición para ejecutar su sección crítica. Esto ocurre aunque el proceso1 haya logrado cambiar turno a 0, porque una vez que el proceso0 ejecuta y cambia turno a 1, no encuentra respuesta en el otro proceso.
- ¿Que ritmo se sigue cuando los dos procesos tardan tiempos diferentes? Se utiliza el ritmo del más lento, ya que es el que va marcando los tiempos de finalizado de procesos.

Segundo intento

En el algoritmo 3 sí se pueden regiones críticas de un proceso consecutivamente. Si un proceso se cae en la región crítica, afectará a la ejecución del otro, pero si ocurre en <resto de código>, no afecta al otro. Sin embargo, no garantiza la exclusión mutua ya que dependemos de la rapidez relativa de ejecución de turno[0]:=true y turno[1]:=true.

Tercer intento

En el algoritmo 5 se cumplen todas las condiciones saludables. Pero una caída en la región crítica bloquea al otro proceso. En una ejecución estrictamente paralela, los dos procesos se ponen a true, de forma que ninguno de ellos puede continuar (se quedan en el bloqueo del do {nada} esperando indefinidamente). Esto se denomina interbloqueo.

Cuarto intento

En este algoritmo no se produce interbloqueo ya que ambos blabñabña analizar código en casa calentito. Se sigue dando ese proble

Intento final

En este caso un proceso puede ejecutar consecutivamente regiones críticas. Además, se previene el interbloqueo y aunque un proceso se caiga fuera de la región crítica no afecta al otro. Si se cae en región crítica, el otro queda bloqueado. Se cumple exclusión mutua.

4.4.2. Algoritmo de Peterson

Este algoritmo garantiza la exclusión mutua debido al uso de una variable compartida, turno, que se chequea cada vez.

Algoritmo 3 Algoritmo de Dekker, segundo intento

```
Program EMADS2

var turno:array[0..1] of boolean;

process 0
begin
   while true
   begin
while turno[1] do {nada};
   turno[0]:=true;
   <sección crítica>
   turno[0]:=false;
   <resto de código>
   end;
end;

process 1
begin
   while true
   begin
while turno[0] do {nada};
   turno[1]:=true;
   <sección crítica>
   turno[1]:=false;
   <resto de código>
   end;
end;

begin
   turno[0]:= false;
   turno[1]:= false;
   parbegin
      process0;
      process1;
   parend;
end;
```

Algoritmo 4 Algoritmo de Dekker, tercer intento

```
Program EMADS3

var turno:array[0..1] of boolean;

process 0
begin
   while true
   begin
turno[0]:=true;
while turno[1] do {nada};
   <sección crítica>
   turno[0]:=false;
   <resto de código>
   end;
end;

process 1
begin
   while true
   begin
turno[1]:=true;
while turno[0] do {nada};
   <sección crítica>
   turno[1]:=false;
   <resto de código>
   end;
end;

begin
   turno[0]:= false;
   turno[1]:= false;
   parbegin
      process0;
      process1;
   parend;
end;
```

Algoritmo 5 Algoritmo de Dekker, cuarto intento

```
Program EMADS4

var turno:array[0..1] of boolean;

process 0
begin
   while true do
   begin
turno[0]:=true;
while turno[1] do
   begin
turno[0]:=false;
delay(random)
turno[0]:=true;
   end
<seccion crítica>
turno[0]:=false;
<resto de código>
   end;
end;

process 1
begin
   while true
   begin
turno[1]:=true;
while turno[0] do;
  begin
     turno[1]:=false;
     delay(random)
     turno[1]:=true;
  end;
<región crítica>
turno[1]:=false;
<resto de código>
   end;
end;

begin
   turno[0]:= false;
   turno[1]:= false;
   parbegin
      process0;
      process1;
   parend;
end;
```

Algoritmo 6 Algoritmo de Dekker, intento final

```
Program EMADSF

var
   señal : [0..1] of boolean;
turno    : 0..1;

process cero
begin
   while true do
   begin
señal[0]:=true;
while señal[1] do
begin
   if (turno=1) then
   begin
señal[0]:=false;
while turno=1 do {nada};
señal[0]=true;
   end
end
<seccion crítica>
turno:=1;
señal[0]:=false;
<resto de código>
   end;
end;

process uno
begin
   while true do
   begin
señal[1]:=true;
while señal[0] do
   begin
if (turno=0) then;
   begin
señal[1]:=false;
while turno=0 do {nada};
   señal[1]:=true;
   end;
end;
<región crítica>
turno:=0;
señal[1]:=false;
<resto de código>
   end;
end;

begin
   turno:=0;
   señal[0]:=false;
   señal[1]:=false;
   parbegin
      process cero;
      process uno;
   parend;
end;
```

Algoritmo 7 Algoritmo de Peterson

```
Program EMADSF

var
   señal : [0..1] of boolean;
turno    : 0..1;

process cero
begin
   while true do
   begin
señal[0]:=true;
while señal[1] do
begin
   if (turno=1) then
   begin
señal[0]:=false;
while turno=1 do {nada};
señal[0]=true;
   end
end
<seccion crítica>
turno:=1;
señal[0]:=false;
<resto de código>
   end;
end;

process uno
begin
   while true do
   begin
señal[1]:=true;
while señal[0] do
   begin
if (turno=0) then;
   begin
señal[1]:=false;
while turno=0 do {nada};
   señal[1]:=true;
   end;
end;
<región crítica>
turno:=0;
señal[1]:=false;
<resto de código>
   end;
end;

begin
   turno:=0;
   señal[0]:=false;
   señal[1]:=false;
   parbegin
      process cero;
      process uno;
   parend;
end;
```

4.4.3. Semáforos

Los semáforos pueden contemplarse como variables que tienen un valor entero sobre las que se definen las tres operaciones siguientes:

- Un semáforo puede inicializarse con un valor no negativo.
- La operación WAIT decrementa el valor del semáforo. Si el valor se hace negativo, el proceso que ejecuta WAIT queda bloqueado.
- La operación SIGNAL incrementa el valor del semáforo. Si el valor no es positivo, se desbloquea a un proceso bloqueado previamente por una operación WAIT.

Veamos cuál sería la implementación de un semáforo.

Algoritmo 8 Semáforos

```
type semaforo: record
   contador : entero;
   cola : list of proceso;
end;

var s: semaforo;

wait(s);
   s.contador:=s.contador -1;
   if s.contador < 0
then begin
   poner este proceso en cola
   bloquear este proceso
end

signal(s);
   s.contador := s.contador +1;
   if s.contador <=0
then begin
   quitar proceso P de la cola
   poner proceso P en la cola de preparados
end;
```

En cualquier instante, el valor de s.contador puede interpretarse como sigue. Si s.contador es mayor o igual que 0, indica el número de procesos que pueden ejecutar WAIT sobre el semáforo sin quedar bloqueados. Si s.contador es menor estricto que 0, su valor absoluto es el número de procesos bloqueados en la cola del semáforo.

ejemplo Inicializamos un semáforo a 3. Llega tres procesos y ejecutan WAIT, que hace que baje el contador a 0. El cuarto, coloca el contador a -1 y se queda bloqueado. Llega un quinto, coloca el contador a -2 y se bloquea. Si un proceso hace SIGNAL, el contador se ve incrementado en uno.

Las primitivas WAIT y SIGNAL son atómicas, es decir, no pueden ser interrumpidas y cada rutina puede considerarse indivisible.

Un semáforo binario es aquél que sólo puede tomar los valores 0 y 1. Su implementación es muy sencilla y podemos realizarla a partir del algoritmo 8

Algoritmo 9 Semáforos binarios

```
type semaforo: record
   valor : 0..1;
   cola : list of proceso;
end;

var s: semaforo;

wait(s);

   if s.valor = 1 then
s.valor = 0;
   else begin
   poner este proceso en cola
   bloquear este proceso
   end;
end;


signal(s);

   if s.cola está vacía
s.valor = 1;
   else begin
   quitar proceso P de la cola
   poner proceso P en la cola de preparados
   end;
end;
```

Si ni hay ningún proceso en la cola de bloqueados, SIGNAL 'activa' el interrputor, es decir, coloca s.valor a 1.

Lo siguiente que veremos es cómo implementar una región crítica con un semáforo.

A continuación veremos una solución al problema de productores-consumidores con la restricción de disponer de un *buffer* finito o limitado. Podemos considerar que este problema tiene las siguientes especificaciones:

- El número de elementos contenidos en un momento dado en el *buffer* vendrá dado por la siguiente expresión.

$$nDatos = Producidos - Consumidos$$

  Teniendo en cuenta que el *buffer* tendrá una capacidad finita, siempre se verificará que $0 \leq nDatos \leq capacidad$.

Algoritmo 10 Implementación de una sección crítica con semáforo

```
program exclusion_mutua

var s:semaforo;

procedure P(i : integer)
begin
   repeat
wait(s);
<sección crítica>
signal(s);
<resto de código>
   forever
end;

begin
   s:=1;
   par begin
P(1);
P(2);
   par end
end;
```

- Un proceso productor sólo podrá ejecutarse cuando $nDatos < capacidad$ y un proceso consumidor cuando $nDatos > 0$. Una cuestión adicional es que se va a implementar el *buffer* de una forma circular. Contaremos con dos punteros denotados por ENT y SAL que apuntarán respectivamente al siguiente hueco donde producir y al siguiente elemento que consumir.

La idea es tener n productores y n consumidores a la vez para que produzcan y consuman en concurrencia.

4.4.4. Monitores

Un monitor es, esencialmente, una colección de datos y de procedimientos para su manipulación junto con una secuencia de inicialización. Las variables de datos globales son generalmente privadas al monitor por lo que sólo son accesibles a los procedimientos de éste. Los procedimientos del monitor podrán ser públicos o privados. Un monitor puede considerarse como una estructura estática que se activa únicamente cuando alguno de sus procedimientos públicos es llamado por un proceso en ejecución y se dice, entonces, que el proceso en cuestión entra o tiene acceso al monitor.

Solamente un proceso puede estar ejecutándose en el monitor en un instante determinado. Una estructura de datos compartida puede así protegerse situándola dentro de un monitor que ofrecerá un servicio de exclusión mutua para dicha estructura. Para que resulten útiles en el procesamiento concurrente, los monitores deben incluir alguna herramienta de sincronización de forma que se impida el acceso al monitor a un proceso cuando otro está ejecutando dentro de él. Esta sincronización se consigue por medio

Algoritmo 11 Productores/consumidores

```
program prod_consum_sem
type
   dato =  ....
var
   buffer  : array[capacidad] of dato;
   puedeProducir,puedeConsumir : Semaforo_General;
   pmutex,cmutex  : SemaforoBinario;
   ent,sal  : 1..capacidad;

procedure productorX
var
   pDato : dato;
begin
   while true do
   begin
wait(puedePrducir); {comprobamos que podemos producir}
pDato:=producir(); {producimos}
wait(pmutex);
buffer[ent]:=dato; {hay que llegar aquí en exclusión mutua para que no
  haya conflictos. Por eso hay antes un semáforo binario}
ent:=(ent mod capacidad)+1; {aquí está la idea de buffer circular}
signal(pmutex); {liberamos la sección crítica}
signal(puedeConsumir); {hacemos esto porque por cada elemento que deje en el buffer
habrá un consumidor que lo pueda consumir y eso ha de quedar reflejado}
   end;
end; { productorX }

procedure consumidorZ
var
   cDato : dato
begin
   while true do
   begin
wait(puedeConsumir); {sólo si hay datos en el buffer podremos consumir,
    en caso contrario, nos quedaremos bloqueados}
wait(cmutex);
cDato:=buffer[sal]; {consumimos en exclusión mutua}
sal:=(sal mod capacidad)+1; {dejamos la variable sal al siguiente slot del buffer}
signal(cmutex);  {habilitamos de nuevo el que se pueda consumir}
signal(puedeProducir); {nuevos productores podrán aprovechar el hueco dejado}
consumir(cDato); {consumimos el datos propiamente}
   end;
end; { consumidorZ }

begin
   ent:=1;
   sal:=1;
   signal(cmutex);
   signal(pmutex);
   puedeConsumir:=0;
   for(i=1 to capacidad)do
signal(puedeProducir);  {por cada slot, habilitamos un proceso productor}
   par begin
Productor1
.
Productorn
Consumidor1
.
Consumidorn
   par end;
end;
```

de unas variables de condición que se incluyen en el monitor y que sólo son accesibles dentro de él. A diferencia de los semáforos, estas variables de condición no toman valor true o false ni ninguno otro sino que constituyen una cola con procesos en espera. Para trabajar con estas variables de condición se establecen las mismas primitivas que para los semáforos. Así, la primitiva **wait** suspende la ejecución de un proceso bajo una determinada condición con lo que el monitor quedaría disponible para ser utilizado por otro proceso. La primitiva **signal** reanudaría la ejecución de un proceso bloqueado en una determinada variable de condición.

Una característica básica de los monitores es proporcionar control sobre las operaciones realizadas sobre los elementos compartidos con el fin de prevenir actuaciones dañinas o sin significado. De esta forma, se limitan los tipos de actuaciones proporcionando un conjunto de procedimientos de manipulación fiables y bien probados. Los monitores avanzan un paso en este sentido haciendo los datos críticos accesibles indirecta y exclusivamente mediante un conjunto de procedimientos públicos disponibles.

Los monitores encapsulan los datos utilizados por los procesos concurrentes y permiten su manipulación sólo por medio de operaciones adecuadas y sincronizadas. Nunca existirá peligro de actualización inconsistente por entrelazamiento de llamadas concurrentes ya que los procesos del monitor siempre se ejecutarán en exclusión mutua.

Veremos a continuación el problema de productores/consumidores utilizando un monitor.

4.4.5. Paso de mensajes

Los mensajes constituyen un mecanismo relativamente sencillo y adecuado tanto para la comunicación como para la sincronización entre procesos que trabajan en entornos centralizados o entornos distribuidos. En esencia, un mensaje es una colección de información que puede ser intercambiada entre un proceso emisor y un proceso receptor.

Un mensaje puede contener datos, órdenes de ejecución e, incluso, código a transmitir entre dos o más procesos. Aunque, en general, el contenido de un mensaje quedará dividido en dos campos bien separados; Por un lado, la cabecera -que habitualmente tiene un formato fijo para cada sistema operativo- y, por otro lado, el cuerpo del mensaje -que contiene el mensaje en sí y cuya longitud puede variar incluso dentro de un mismo SO. Las operaciones de mensaje típicas proporcionadas por el SO son: **enviar (send)** y **recibir (receive)**. Las implementaciones del envío y recepción de mensajes pueden diferir en una serie de detalles pero todas ellas mantienen la importancia de un conjunto de cuestiones que son:

1. Denominación o direccionamiento: Utilizar una denominación directa significa que cuando se invoca una operación de mensaje cada emisor debe designar el receptor específico y a la inversa, cada receptor debe designar a la fuente desde la cual desea recibir el mensaje. send(B, mensaje) y receive(A,mensaje).

   Un método alternativo es la comunicación indirecta de mensajes donde éstos son enviados y recibidos a través de dispositivos especializados dedicados a este fin. Estos dispositivos se suelen denominar *buzones* debido a su modo de funcionamiento.

Algoritmo 12 Implementación de un monitor

```
{implementación de un monitor}
wait_signal:monitor
begin
   ocupado:boolean;
   libre:condition;
procedure mwait
begin
   if ocupado then libre.wait;
   ocupado:=true;
end; { mwait }

procedure msignal
begin
   ocupado:=false;
   libre.signal;
end; { msignal }
{cuerpo del monitor}
ocupado:=false;
end;


process uno
[...]
wait_signal.mwait
<sección crítica>>
waitsignal.wsignal
[...]


process dos
[...]
wait_signal.mwait
<sección crítica>
wait_signal.msignal
[...]
```

Algoritmo 13 Implementación del problema de Productores/Consumidores utilizando un monitor

```
program m_prod_cons

const
   capacidad = ...
var
   b1 : monitor
begin
   buffer:array[1..capacidad] of dato;
   ent,sal:(1..capacidad);
   cuenta:(0..capacidad);
   puedeProducir,puedeConsumir:condition;

procedure mdepositor(pDato dato){público}
begin
   if(cuenta=capacidad) then puedeProducir.wait; {se bloquea el proceso que intenta producir}
   buffer[ent]:=pDato;
   ent:=(ent mod capacidad)+1;
   cuenta:=cuenta +1;
   puedeConsiumir.signal;
end; { mdepositor }

procedure mconsumir(var cDato : dato){público}
begin
   if cuenta=0 then puedeConsumir.wait;
   cDato:=buffer[sal];
   sal:=(sal mod capacidad)+1;
   cuenta:=cuenta -1;
   puedeProducir.signal;
end; { mconsumir }
{inicialización}
ent:=1;
sal:=1;
cuenta:=0;

end;{m_prod_cons}
```

send(buzon,mensaje) receive(buzon,mensaje).

2. Copia: El intercambio de mensajes entre dos procesos, por definición, transfiere el contenido del mensaje desde el espacio de direcciones del emisor al espacio de direcciones del receptor. Esto puede lograrse copiando todo el mensaje de un espacio de direcciones a otro, o bien, simplemente, pasando un puntero al mensaje entre los dos procesos, en otras palabras, la transferencia del mensaje puede ser por valor o por referencia. En sistemas distribuidos que no disponen de memoria compartida la copia es inevitable mientras que en sistemas centralizados el compromiso está entre seguridad y eficiencia.

3. Intercambio síncrono *vs* intercambio asíncrono: Cuando un intercambio de mensajes es síncrono, tanto el emisor como el receptor deben proceder juntos para completar la transferencia. En sistemas síncronos la operación de envío es bloqueante, es decir, cuando un proceso emisor desea enviar un mensaje para el que no se ha emitido el correspondiente **receive()** por parte del proceso receptor. El emisor quedará bloqueado hasta que el receptor acepte el mensaje. Como consecuencia sólo puede haber un mensaje pendiente como máximo por cada emisor/receptor . En el intercambio asíncrono de mensajes, el emisor no queda bloqueado cuando no hay un **receive()** pendiente. El envío en un sistema asíncrono se implementa haciendo que el SO almacene temporalmente los mensajes pendientes hasta que se emita el correspondiente **receive()**. Como resultado, el proceso emisor puede continuar su ejecución después de enviar un mensaje y no necesita quedar bloqueado.

   Un problema común a ambas implementaciones es el *aplazamiento indefinido* que tiene lugar cuando un mensaje se envía pero nunca se recibe. Una aproximación para resolver este problema consiste en especificar un límite de tiempo dentro del cual debe completarse un intercambio de mensajes particulares.

4. Longitud: La última cuestión en cuanto al diseño de los mensajes es si éstos deberían tener una longitud fija o variable. Los mensajes de tamaño fijo producen generalmente una baja carga de proceso en virtud de que permiten que los *buffers* del sistema sean también de tamaño fijo, lo que hace su asignación bastante sencilla y eficaz. El problema es que los mensajes utilizados para comunicación tienen tamaños variados, con lo que los diferentes tamaños deberán ser adaptados a un único tamaño fijo con el consiguiente desaprovechamiento de parte del espacio para los mensajes más cortos.

4.4.6. Soluciones hardware para la exclusión mutua

Antes de discutir estrategias hardware específicas debemos indicar que todas ellas pueden caracterizarse de forma genérica como, esencialmente, *pesimistas* u *optimistas*.

Las **estrategias pesimistas** tienden a suponer el peor caso y a defenderse contra él tomando mediadas relativamente drásticas que terminan por limitar la concurrencia del sistema.

Algoritmo 14 Implementación del problema de Productores/Consumidores utilizando mensajes

```
program prod_cons_mensj

type
   mensaje =  record...;

const
   capacidad = ...;
   nulo  = ...; {mensaje vacío}

var
   i : integer

process productorX
var
   pmsj : mensaje
begin
   while true do
   begin
receive(puedeProducir,pmsj);
pmsj:=producir();
send(puedeConsumir,pmsj);
<resto de código>
   end;
end;
process consumidorZ
var
   cmsj : mensaje;
begin
   while true do
   begin
receive(puedeConsumir,cmsj);
consumir(cmsj);
send(puedeProducir,cmsj);
<resto de código>
   end;
end;

{proceso general}
begin
   crear_buzon(puedeProducir);
   crear_buzon(puedeConsumir);
   for i:=1 to capacidad do:
send(puedeProducir,nulo); {inicializamos a nulo}

   initiate
   productor1
   [...]
   productorn
   consumir1
   [...]
   consumidorn
end;
end;
```

Suponiendo la actualización de una variable global compartida o de una variable semáforo, una solución pesimista típica puede actuar del siguiente modo:

1. Bloquear todo aquello que presumiblemente puede interrumpir, de modo que nada pueda interferir.
2. Actualización de la variable global.
3. Desbloqueo de la parte del sistema bloqueada en el primer paso.

Las **estrategias optimistas** se basan en la suposición de que lo probable es que no haya conflicto o que se experimenten muy pocos por parte de los usuarios del recurso compartido. Consiguientemente se suelen consentir referencias bastante permisivas a los datos compartidos.

Cuando se presumen conflictos las estrategias optimistas mantienen la integridad del sistema descartando las actualizaciones invalidadas por los procesos concurrentes contendientes, esto generalmente implica una parcial vuelta atrás en el estado del sistema y rehacer parte de las actualizaciones afectadas.

Una solución optimista típica puede estructurarse del siguiente modo:

1. Lectura de la variable global y preparación de la actualización local tentativa basada en ese valor de forma que la variable global permanece durante ese tiempo accesible a los restantes usuarios.
2. Comparar el valor actual de la variable global con el valor utilizado para preparar la actualización tentativa. Si el valor de la variable global no ha sido modificado se estará en condiciones de aplicar la actualización local tentativa a la variable global, en caso contrario, es decir si la variable global ha sido modificada mientras tanto haciendo que quede obsoleta la actualización preparada se descartará dicha actualización tentativa y deberá repetirse el paso 1.

Primera solución: habilitación/deshabilitación de interrupciones

La idea básica de este mecanismo sigue el principio de que el recurso debe ser obtenido en exclusividad por el proceso que desea entrar a su sección crítica y posteriormente liberado para su uso por el resto de procesos. Esto puede conseguirse mediante la siguiente secuencia:

```
DI; deshabilitar interrupciones
sección crítica
EI; habilitar interrupciones
```

El propósito de deshabilitar interrupciones es evitar cualquier interferencia de interrupción durante la sección crítica. Este mecanismo resulta sencillo pero implementa una filosofía pesimista en cuanto que impide toda concurrencia cada vez que un proceso va a utilizar un recurso compartido. De esta forma se deshabilita no sólo a los procesos

que compiten para acceder al recurso, sino también a todos los procesos restantes que nada tienen que ver con ellos. Además este mecanismo no es en absoluto adecuado para sistemas multiprocesadores donde no garantiza la exclusión mútua.

Instrucción Comprobar y Fijar (Test and Set)

Esta instrucción está pensada para dar soporte hardware directo a la exclusión mutua. Está diseñada para resolver conflictos entre procesos contendientes haciendo posible que solo uno de ellos reciba permiso para entrar en su sección crítica. La idea básica es fijar una variable de control global al valor libre cuando esté disponible el recurso compartido asociado a ella. Cada proceso que desee acceder a este recurso debe obtener el correspondiente permiso ejecutando la instrucción *Test and Set* con la variable de control asociada como operando. el funcionamiento de la instruccion *Test and Set* es como sigue:

```
TS operando;
```

1. Se compara el valor del operando con `ocupado` y se modifican los códigos de condición correspondientes para que reflejen el resultado de esta comparación.
2. Si el estado del operando es libre entonces TS lo pone a `ocupado`.

Una característica fundamental es que los dos pasos descritos se realizan en una única operación indivisible.

Vemos la implementación de la operación `wait` de los semáforos con una sentencia TS.

```
wait: TS S
      BNF[1] wait
      RETURN
```

Instrucción Comparar e Intercambiar (compare and swap)

La instrucción compare and swap (CS) sigue una estrategia optimista para resolver el problema de la exclusión mutua. Esta instrucción no está pensada para implementar directamente operaciones de semáforos, sino para la actualización consistente de variables globales en presencia de actividad concurrente. CS tiene tres operandos.

- Un registro que contiene el valor de la variable global en el cual se basa la actualización tentativa (VIEJOREG).
- Un registro que contiene la actualización tentativa (NUEVOREG).
- La dirección de la variable global en cuestión (VARGLOB).

CS consta de las siguiente secuencia de pasos que se ejecuta como una operación única e indivisible:

Veamos un ejemplo de uso con TS, otro con CS

[1] `BNF` es Branch if Not Free (saltar si no está ocupado)

Algoritmo 15 Actualización consistente de variables globales en actividad concurrente

```
CS VIEJOREG, NUEVOREG, VARGLOB


COMPARA VIEJOREG,VARGLOB
FIJA LOS CÓDIGOS DE CONDICIÓN
SI VIEJOREG==VARGLOB
   VARGLOB=NUEVOREG
SI NO
   VIEJOREG=VARGLOB
```

Algoritmo 16 Ejemplo de uso con TS

```
#la actualización tentativa suma 3

T&S: TS mutex

BNF T&S
MOVE ACC,suma
ADD ACC, #B
MOVE suma, ACC
MOVE mutex,1
```

Algoritmo 17 Ejemplo de uso con CS

```
#la actualización tentativa suma 3

T&S: TS mutex

BNF T&S
MOVE ACC,suma
ADD ACC, #B
MOVE suma, ACC
MOVE mutex,1
```

El problema de la cena de los filósofos

Tenemos una cena muy especial a la que hemos invitado a la cena de los filósofos, que o comen o piensan, pero nunca a la vez. El problema es que les hemos preparado espaguetis y necesitan dos tenedores para comer, pero sólo le han colocado uno. El filósofo que consiga un tenedor de un compañero, podrá comer.

La implementación es como sigue:

Algoritmo 18 Cena de Filósofos

```
program cenaFilosofos
var
   tenedor : array[1..4] of semaforoBinario; {número de filósofos}
   i       : integer;

process filosofo(i:integer)
begin
   while true do
   begin
wait (tenedor[i]);
wait (tenedor[(i+1) mod 4]);
comer;
signal (tenedor[i]);
signal (tenedor[(i+1) mod 4]);
pensar;
   end;
end;

begin
   for (i==1 to 4) do
tenedor[i]==1;
   parbegin
     filosofo(1);
   .
   .
   .
     filosofo(4)
   parend
end;
```

Si los cuatro filósofos cogen sus tenedores a la vez, se quedarán interbloqueados. Para resolver este problema, dejaremos sólo $n - 1$ filósofos a la mesa. De esta forma, siempre habrá al menos uno que pueda comer. ¿Cómo afecta esto a la implementación? Añadimos un semáforo general para controlar los filósofos que entran en la habitación.

Algoritmo 19 Cena de Filósofos con habitación

```
program cenaFilosofos
var
   tenedor   : array[1..4] of semaforoBinario; {número de filósofos}
   i    : integer;
   habitacion : semaforoGeneral;


process filosofo(i)
begin
   while true do
   begin
wait (habitacion);
wait (tenedor[i]);
wait (tenedor[(i+1) mod 4]);
comer;
signal (tenedor[i]);
signal (tenedor[(i+1) mod 4]);
signal (habitacion);
pensar;
   end;
end;

begin
   habitacion = 3;
   for (i==1 to 4) do
tenedor[i]==1;
   parbegin
     filosofo(1);
   .
   .
   .
     filosofo(4)
   parend
end;
```

ejercicio propuesto en un sistema tenemos tres procesos A,B y C. La región crítica de cada proceso consiste simplemente en escribir "soy el proceso X". Queremos conseguir que el orden de ejecución sea el siguiente: ABC, ABC, ABC, ABC, ABC. (eso es lo que ha de salir por pantalla).

Algoritmo 20 Solución al ejercicio propuesto "ABC" para los procesos a, b y c

```
program Ejabc
var
   a,b,c  : semaforoBinario;

procedure PA
begin
   while true do
begin
   wait(a);
   printf("Soy A");
   signal(b);
end;
end;
procedure PB
begin
   while true do
begin
   wait(b);
   printf("Soy B");
   signal(c);
end;
end;
procedure PC
begin
   while true do
begin
   wait(c);
   printf("Soy C");
   signal(a);
end;
end;

begin

   a=1;
   b=0;
   c=0;

   parBegin
   PA;
   PB;
   PC;
   parend
end;
```

5 Interbloqueos

5.1. Principios de interbloqueo

Una situación de interbloqueo tiene lugar cuando ninguno de los procesos que compiten por los recursos del sistema o interactúan entre sí puede avanzar por carecer de algún recurso o esperar a que se produzca algún tipo de evento.

5.1.1. Recursos reutilizables

Un recurso reutilizable es aquel que puede ser utilizado por un proceso y no se agota por hacer uso del mismo, los procesos obtienen unidades de estos recursos y tras utilizarlas las liberan para que puedan ser reutilizadas por otros procesos. Como ejmplos de recursos reutlizables tenemos el procesador, la memoria principal y los dipositivos E/S.

5.1.2. Recursos consumibles

Un recurso consumible es aquel que puede ser producido y consumido. Normalmente no hay límite en el número de recursos consumibles de un tipo particular. Así un proceso productor podrá liberar cualquier número de recursos consumibles. Las únicas restricciones en este sentido vendrán impuestas por la capacidad de almacenamiento temporal del sistema. Cuando un proceso consume un recurso de este tipo la parte consumida queda excluida del sistema. Ejemplos típicos son: interrupciones, señales y mensajes.

A continuación veremos una secuencia que muestra la posibilidad de interbloqueo entre procesos que utilizan recursos consumibles.

```
P1
recibir(P2,M)
enviar(P2,M)

P2
recibir(P1,M)
enviar(P1,M)
```

A continuación veremos otra secuencia que produce interbloqueo entre procesos que utilizan recursos reutilizables.

```
P1
solicitar(A)
solicitar(B)
```

```
P2
solicitar(B)
solicitar(A)
```

5.1.3. Condiciones de interbloqueo

Deben darse tres condiciones para que se produzca interbloqueo

1. Que exista acceso a algún recurso en exclusión mutua.
2. Que un proceso pueda retener los recursos que le han sido asignados mientras espera que se le asignen los que necesitan.
3. Que ningún proceso pueda ser obligado a abandonar los recursos que retenga.

Estas tres condiciones de interbloqueo son condiciones necesarias pero no suficientes, es decir, pueden producirse tales situaciones y que el sistema no evolucione a un interbloqueo[1] .

Para que se produzca interbloqueo, debe darse una cuarta condición que consiste en la existencia de una cadena cerrada de procesos donde cada uno de los cuales retiene al menos un recurso de los que necesita el siguiente proceso de la cadena para continuar su ejecución. A esta condición se le denomina *espera circular.*

5.2. Prevención de interbloqueos

La estrategia de prevención consiste, a grandes rasgos, en diseñar un sistema de manera que esté excluida a priori la posibilidad de interbloqueo. Los métodos para prevenir interbloqueos son de dos tipos:

- Métodos indirectos; que consisten en prevenir o impedir la aparición de alguna de las tres condiciones iniciales de interbloqueo.
- Métodos directos; que consisten en evitar la aparición del círculo vicioso de espera, es decir, la cuarta condición.

A continuación se examinarán las técnicas empledas para impedir cada una de las cuatro condiciones.

1. Condición de exclusión mutua: No puede anularse, ya que si el acceso a un recurso requiere exclusión mutua, el SO debe soportarlo.
2. Retención y espera: Puede prevenirse exigiendo que todos los procesos soliciten todos los recursos que necesitan a un tiempo y bloqueando al proceso hasta que todos los recursos puedan concedérsele simultáneamente. Esta solución resulta ineficiente por dos factores:

[1] De hecho, estas tres situaciones se dan con mucha frecuencia de forma natural.

a) En primer lugar, un proceso puede estar bloqueado durante mucho tiempo esperando que se le concedan todas sus solicitudes de recursos cuando, de hecho, podría haber avanzado con sólo alguno de los recursos.

b) Los recursos asignados a un proceso pueden permanecer sin usarse durante periodos considerables de tiempo durante los cuales se priva a otros procesos de acceder a estos recursos.

3. Condición de No Apropiación: Esta condición puede prevenirse de varias formas:

   a) Si a un proceso que retiene ciertos recursos, se le deniega una nueva solicitud, dicho proceso deberá liberar los recursos que poseía y solictarlos de nuevo junto con el recurso que le ha sido denegado[2].

   b) Si un proceso solicita un recurso que está retenido por otro proceso, el SO puede expulsar al segundo proceso y exigirle que libere el recurso. Este último esquema evita el interbloqueo sólo si dos procesos no pueden tener la misma prioridad con respecto a la posesión de un recurso.

4. Cículo vicioso de espera: Esta condición puede prevenirse definiendo una ordenación lineal en los tipos de recursos. Si a un proceso se le han asignado recursos de tipo R sólo podrá realizar peticiones posteriores sobre los recursos de los tipos siguientes a R en la ordenación. Para implementar esta estrategia se asocia un índice a cada tipo de recurso de forma que si un proceso solicita el recurso R_i y a continuación el recurso R_j debe cumplirse que $i < j$.

5.3. Detección de interbloqueos

Las estrategias de detección de interbloqueos no limitan el acceso a los recursos ni restringen las acciones de los procesos como ocurría con las estrategias de prevención de interbloqueos, mediante las estrategias de detección de interbloqueos se concederán los recursos que los procesos necesitan siempre que sea posible. Periódicamente el SO ejecuta un algoritmo que permite detectar la condición de círculo de espera. Los algoritmos de detección más comunmente utilizados son algoritmos basados en grafos dirigidos. El control del interbloqueo puede llevarse a cabo tan frecuentemente como las solicitudes de los recursos o con una frecuencia menor, dependiendo de la probabilidad de que se produzca interbloqueo.

La comprobación en cada solicitud de recurso tiene dos ventajas:

- Conduce a una pronta detección.
- El algoritmo es relativamente simple puesto que está basado en cambios incrementales del estado del sistema.

[2] Ésta parece la más interesante de las dos.

Por otro lado, la frecuencia de comprobación consume un tiempo de CPU considerable.

Una vez detectado el interbloqueo alguna estrategia de recuperación, las siguientes técnicas son posibles enfoques enumerados por orden de sofisticación.

1. Abandono de todos los procesos bloqueados: esta es la técnica más utilizada por los SSOO.

2. Retroceder cada proceso interbloqueado hasta algún punto de control definido previamente y volver a ejecutar todos los procesos. El riesgo de esta solución es que puede volver a producirse el interbloqueo inicial, sin embargo el no determinismo del procesamiento concurrente posibilita que esto no vuelva a ocurrir.

3. Abandonar sucesivamente los procesos bloqueados hasta que deje de haber interbloqueo. Para ello, se seguirá un criterio de mínimo coste. Después de abandonar cada proceso se debe ejecutar de nuevo el algoritmo de detección para ver si todavía existe interbloqueo.

4. Apropiación sucesiva de recursos hasta que deje de haber interbloqueo por parte de alguno de los procesos. Se debe emplear también una solución basada en el coste y hay que ejecutar de nuevo el algoritmo de detección después de cada apropiación. Un proceso que pierda un recurso porque otro se lo apropie deberá retroceder hasta un momento anterior a la adquisición de este recurso.

Para las dos últimas estrategias, el criterio de selección podría ser uno de los siguientes consistentes en escoger el proceso con:

1. La menor cantidad de tiempo de procesador consumido hasta el momento. Se aplica el criterio de mínimo coste ya que el proceso hay que repetirlo.

2. Menor número de líneas de salida producidas hasta el momento.

3. Mayor tiempo restante estimado.

4. Menor número de recursos asignados hasta el momento.

5. Prioridad más baja.

El algoritmo de detección de interbloqueo no se ejecuta cada vez que un proceso solicita un recurso, sino con una frecuencia menor.

5.4. Predicción de interbloqueo. Algoritmo del banquero

En la predicción de interbloqueo, se decide dinámicamente si la petición actual de un recurso podría, de concederse, llevar potencialmente a un interbloqueo. La predicción de interbloqueo necesita, por tanto, conocer las peticiones futuras de recursos. A continuación describiremos los dos enfoques para la predicción del interbloqueo.

5.4.1. Negativa de iniciación de procesos

Este enfoque consiste en no iniciar un proceso si sus demandas de recursos pueden llevar a un interbloqueo. Consideremos un sistema con n procesos activos y m tipos diferentes de recursos. Definiremos los vectores y matrices siguientes:

1. Vector de recursos : $V_R = \begin{pmatrix} R_1 \\ \vdots \\ R_m \end{pmatrix}$ denota R_i denota la cantidad del recursos i que hay en el sistema.
2. Vector de recursos disponibles: $AV_R = \begin{pmatrix} AV_1 \\ \vdots \\ AV_m \end{pmatrix}$ donde AV_i denota la cantidad de recurso i disponible en un momento dado en el sistema.
3. Matriz demanda $C_R = \begin{pmatrix} C_{11} & \cdots & C_{n1} \\ \vdots & \ddots & \vdots \\ C_{1m} & \cdots & C_{nm} \end{pmatrix}$ donde C_{ij} la exigencia máxima que el proceso i tiene del recursos j[3].
4. Matriz de asignación $A_R = \begin{pmatrix} A_{11} & \cdots & A_{n1} \\ \vdots & \ddots & \vdots \\ A_{1m} & \cdots & A_{nm} \end{pmatrix}$ donde A_{ij} denota la cantidad de recurso j que tiene el proceso i en un instante determinado. Es decir, el total de recursos que tiene asignado un proceso vendrá dado por el vector $A_{iR} = \begin{pmatrix} A_{i1} \\ \vdots \\ A_{im} \end{pmatrix}$ donde i identifica al proceso.

Después de definir estas matrices y vectores, deben cumplirse las siguientes condiciones.

1. $\forall j \in [1,m]\ \sum_{k=1}^{n} A_{kj} + AV_j = R_j$. El número de unidades de un recurso es la suma de las unidades utilizadas y las unidades ociosas.
2. $\forall i \in [1,n], \forall k \in [1,m]\ C_{ik} \leq R_k$. La demanda de ningún proceso sobre ningún recurso puede superar la cantidad del recurso.
3. $\forall i \in [1,n], \forall k \in [1,m]\ A_{ik} \leq C_{ik}$. Ningún proceso puede tener asignada más cantidad de un recurso que la que especifica su demanda máxima.

[3]Se lee por columnas: La columna 1 indica las exigencias máximas del recurso1 respecto de todos los recursos. Se lee por filas: La fila 1 indica la exigencia de todos los procesos sobre el recurso 1.

Teniendo en cuenta estas restricciones, un proceso $n+1$ sólo puede arrancarse si

$$\forall j \in [1,m] \ \sum_{k=1}^{n+1} C_{kj} \leq R_j$$

en palabras; la suma de las demandas máximas de todos los procesos (incluido el candidato a nuevo) en relación a un recurso no debe superar nunca la cantidad de ese recurso en el sistema. De esta forma, nos aseguramos de que en el peor de los casos (todos piden demanda máxima) todos podrán ser satisfechos.

ejercicio propuesto en un sistema tenemos tres procesos A,B y C. La región crítica de cada proceso consiste simplemente en escribir "soy el proceso X". Queremos conseguir que el orden de ejecución sea el siguiente: ABC, CAB, ABC, CAB, ABC. (eso es lo que ha de salir por pantalla).

Algoritmo 21 Algoritmo para el ejercicio propuesto

5.4.2. Negativa de asignación de recursos

Esta estrategia también se denomina algoritmo de Banquero y fue propuesta por primera vez por Dijkstra. Se comienza definiendo los conceptos de *estado* y *estado seguro*. El estado de un sistema en un momento dado es simplemente la asignación actual de recursos a los procesos, así pues, el estado estará formado por los vectores de recursos y de recursos disponibles, y por las matrices de demanda y asignación definidas previamente. Teniendo esto en ecuenta, un estado seguro es un estado en el cual existe al menos un orden en que todos los procesos pueden ejecutar hasta el final sin generar un interbloqueo. Un estado inseguro es, lógicamente, todo estado que no sea seguro.

Ejemplo Supongamos el siguiente sistema con las matrices y vectores de recursos y procesos que siguen:

$$C_R = \begin{array}{c|cccc} & P_1 & P_2 & P_3 & P_4 \\ R_1 & 3 & 6 & 3 & 4 \\ R_2 & 2 & 1 & 1 & 2 \\ R_3 & 2 & 3 & 4 & 2 \end{array} ; A_R = \begin{array}{c|cccc} & P_1 & P_2 & P_3 & P_4 \\ R_1 & 1 & 6 & 2 & 0 \\ R_2 & 0 & 1 & 1 & 0 \\ R_3 & 0 & 2 & 1 & 2 \end{array} ; AV_R = \begin{array}{cc} R_1 & 0 \\ R_2 & 1 \\ R_3 & 1 \end{array} ; R = \begin{array}{cc} R_1 & 9 \\ R_2 & 3 \\ R_3 & 6 \end{array}$$

Podemos comprobar que las restricciones que se imponían más atrás se cumplen. ¿Es un estado seguro? De momento, vemos que P_1 no podría continuar ya que requiere 3 unidades del recurso R_1 y éste no puede ofrecérselas. P_2 sí puede continuar y cuando termine, liberará sus recursos dejando la matriz de disponibles en $AV_R = \begin{array}{cc} R_1 & 6 \\ R_2 & 2 \\ R_3 & 3 \end{array}$. Ahora P_1 sí puede ejecutarse sin problemas y al terminar libera

sus recursos. Ahora P_3 también puede ejecutarse y de nuevo, al terminar, libera sus recursos utilizados, permitiendo a P_4 ejecutarse y terminar. Como hemos encontrado *un* camino por el que todos los procesos se ejecutan al final de un tiempo, estamos ante un estado seguro.

La estretegia de predicción de interbloqueo consiste en asegurar que el sistema esté siempre en un estado seguro. Para conseguir esto, cuando un proceso realiza una solicitud de un recurso o de un conjunto de r ecursos, se supone que la solicitud se le concede, a continuación se actualiza el estado del sistema para que refleje la nueva situación y se determina si en esa nueva situación, el sistema se encuentra en un estado seguro. Si el estado es seguro se concede la solicitud, mientras que si no lo es, el proceso solicitante es bloqueado hasta que concederle los recursos lleve a un estado seguro.

6 Gestión de memoria

6.1. Reubicación

El término reubicabilidad de programa se refiere a la capacidad de cargar y ejecutar un programa determinado en una posición arbitraria de memoria en contraposición a un conjunto fijo de posiciones especificadas durante la compilación de dicho programa. Las instrucciones de un proceso cargado en memoria contendrán referencias a posiciones de memoria de dos tipos:

1. Referencias a datos empleados en instrucciones de carga, almacenamiento y algunas instrucciones aritmético-lógicas.

2. Referencias a otras instrucciones empleadas fundamentalmente en bifurcaciones de control de flujo o en instrucciones de llamadas.

Ambos tipos de direcciones no serán fijas durante todo el periodo de permanencia del proceso en el sistema, sino que pueden variar si el proceso es suspendido y cargado posteriormente en memoria o, simplemente, si es desplazado dentro de ésta.

Distinguiremos, pues, entre dos tipos de direcciones:

1. Una dirección lógica o virtual es un identificador utilizado para referenciar información dentro del espacio de direcciones de un programa y, por tanto, es independiente de la asignación actual de datos a memoria debiéndose realizar una traducción a dirección física antes de poder realizar un acceso a memoria.

2. Una dirección física o absoluta designa una posición real de memoria física donde se almacena información en tiempo de ejecución

Dependiendo de cómo y cuándo tenga lugar la traducción del espacio de direcciones virtuales al espacio de direcciones físicas en un esquema de reubicación determinado, pueden considerarse dos tipos básicos de estrategias: Reubicación estática y reubicación dinámica.

Reubicación estática Implica generalmente que la reubicación es realizada antes o durante la carga del proceso en memoria. Las constantes (valores literales), los desplazamientos relativos al PC, no dependen de esta condición y no necesitan ser ajustados durante la reubicación.

Reubicación dinámica Implica que la correspondencia entre el espacio de direcciones virtuales y el espacio de direcciones físicas se efectúa en tiempo de ejecución. Usualmente con asistencia del hardware. Cuando el proceso en cuestión está siendo ejecutado, todas sus referencias a memoria son reubicadas durante la ejecución antes de acceder realmente a la memoria física. Este proceso se suele implementar por medio de registros base especializados.

A continuación veremos el mecanismo hardware que posibilita tanto la reubicación dinámica como la protección. Esta última consiste en impedir el acceso de un proceso a un espacio de direcciones que no le corresponde.

El **registro base** contiene la dirección de carga del proceso y el **registro límite** contiene la última dirección correspondiente al espacio de memoria asignado al proceso.

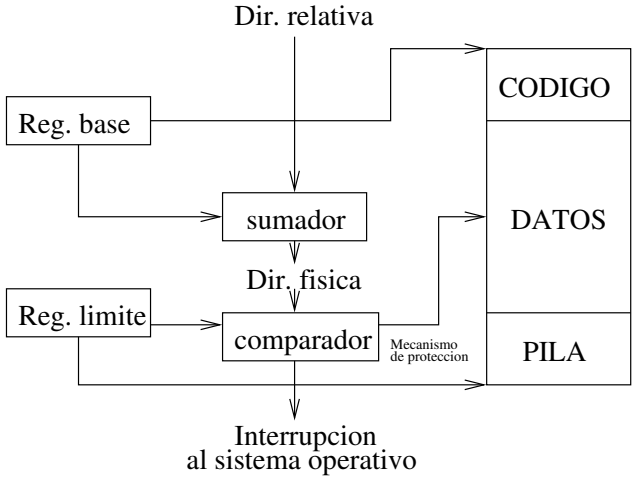


Figura 6.1: Esquema Hardware que da soporte a... de direcciones

6.2. Asignación de memoria con particiones fijas

En la mayoría de los esquemas de gestión de memoria se puede suponer que el SO ocupa una parte de la memoria principal y que el resto de la memoria está disponible para ser utilizada por los procesos de usuario. El esquema más sencillo de gestión de la memoria es dividirla en regiones con límites fijos. Una posibilidad es emplear particiones fijas de igual tamaño, en este caso cualquier proceso con tamaño menor o igual al tamaño de la partición puede cargarse en cualquier partición libre, si todas las particiones están ocupadas el SO puede sacar un proceso de alguna de ellas y cargar otro. La utilización de particiones fijas plantea dos dificultades:

1. Un programa puede ser demasiado grande para caber en una partición, en ese caso el programador debe diseñar el programa mediante superposiciones para que solo

una parte del programa estñe en memoria principal en cada instante. Cuando se necesita un módulo que no esa presente el programa de usuario debe cargar dicho módulo en la partición del programa superponiéndolo a los programas y datos que se encuentren en él.

2. El uso de la memoria principal es extremadamente ineficiente, ya que cualquier programa sin importar lo pequeño que sea ocupará una partición completa. Este fenómeno donde se desperdician espacio interno de una partición porque el bloque de proceso que es más pequeño que ella se denomina **fragmentación interna**.

Con particiones del mismo tamaño la ubicación de un proceso en memoria resulta trivial. Puesto que todas las particiones son de igual tamaño no importa que partición se utilice y se eligirá siempre la primera libre que se encuentre. Los problemas que presenta el uso de particiones fijas de igual tamaño pueden reducirse aunque no solventarse por medio del uso de particiones de distintos tamaños. El uso de estas particiones proporciona un cierto grado de flexibilidad a las particiones fijas, además ambos tipos de esquema de partición fija son relativamente simples y exigen un software de SO y una sobrecarga de proceso mínimos.

Con particiones de distinto tamaño hay dos maneras posibles de asignar los procesos a las particiones:

1. La forma más simple es asignar cada proceso a la partición más pequeña en la que quepa, en este caso hace falta una cola de planificación para cada partición. Esta cola albergará a los procesos cuyo destino es dicha partición. La ventaja de este enfoque es que los procesos se asignan de una forma en la que se desperdicia el menor espacio de memoria posible, sin embargo aunque esta técnica parec óptima desde el punto de vista de una partción individual no lo es desde el punto de vista del sistema global ya que puede darse la situación de que existan particiones sin utilizar que podrían ser aprovechadas por procesos que esperan en las colas de planificación de las particiones a las que han sido asignados.

2. Consiste en seleccionar la partición más pequeña disponible que pueda albergar al proceso.

La utilización de particiones fijas ya sean de igual o distintos tamaños plantea los siguientes problemas:

- El número de particiones especificadas en el momento de la generación del sistema limita el número de procesos activos en dicho sistema.

- Puesto que los tamaños de partición se programan en el momento de la generación del sistema los trabajos pequeños no hacen un uso eficiente del espacio de las particiones en un entorno en el que los requisitos básicos de almacenamiento de todos los procesos se conocen de antemano puede ser una técnica razonable, pero en la mayoría de los casos es ineficiente.

6.3. Asignación de memoria con particiones dinámicas

En este esquema las particiones van a ser variables en número y longitud. Cuando se trae un proceso a memoria se le asigna exactamente tanta memoria como necesita y no más.

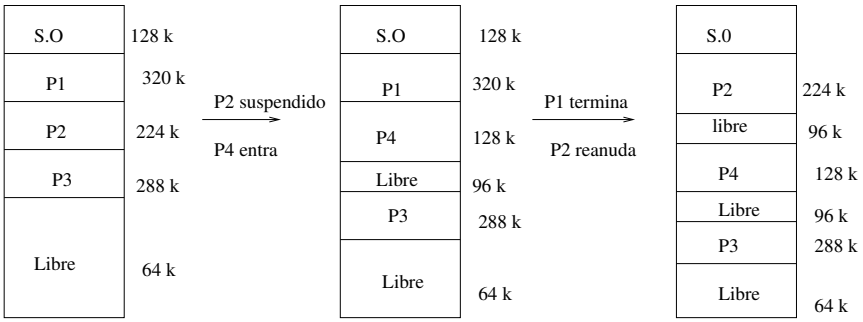


Figura 6.2: Fragmentación de memoria

Como muestra este ejemplo, a medida que pasa el tiempo, la memoria empieza a estar fragmentada y el rendimiento decae. A este fenónemo se le denomina *fragmentación externa* y se refiere al hecho de que la memoria externa a todas las particiones se fragmenta cada vez más. Una técnica para superar esta fragmentación es la compactación o defragmentación que consiste en desplazar los procesos para que estén contiguos de forma que toda la memoria libre esté junta en un bloque. La compactación requiere además la capacidad de reubicación dinámica, es decir, se debe poder mover un proceso de una región a otra de memoria principal sin invalidar sus referencias a memoria.

A la hora de ubicar procesos en memoria atañe al diseñador del SO decidir cómo se va a llevar a cabo esta ubicación. Los tres algoritmos que se pueden considerar son:

1. El Mejor Ajuste (*Best Fit*): Lo que se hace es elegir el bloque con tamaño más parecido al solicitado.
2. El Primer Ajuste (*First Fit*): Se recorre la memoria desde el principio y se escoge el primer bloque disponible que sea suficientemente grande.
3. El Siguiente Ajuste (*Next Fit*): Es similar a El Primer Ajuste pero se recorre la memoria desde el lugar de la última ubicación.

6.4. Asignación de memoria con paginación simple

Tanto las particiones estáticas como las dinámicas hacen un uso ineficiente de la memoria. Las primeras generan fragmentación interna mientras que las segundas generan fragmentación externa.

Supongamos la memoria principal particionada en trozos iguales de tamaño fijo relativamente pequeños y que cada proceso está dividido también en pequeños trozos de

tamaño fijo e igual a los de memoria. En tal caso, los trozos del proceso conocidos como páginas pueden asignarse a los trozos libres de memoria conocidos como marcos o marcos de página.

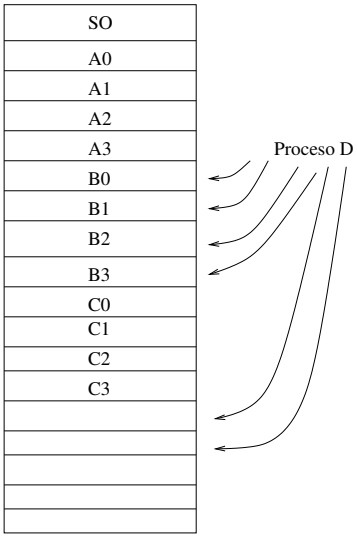


Figura 6.3: Asignación de memoria con paginación simple; marcos y palabras

Supongamos que el proceso B termina su ejecución y libera sus recursos de memoria. Entonces llega el proceso D que requiere 5 páginas de memoria. No hay ningún problema en asignarle los tres de B y dos del espacio libre. En este esquema la fragmentación interna constaría sólo de una fracción del último marco de página ocupado por el proceso y además no existe fragmentación externa puesto que siempre seremos capaces de aprovechar los huecos.

En los esquemas de partición de memoria basados en particiones fijas, las direcciones físicas se obtenían sumando las virtuales a la dirección de carga del proceso. En el esquema de gestión de memoria con paginación, sin embargo, ya no será suficiente con un simple registro para la traducción de direcciones. En su lugar, el SO mantiene una tabla de páginas para cada proceso. Cada una de estas tablas contiene una entrada por cada página del proceso por lo que se indexarán de forma fácil mediante el número de páginas comenzando siempre por la página 0. En cada entrada de la tabla de páginas se encuentra el número del marco de memoria que alberga la página correspondiente. Además, el SO mantiene una lista de marcos libres con todos los marcos de memoria que actualmente están vacíos y disponibles para las páginas.

Dentro del programa cada dirección lógica constará de un número de página y un desplazamiento dentro de la página y será también el hardware del procesador el que se encargue de realizar la traducción de direcciones lógicas a direcciones físicas.

Para aplicar convenientemente este esquema de paginación, el tamaño de página y, por tanto, el temaño de marco, deben ser una potencia de 2.En este caso, la dirección relativa definida en relación a la dirección de carga del proceso y la dirección lógica expresada

como un número de página y un desplazamiento son las mismas.

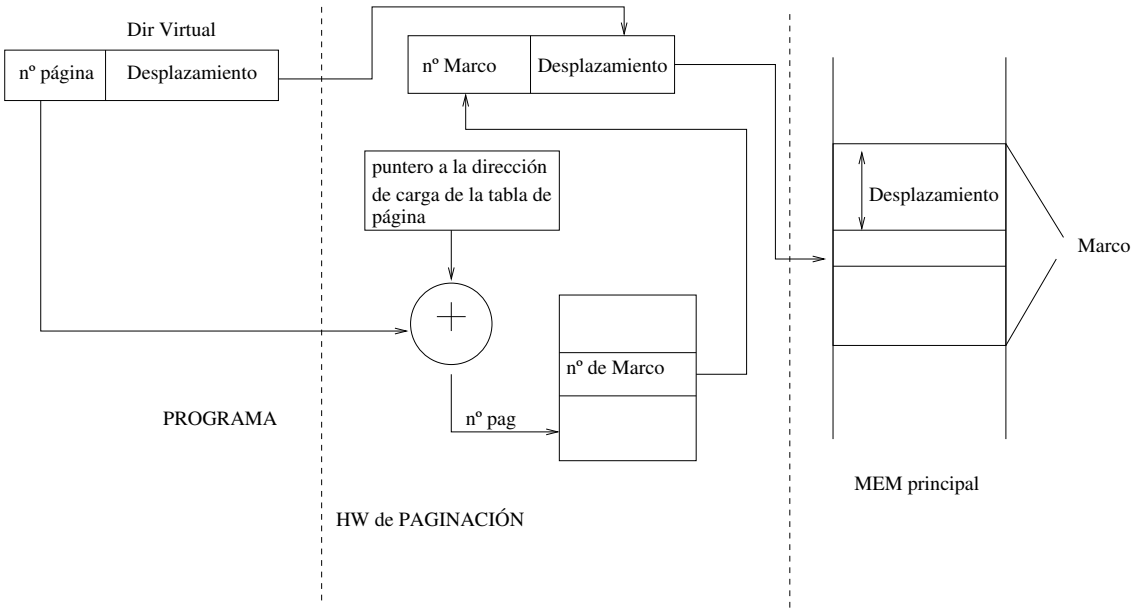


Figura 6.4: Esquema hardware para la paginación

Supongamos un sistema donde se emplean direcciones de 16 bits siendo el tamaño de página de $1k$ palabras. Con este tamaño de página se necesitan 10 bits para el campo de desplazamiento y los 6 restantes determinan el número de página. De esta forma, la capacidad total de la memoria es de 64 $k$palabras.

Las consecuencias de utilizar un tamaño de página *potencia de 2* son las siguientes:

1. El esquema de direccionamiento lógico es transparente al programador, al montador y al ensamblador. Cada dirección lógica de un proceso será, así, idéntica a su dirección relativa.

2. Resulta relativamente sencillo realizar una función hardware para llevar a cabo la traducción de direcciones dinámicas en tiempo de ejecución. Consideramos una dirección de $d = n + m$ bits en la que los n bits más significativos corresponden al número de página y los m bits menos significativos corresponden al desplazamiento dentro de la página. Para la traducción de direcciones hay que dar los siguientes pasos:

   a) Obtener el número de página a partir de los bits de la dirección lógica.

   b) Emplear ese número de página como índice en la tabla de páginas del proceso para determinar el marco k en que se alberga la página.

c) El comienzo de la dirección física del marco k será $k \cdot 2^m$[1] y la dirección física de la palabra referenciada será este número más el desplazamiento determinado por los m bits menos significativos de la dirección lógica.

Así pues, la paginación simple tal como se ha descrito es similar a la utilización de particiones fijas de idéntico tamaño con la diferencia de que las particiones son más pequeñas, un proceso puede ocupar más de una partición y las particiones correspondientes a un proceso no tienen porqué estar contiguas. A modo de resumen, mediante la paginación simple, la memoria principal se divide en pequeños marcos del mismo tamaño. Cada proceso se divide en páginas del tamaño del marco. Cuando un proceso se carga en memoria, se cargan todas sus páginas en marcos libres y se rellena su tabla de páginas.

A continuación veremos el esquema hardware que permite realizar la traducción de direcciones virtuales a físicas. figura 04

6.5. Asignación de memoria con segmentación simple

En segmentación, un programa y sus datos asociados se dividen en un conjunto de segmentos. No se impone que todos los segmentos de todos los programas tengan la misma longitud aunque sí existe una longitud máxima de segmento. Como en paginación, una dirección lógica segmentada consta de dos partes: número de segmento y desplazamiento dentro del segmento. Como consecuencia del empleo de segmentos de distinto tamaño, la segmentación resulta similar al esquema de asignación de memoria con particiones dinámicas. La diferencia con éste radica en que con segmentación un programa puede ocupar más de una partición y éstas no tiene por qué estar contiguas.

La segmentación elimina la fragmentación interna pero sufre de fragmentación externa, sin embargo, debido a que los procesos se dividen en un conjunto de partes más pequeñas la fragmentación externa será menor. Mientras que la paginación es transparente al programador, la segmentación es generalmente visible y se proporciona como una comodidad para la organización de programas y datos. Normalmente, el programador asigna los programas y datos a distintos segmentos.

Otra consecuencia del tamaño desigual de los segmentos es que no hay una correspondencia simple entre direcciones lógicas y direcciones físicas. Un esquema de segmentación hará uso de una tabla de segmentos para cada proceso y una lista de bloques libres en memoria principal. Cada entrada de la tabla de segmento deberá contener la dirección de comienzo del segmento correspondiente en memoria principal y también la longitud del mismo para asegurar el uso de direcciones válidas. Cuando un proceso pasa al estado de ejecución se carga la dirección de su tabla de segmentos en un registro especial del hardware de gestión de memoria.

Considerando una dirección lógica formada por $n + m$ bits, los n bits más significativos indican el número de segmento mientras que los m bits restantes indicarían el desplazamiento. Para la traducción de direcciones hay que dar los siguientes pasos:

[1] En nuestro ejemplo, tendríamos una secuencia de 0,1024,2048,...

1. Extraer el número de segmento de los n bits más significativos de la dirección lógica.
2. Emplear ese número de segmento como índice en la tabla de segmentos del proceso para determinar la dirección física de comienzo del segmento.
3. Comparar el desplazamiento expresado por los m bits menos significativos con la longitud del segmento. Si el desplazamiento es mayor que la longitud la dirección no es válida.
4. La dirección física final será la suma de la dirección física de comienzo de segmento más el desplazamiento.

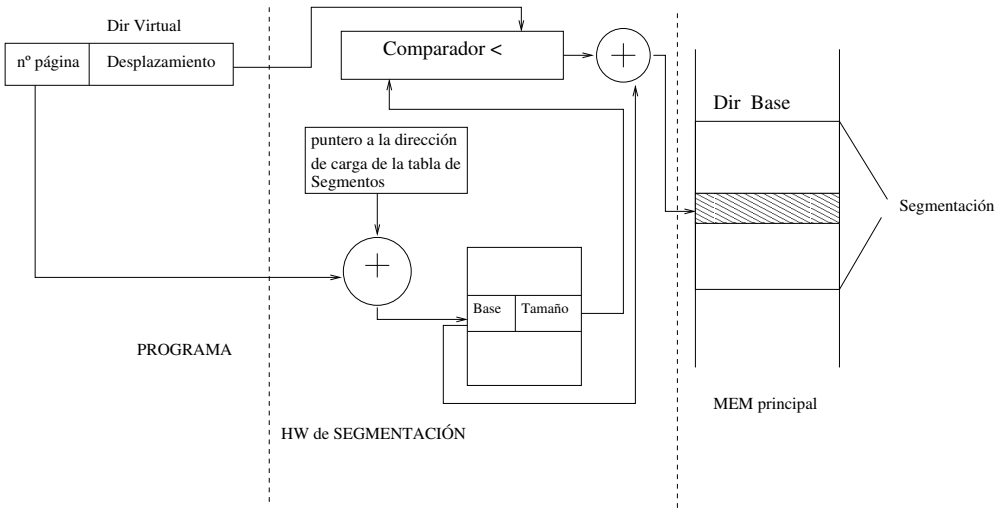


Figura 6.5: Asignación de memoria con segmentación simple

En este caso habrá que sumar ya que cada segmento tendrá tamaño variable no será obligatorio que comience en una potencia de 2.

6.6. Memoria virtual

6.6.1. Estructuras Hardware y de control

Las características fundamentales del avance introducido por el empleo de técnicas de paginación o segmentación son, fundamentalmente, dos.

1. Todas las referencias a memoria dentro de un proceso son direcciones lógicas que se traducen dinámicamente a direcciones físicas en tiempo de ejecución.
2. Un proceso puede dividirse en varias partes y no es necesario que estas partes se encuentren contiguas en memoria principal durante la ejecución.

Si estas dos características están presentes, no será necesario que todas las páginas o segmentos del proceso estén en memoria principal durante la ejecución. Éste es el concepto que da pie a lo que se conoce como Memoria Virtual.

Denominaremos *conjunto residente del proceso* a la parte de dicho proceso que está realmente en memoria principal. Cuando el proceso se ejecute todo irá bien mientras las referencias a memoria estén en posiciones pertenecientes al conjunto residente. Si el procesador encuentra una dirección lógica que no se ubica en memoria principal se produce lo que se denomina un *fallo de página* y se genera la correspondiente interrupción para que el SO bloquee al proceso y tome el control. El SO se encargará de traer a memoria principal el fragmento de proceso que contiene la dirección lógica que provocó el fallo de página. Una vez que este fragmento del proceso se ha cargado en memoria principal, el proceso bloqueado está en condiciones de continuar su ejecución y se pasa al estado de Listo o Preparado.

Las dos implicaciones principales de la utilización de memoria virtual son las siguientes:

1. Se puede mantener un mayor número de procesos en memoria principal.

2. Resulta posible que un proceso sea más grande que toda la memoria principal. De esta forma se elimina una de las limitaciones más notoria de la programación.

Como los procesos para ejecutar necesitan estar cargados en memoria principal, a esta memoria también se la denomina *memoria real*. Sin embargo, el programador o usuario percibe en potencia una memoria mucho mayor que es la memoria secundaria. A esta memoria también se la denomina Memoria Virtual.

6.6.2. Hiperpaginación y cercanía de referencias

En un estado estable, prácticamente toda la memoria principal estará ocupada con fragmentos de procesos, por lo que el procesador y el SO tendrán acceso directo a la mayor cantidad de proceso posible.

Así pues, cuando el SO traiga a memoria un fragmento, es posible que no exista en memoria principal espacio para alojarlo. En esta situación el SO deberá elegir un fragmento de igual o superior tamaño para ser expulsado a memoria secundaria y crear así el espacio necesario para alojar al nuevo fragmento. Si el fragmento expulsado va a ser referenciado justo después de su expulsión, deberá ser traido a memoria de forma inmediata. Demasiados intercambios de fragmentos entre memoria principal y secundaria conducen a lo que se denomina hiperpaginación o thrashing.

Estos argumentos se basan en el principio de cercanía de referencias que afirma que las referencias a los datos y al código del proceso tienden a agruparse y, por tanto, resulta válida la suposición de que durante periodos cortos se necesitarán sólo unos pocos fragmentos del proceso. Además será posible hacer predicciones inteligentes sobre qué fragmentos de un proceso se necesitarán en un futuro cercano y evitar así la hiperpaginación.

6.6.3. Memoria virtual con paginación y buffer de traducción adelantada (TLB)

El término Memoria virtual se asocia normalmente con sistemas que emplean paginación. Cuando se considera un esquema de memoria virtual basado en paginación se necesita la misma estructura que en paginación simple, es decir, la tabla de páginas. En este caso, sin embargo, las entradas de la tabla de páginas pasan a ser más complejas puesto que sólo algunas de las páginas de un proceso pueden estar en memoria principal.

Se empleará un bit en cada entrada de la tabla para indicar si la página correspondiente está en memoria principal o no. Si el bit indica que la página se encuentra en memoria, la entrada incluirá también el número de marco en el que se encuentra ubicada dicha página. A este bit se le conoce como *bit de presencia*I (P).

Otro bit de control necesario es el *bit de modificación* (M) que indicará si el contenido de la página correspondiente se ha alterado desde que la página se cargó en memoria principal. Si no ha habido cambios, no será necesario escribir la página cuando sea sustituida en el marco que ocupa actualmente.

Cada referencia a una dirección virtual puede generar dos accesos a memoria.

1. Para obtener la entrada de la tabla de páginas correspondiente.

2. Para obtener el dato deseado.

Así pues, se podría tener el efecto de doblar el tiempo de acceso a memoria. Para solucionar este problema, la mayoria de los esquema de memoria virtual hacen uso de una caché especial para las entradas de las tablas de páginas llamada *buffer de traducción adelantada* (Translation Lookaside Buffer). Esta caché contiene aquellas entradas de las tablas de páginas utilizadas hace menos tiempo. Dada una dirección virtual, el procesador examinará primero el TLB. Si la entrada de la tabla de página está presente, se obtiene el número de marco y se forma la dirección real. Por el contrario, si no se encuentra la entrada de la tabla de página buscada, el procesador emplea el número de página para buscar en la tabla de páginas del proceso. Si se encuentra activo el bit de presencia es que la página está en memoria principal y el procesador puede obtener el número de marco para formar la dirección real. El procesador, además, actualiza el TLB para incluir esta nueva entrada de la tabla de páginas.

Por último, si el bit de presencia no está activo, se produce un fallo de página. En este punto se abandona el ámbito hardware y se invoca al SO para cargar la página necesaria y actualizar la tabla de páginas.

6.6.4. Software del SO para la gestión de memoria virtual

Políticas de lectura

La política de lectura (FETCH) está relacionada con la decisión de cuándo se debe cargar una página en memoria principal. Sus dos opciones más comunes son la paginación por demanda y la paginación previa:

La paginación por demanda consiste en traer una página a memoria principal sólo cuando se hace referencia a una posición de esta página.

En paginación previa se cargan además de la página demandada, páginas secuencialmente consecutivas a ella. El principal atractivo de esta estrategia está en aprovechar el tiempo de búsqueda de la página demandada en memoria secundaria. Una vez encontrada, sólo tendremos que esperar un tiempo correspondiente a la latencia de giro del dispositivo de almacenamiento para acceder a las páginas secuencialmente contiguas.

Políticas de ubicación

La política de ubicación tiene que ver con determinar dónde va a residir una parte del proceso en memoria principal. En un sistema de segmentación puro, la política de ubicación es un aspecto muy importante de diseño, teniendo como posibles alterantivas las políticas de mejor ajuste, primer ajuste y siguiente ajuste.

Políticas de reemplazo

Cuando todos los marcos de memoria principal están ocupados y es necesario traer a memoria una nueva página para atender un fallo de página, al política de reemplazo se encarga de seleccionar la página a reemplazar de entre las que se encuentren actualmente en memoria. Todas las políticas tienen como objetivo que la página a reemplazar sea la que tenga una menor probabilidad de ser referenciada en un futuro cercano.

En la política de reemplazo se encuentran involucrados conceptos interrelacionados como los siguientes:

1. Número de marcos de página a asignar a cada proceso activo.
2. Si el conjunto de páginas candidatas para el reemplazo debe limitarse a las del proceso que provocó el fallo de página o abarcará a todos los marcos de páginas situadas en memoria principal.
3. De entre el conjunto de páginas candidatas, la página que debe elegirse en particular para el reemplazo.

Algunos de los marcos de memoria principal pueden estar bloqueados. Cuando un marco se encuentra bloqueado, la página cargada actualmente en él no puede ser reemplazada. La mayoría del núcleo del SO así como las estructuras de control son albergados en marcos bloqueados.

Para estudiar algunas de las políticas de algoritmos de reemplazo vamos a considerar un caso donde un proceso hace referencia hasta a cinco páginas distintas y el SO permite una asignación constante de tres marcos por proceso. La cadena de referencia a las páginas durante la ejecución del programa es la siguiente:

2	3	2	1	5	2	4	5	3	2	5	2

La primera política que vamos a ver es la Política Óptima. Esta Política selecciona

para reemplazar la página que tiene que esperar más tiempo hasta que se produzca la referencia siguiente. Se puede demostrar que esta política genera el menor número de fallos de página, sin embargo, este algoritmo resulta imposible de implementar porque requiere que el SO tenga un conocimiento exacto de los sucesos futuros. De todas formas, sirve como estándar para comparar los otros algoritmos.

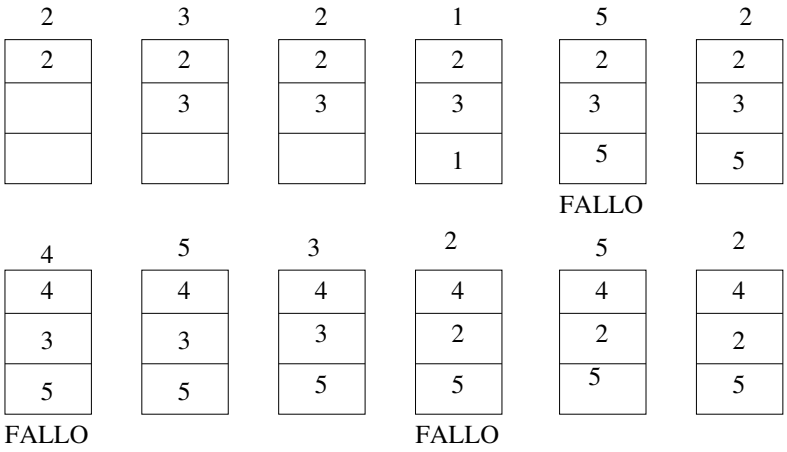


Figura 6.6: Esquema de Política Óptima para el caso propuesto

LRU (Last Recently Used): este algoritmo reemplaza la página de memoria que no ha sido referenciada desde hace más tiempo. Debido al principio de cercanía de referencias, ésta debería ser la página con menor probabilidad de ser referenciada en el futuro. La política LRU afina casi tanto como la política óptima pero plantea una gran dificultad de implementación.

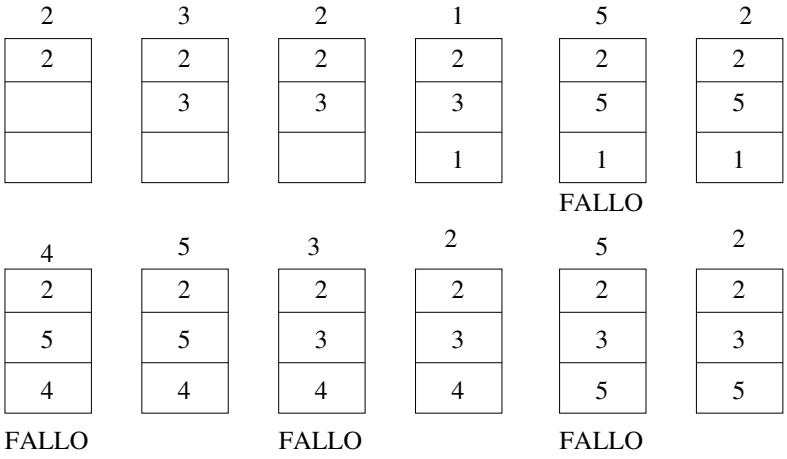


Figura 6.7: Esquema de Política LRU para el caso propuesto

FIFO: La Política FIFO trata los marcos asignados a un proceso como un *buffer*

circular y las páginas se suprimen de memoria según la técnica de espera circular Round-Robin. Todo lo que se necesita un puntero que circula a través de los marcos del proceso. Resulta, por tanto, una de las políticas de reemplazo más fáciles de implementar. La lógica que hay detrás de esta elección, además de su sencillez, es reemplazar la página que ha estado más tiempo en memoria.

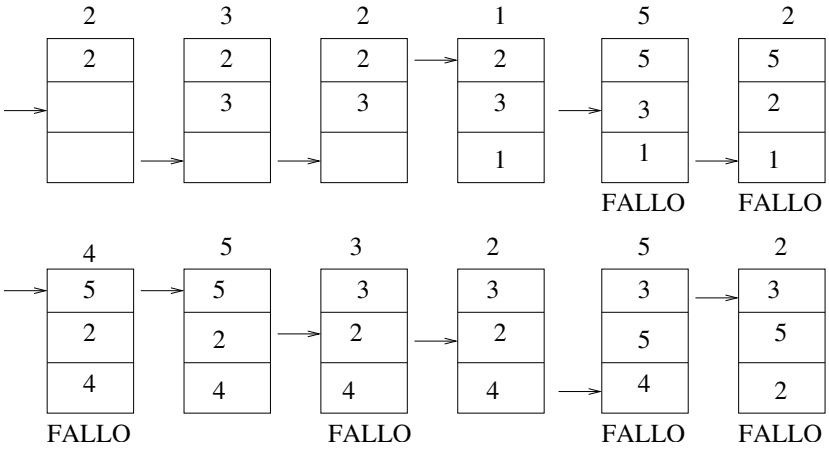


Figura 6.8: Esquema de Política FIFO para el caso propuesto

Política del Reloj: La forma más simple de esta política requiere asociar un bit adicional a cada marco, denominado bit de uso. Cuando se cargue una página por primera vez, este bit se pone a 0 y cuando se hace referencia posteriormente a la página el bit de uso se pone a 1. Para el algoritmo de reemplazo de páginas, el conjunto de marcos candidatos a ser reemplazado se considera como un *buffer* circular con un puntero asociado. El alcance es local si los candidatos son de un solo proceso y global si procede de toda la memoria. Cuando llega el momento de reemplazar una página, el SO recorre el *buffer* buscando un marco con el bit de uso a 0, eligiendo para reemplazar el primero que encuentre. Cada vez que se encuentra un marco con el bit de uso a 1, se pone a 0.

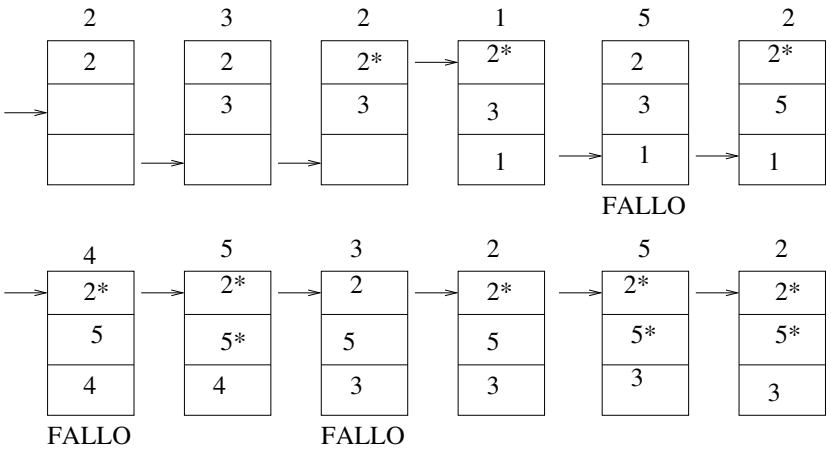


Figura 6.9: Esquema de Política del Reloj para el caso propuesto

Índice alfabético

acceso en exclusión mutua, 48
Algoritmo de Banquero, 78
arquitectura Von Neuman, 2

bit de modificación, 90
bit de presencia, 90
bit de uso, 93
bloque de control de proceso, 28
buffer de traducción adelantada, 90
buzones, 60

códigos de condición, 3
conjunto residente del proceso, 89
conmutación de procesos, 31
contador de programa, 4

división implicita de tareas, 16
division explicita de tareas, 16

El Mejor Ajuste, 84
El Primer Ajuste, 84
El Siguiente Ajuste, 84
espera circular, 74
estado bloqueado, 20, 21
estado de ejecución, 21
estado listo, 20, 21
estado nuevo, 21
estado preparado, 21
estado terminado, 21
Event-Driven, 42

fallo de página, 89
familia de procesos, 17
FCFS, 39
fragmentación externa, 84
fragmentación interna, 83

grado de multiprogramación, 12

interbloqueo, 50

llamadas al sistema, 1

Matriz de asignación, 77
Matriz demanda, 77
memoria principal, 2
MLQ, 43
multiprogramación, 12
multitarea, 12

paginación por demanda, 91
paginación previa, 91
palabra de estado del programa, 4
palabras de control, 4
Planificación por colas MultiNivel, 43
planificador, 35
Política Óptima, 91
prcoeso suspendido, 23
proceso, 15
proceso de sistema, 17
proceso de usuario, 16
produtividad, 35
Puntero de pila, 3
puntero de pila, 4
Puntero de segmento, 3

Registro índice, 3
registro base, 82
registro de estado, 4
registro de instrucción, 4
registro límite, 82
registros de códigos de condición, 3
registros de control y estado, 4

registros de datos, 3
registros de dirección, 3
reubicación dinámica, 82
reubicación estática, 81
Round-Robin, 39

seccion crítica, 47
sistemas multiprocesadores, 45

Tablas de memoria, 27
Test and Set, 66

unidad aritmético-lógica, 2
unidad central de proceso, 2
unidad de control, 2
unidad de Entrada/Salida, 2

Vector de recursos, 77
Vector de recursos disponibles, 77

Historia

0.5.0 - 15 de abril de 2004

- Primera versión pública a partir de los materiales y las clases preparados por Antonio Reinoso.

Las siguientes tareas merecen atención, a juicio de los editores y autores:

- Ampliar el tema de Programación concurrente
- Mejorar algunas figuras
- Incorporar una bibliografía

Creative Commons Deed

Attribution-NonCommercial-ShareAlike 1.0: Key License Terms

Attribution. The licensor permits others to copy, distribute, display, and perform the work. In return, licensees must give the original author credit.

Noncommercial. The licensor permits others to copy, distribute, display, and perform the work. In return, licensees may not use the work for commercial purposes – unless they get the licensor's permission.

Share Alike. The licensor permits others to distribute derivative works only under a license identical to the one that governs the licensor's work.

Whoever has associated this Commons Deed with their copyrighted work licenses his or her work to you on the terms of the Creative Commons License found here: Legal Code (the full license)

This is not a license. It is simply a handy reference for understanding the Legal Code (the full license) - it is a human-readable expression of some of its key terms. Think of it as the user-friendly interface to the Legal Code beneath. This Deed itself has no legal value, and its contents do not appear in the actual license.

Creative Commons is not a law firm and does not provide legal services. Distributing of, displaying of, or linking to this Commons Deed does not create an attorney-client relationship.

Learn how to distribute your work using this license

Manifiesto de Alqua

Origen y metas del proyecto

En 1999 fundamos el proyecto Alqua con el objetivo de promover la creación de un fondo de documentos libres de carácter científico que permita a cualquiera aprender con libertad.

Al constatar la duplicación de esfuerzos en la preparación de materiales didácticos para la física y con el deseo de compartir nuestros conocimientos, nos inspiramos en los principios de libertad que rigen el movimiento del software libre para establecer aquéllos de Alqua. Primero pensamos que lo que escribiésemos debería poder disfrutarse sin merma de libertad por las personas interesadas, y más tarde decidimos organizar nuestros esfuerzos para ayudar a otras personas que compartían nuestra visión a difundir sus saberes mediante un esfuerzo cooperativo.

Para hacer efectivos dichos principios decidimos que los documentos publicados deben ser libres en un sentido amplio: pueden reproducirse y distribuirse (gratuitamente o no, es irrelevante) pero también pueden modificarse y usarse como base para otros trabajos. A fin de evitar que estas libertades del lector-autor se restrinjan posteriormente, los documentos contienen una licencia que explica los derechos que posee y estipula que nadie que distribuya el documento, modificado o no, puede hacerlo de modo no libre.

Las ventajas de los documentos libres

Actualmente es ilegal compartir o modificar la mayoría del conocimiento científico en fuentes impresas, que suelen ser inaccesibles para la mayoría de los estudiantes y bibliotecas del mundo en virtud de su precio y se actualizan con poca frecuencia debido a su sistema de distribución tradicional.

En este contexto los documentos libres presentan ciertas ventajas.

Por una parte, en algunas disciplinas los documentos libres permiten facilitar el establecimiento de un sistema de mérito reduciendo las barreras de precio y disponibilidad. El modelo de desarrollo libre para la ciencia se apoya sobre las libertades de distribución y modificación. Éstas se ven favorecidas por el medio digital, así como por la concepción del conocimiento como un patrimonio comunitario. Todo lo anterior permite reducir el coste del documento a una cantidad marginal y anima a que lo mejor se combine con lo mejor para producir un resultado excelente a la vez que actualizado.

Por otra parte, en casos donde la evaluación del mérito es más subjetiva, los documentos libres pueden aportar una base sobre la que elaborar con un menor esfuerzo diferentes perspectivas doctrinales o estéticas, mutaciones, iteraciones y apuestas que incentivan la

creación como un aspecto más del disfrute de la obra.

En suma, los documentos libres fomentan un acceso a la cultura más justo y completo. Para algunos dominios del conocimiento científico el proceso de desarrollo libre facilita la recombinación, lo que permite la producción de obras muy sofisticadas y completas mientras que en otros ámbitos facilita la difusión de perspectivas plurales y la experimentación creativa.

Una nueva dinámica de creación y aprendizaje

Algunas personas que hemos conocido están interesadas por este modelo de colaboración, pero se preguntan qué clase de control tienen sobre sus documentos libres. La respuesta es sencilla: la licencia está diseñada de modo que a cada cual se le atribuya aquello de lo que es responsable y nada más. Para ello, se incluye en el documento una sección en la que se explica quién hizo qué y cuándo lo hizo.

Uno de los efectos más interesantes de introducir los documentos libres en el aula es que difuminan la frontera entre quien aprende y quien enseña. Los documentos libres son un puente para establecer contacto con una comunidad de interés mucho más vasta que la del centro educativo, permitiendo el aprendizaje continuo y fomentando una experiencia plural y transformadora: el criterio para participar en un documento es, solamente, hacerlo bien.

Un autor puede pensar que distribuir su documento bajo un copyright que restringe la libertad de copia es *más rentable* que otorgar mayores libertades. Esto no es necesariamente así, por varias razones.

En primer lugar, libre no quiere decir gratuito. Una editorial puede publicar un documento libre obteniendo beneficio de ello. De hecho, es una buena idea hacerlo dado lo agradable que resulta manejar un libro bien encuadernado. También los autores pueden aceptar una compensación de los lectores por su trabajo en un determinado documento.

En segundo lugar, la mayor parte de los autores son primeramente lectores. Cabe esperar, pues, que para la mayoría el enorme ahorro derivado del acceso a *muchos* documentos libres supere holgadamente el beneficio económico obtenido de *unos pocos* documentos no libres. La experiencia del software libre lo avala.

Finalmente, no se puede poner precio al beneficio social derivado de la existencia de documentos libres. Gracias a los derechos que uno posee sobre un documento libre puede adaptarlo para un curso académico eliminando lo que no es pertinente o es demasiado avanzado y complementando el tema con nuevas aportaciones, desde ejercicios o diagramas hasta apartados enteros.

Pensamos que las universidades u otras instituciones educativas podrían cumplir mejor su función social poniendo a disposición de la sociedad que las financia, en condiciones de libertad, su patrimonio más importante: el conocimiento.

El modelo de cooperación que proponemos (que anima al trabajo en equipo aunque no lo impone) permite abrir todas estas perspectivas y algunas más. Alqua intenta ofrecer los medios para esta tarea y relacionar, a través de los documentos libres, a los que tienen saberes que comunicar y a los que sienten curiosidad por dichos saberes.

Conclusión

Alqua tiene una tarea muy ilusionante y tan ambiciosa que sólo es factible en comunidad. Por ello, pedimos a las personas que forman parte de instituciones o empresas que colaboren con Alqua para que éstas apoyen económicamente el proyecto o patrocinen ediciones impresas y donaciones a las bibliotecas públicas. Ciertamente, los medios materiales son necesarios, pero inútiles si, a nivel particular, no contamos con tu participación como individuo, aprendiendo y enseñando, para que los documentos libres en marcha y otros nuevos alcancen los altos niveles de calidad a los que aspiramos.

Te invitamos a construir un patrimonio científico que nos pertenezca a todos.

Versión 2.0, marzo de 2003

El proyecto libros abiertos de Alqua

El texto que sigue es una explicación de qué es y cómo se utiliza un libro abierto y contiene algunas recomendaciones sobre cómo crear un libro abierto a partir de un documento de Alqua. Si estás leyendo estas páginas como anexo a otro documento, éste es casi con seguridad un *documento libre* de Alqua; libre en el sentido descrito en el manifiesto de Alqua y las directrices para documentos libres de Alqua . Si has obtenido dicho documento en un centro público, como una biblioteca, entonces es además un *libro abierto* de Alqua.

Qué son los libros abiertos

Los libros abiertos son ediciones impresas de los documentos libres de Alqua que se pueden obtener en las bibliotecas u otros centros públicos. La particularidad de los libros abiertos no reside en *qué contienen* (el contenido es el mismo que el de los libros descargados de la red) sino en *cómo pueden utilizarse*.

Al igual que los usuarios de Alqua a través de la red forman una comunidad de interés que aprende colectivamente leyendo los documentos, discutiendo sobre ellos y modificándolos para adaptarlos a propósitos muy variados, los lectores de una biblioteca constituyen también una comunidad. El ciclo de vida de un documento libre es de constante realimentación: las nuevas versiones son leídas, corregidas o quizá bifurcadas, lo que conduce a la publicación de nuevas versiones listas a su vez para un nuevo ciclo del proceso. ¿Por qué no abrir esa dinámica a la participación de comunidades que no se articulan en torno a la red?. No todos disponen del tiempo o los medios para participar efectivamente en el proceso de mejora de los documentos a través de la red, que es la aportación diferencial más importante de los libros libres respecto a los no libres. Por ello queremos poner a disposición de las bibliotecas *libros abiertos* que faciliten lo siguiente:

- El acceso de personas sin recursos informáticos al conocimiento que su estudio proporciona.
- La posibilidad de contribuir a la mejora de dichos documentos por parte de la amplísima comunidad de lectores de las bibliotecas, sin otro medio que un lápiz o una pluma.
- La formación de grupos de interés locales: compartir a través de un documento libre puede compartir su proceso de aprendizaje con personas interesadas por temas afines.

- La constitución, hasta en los centros que cuentan con una financiación más débil, de un fondo de documentos libres que cubra áreas del conocimiento que su presupuesto no permite afrontar.

¿Cómo puedo contribuir a los libros abiertos?

Sólo tienes que utilizarlos como si fuesen tuyos, pero recordando que compartes tu experiencia de aprendizaje con otras personas.

Por ejemplo, contrariamente a lo que harías con cualquier otro libro de la biblioteca puedes escribir en los márgenes de los libros abiertos tus propios comentarios: correcciones, aclaraciones, bibliografía relacionada... Intenta hacerlo ordenadamente, de modo que no interrumpa la lectura.

Si quieres compartir algún razonamiento más largo, puedes utilizar tus propias hojas e incorporarlas al final del documento, poniendo una nota donde corresponda. En este caso, no olvides firmar tu contribución con un nombre o seudónimo y, opcionalmente, una dirección de correo electrónico u otra forma de contacto.

Cualquiera que pueda participar a través de la red puede incorporar tus contribuciones a la versión que se distribuye en línea, con la ayuda de la comunidad de Alqua. De esta manera abrimos el mecanismo de colaboración a los lectores que no están acostumbrados al ordenador o prefieren no usarlo. La firma permite atribuir la autoría en el caso de que los cambios se incorporen y establecer contacto al respecto. Damos por hecho que al escribir tus aportaciones en un libro abierto estás de acuerdo con que sean libremente utilizadas (en el sentido descrito en las directrices para documentos libres ya mencionadas) y por lo tanto incorporadas a las sucesivas versiones digitales.

Los libros abiertos pueden ser editados de modo que se puedan separar sus hojas porque no hay inconveniente en que éstas sean fotocopiadas: no tenemos que usar la encuadernación como un modo de evitar la reproducción, puesto que no sólo no la prohibimos sino que animamos a ella. Por tanto, una vez que obtengas un ejemplar en préstamo puedes llevar contigo sólo la parte que estés utilizando.

Como lector, tu ayuda es necesaria no sólo para mejorar los documentos, sino para que existan: hace falta imprimir, encuadernar y donar a una biblioteca un documento libre de Alqua para que se convierta en un *libro abierto*.

Quienes tengan acceso a una impresora pueden ayudar a que los *libros abiertos* perduren en la biblioteca sustituyendo las partes deterioradas por el uso y actualizando periódicamente el documento impreso. Para facilitar la tarea a continuación proponemos un sistema de encuadernación modular.

¿Cómo puedo publicar un libro abierto?

Los pasos para publicar un libro abierto son los siguientes:

1. Imprimir la versión más actualizada del documento tal cual se distribuye en la página web de Alqua, http://alqua.org

2. Conseguir una encuadernación modular – sugerimos un archivador de anillas con una ventana o de portada transparente. Ello permite llevar consigo sólo la parte del libro que se está usando y añadir hojas con nuevas contribuciones.

3. Encuadernar el libro y situar el título, el autor y la clasificación decimal universal en su lomo y tapas.

4. Si puedes, adjuntar al archivador una copia del CD-ROM de documentos libres de Alqua .

5. Donarlo a la biblioteca y comunicar a Alqua la edición, escribiendo a librosabiertos@alqua.org .

Se trata de un proceso sencillo al alcance tanto de particulares como de bibliotecas y otras instituciones, con un coste marginal que no se verá significativamente incrementado por la conservación y actualización puesto que se puede mantener la encuadernación y sustituir solamente las páginas impresas.

En conclusión

El proyecto *libros abiertos*, consecuencia de los principios establecidos en el manifiesto de Alqua , persigue dotar a las bibliotecas de un fondo amplio y asequible de documentos libres y a la vez facilitar la participación de los usuarios en el proceso creativo del que son fruto.

Tu ayuda es esencial para que el proyecto alcance estos objetivos.

Sistemas Operativos

Pablo Ruiz Múzquiz

descripción

Este documento trata los conceptos fundamentales de los sistemas operativos. Se explica su desarrollo histórico y se abordan la gestión y planificación de procesos, la programación concurrente a un nivel introductorio y el análisis de la gestión de memoria.

requisitos

- Manejo elemental de un ordenador
- Conocimiento básicos de hardware.

http://alqua.org/libredoc/SSOO

Aprende en comunidad - http://alqua.org ◁

otros documentos libres

Variedades, tensores y física - Óptica electromagnética - Ecuaciones diferenciales ordinarias - Introducción a la física cuántica, segunda parte - Redes y sistemas - Sistemas Operativos - Geometría simpléctica - Física del láser - Análisis funcional - Geografía general de España (en preparación).

http://alqua.org/libredoc/

alqua, madeincommunity