T E M A 1

Contenido

TEMA 1

EL NÚCLEO

1.1. Introducción

El núcleo (*kernel*) del sistema operativo constituye el nivel más bajo de éste y proporciona un interface entre el hardware y el resto de niveles del sistema operativo (Figura 1). Su finalidad principal es constituir un entorno adecuado en el que se puedan ejecutar los procesos. Esto implica gestionar los recursos básicos del sistema y proporcionar servicios esenciales para los programas de aplicación y usuarios:

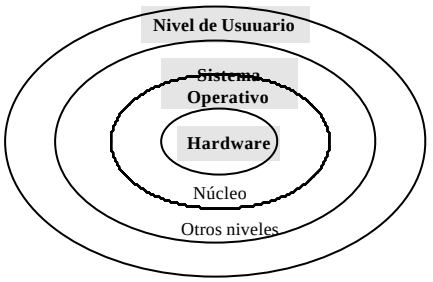

Figura 1.- *Estructura genérica de un sistema operativo.*

- Gestión de recursos básicos:
  - Gestión de memoria.
  - Creación de procesos y planificación de los mismos.
  - Mecanismos para la intercomunicación entre procesos.
  - Mecanismos básicos de entrada/salida.
- Servicios para las aplicaciones y usuarios:
  - Autentificación de usuarios y control de los accesos a los recursos por parte de los procesos.
  - Gestión de ficheros.

El núcleo está constituido directamente sobre el hardware, por lo que es la parte del sistema operativo más dependiente de la máquina y constituye, normalmente, la parte del sistema operativo que obligatoriamente debe de contener código en ensamblador. El resto del sistema puede ser escrito en un lenguaje de más alto nivel, lo que facilita el desarrollo y mantenimiento del mismo. Desde la aparición del sistema operativo UNIX, el lenguaje C suele ser el lenguaje en el que desarrollan los sistemas operativos.

1.2. Requisitos Hardware

El núcleo necesita unos requisitos mínimos de hardware para poder llevar a cabo sus funciones. Estos requisitos incluyen:

- Mecanismo de interrupciones.
- Protección de memoria.
- Repertorio de instrucciones reservadas.
- Reloj de tiempo real.

1.2.1. Mecanismo de Interrupciones

El hardware ha de suministrar un mecanismo por el que se pueda interrumpir el funcionamiento normal de la CPU. Las interrupciones son necesarias, fundamentalmente, porque permiten mejorar el aprovechamiento de la CPU mediante la aplicación de diversas técnicas (multiprogramación, tiempo compatido, etc.). Las principales fuentes de interrupciones son:

- Programas: como consecuencia de la ejecución de una instrucción, se producen algunas de las siguientes situaciones: *overflow*, divisiones por cero, referencias a direcciones de memoria que no pertenecen al espacio de direcciones del proceso, etc.
- Reloj del sistema: cada computador tiene un reloj que interrumpe a la CPU a intervalos regulares. De esta forma es posible, por ejemplo, utilizar políticas de planificación apropiativa (*preemptive scheduling*).
- Dispositivos de entrada/salida: los controladores de dispositivo generan interrupciones cuando se completa una operación o cuando se produce algún error.

- Fallos del hardware: errores de paridad de memoria, caida de la fuente de alimentación, etc.

El mecanismo de interrupciones funciona, generalmente, del siguiente modo. Cada tipo de dispositivo de entrada/salida tiene asociado una dirección de memoria conocida como vector de interrupciones, que contiene la dirección de una rutina que se ejecutará en el caso de que se produzca la interrupción correspondiente. Cuando un dispositivo envía una señal de interrupción al procesador, éste termina la ejecución de la instrucción en curso, determina la fuente de la interrupción, localiza su vector de interrupciones y ejecuta la rutina de tratamiento de la interrupción. Al terminar esta rutina el núcleo suele aplicar la política de planificación de procesos, lo que puede originar que el proceso que continue con su ejecución o bien que se ejecute otro diferente. Cuando un proceso es interrumpido, el núcleo ha de guardar toda la información necesaria para poder reanudar posteriormente su ejecución. Esta información incluye, como mínimo, el contador de programa y el resto de registros de la CPU.

Es posible que durante la ejecución de una rutina de tratamiento de una interrupción se produzca otra. Existen dos alternativas para tratar este tipo de situaciones. La primera consiste en desactivar las interrupciones mientras se está procesando una rutina de tratamiento de interrupciones. Si se producen nuevas interrupciones en esos periodos, normalmente se dejan pendientes y son tratadas cuando se vuelven a rehabilitar. La principal ventaja de este esquema es su simplicidad, ya que las interrupciones se atienden en orden estrictamente secuencial. El inconveniente es que no permite establecer prioridades entre interrupciones, lo que es necesario en determinadas situaciones. La segunda alternativa es, precisamente, el establecimiento de prioridades y permitir la interrupción de rutinas de tratamiento cuando se generan otras de mayor prioridad.

1.2.2. Protección de Memoria

La ejecución concurrente de varios procesos requiere que la memoria usada por cada uno de ellos esté protegida para evitar el acceso no autorizado por parte de otros procesos.

1.2.3. Repertorio de Instrucciones Reservadas

Con el fin de evitar que los procesos concurrentes interfieran entre ellos, parte de las instrucciones del computador deben de reservarse para su empleo exclusivo por parte del sistema operativo. Estas instrucciones llevan a cabo tareas como habilitar y deshabilitar interrupciones, acceder a los registros usados por el hardware de protección de memoria, realizar operaciones de entrada/salida y conmutar un procesador entre distintos procesos.

Para establecer cuándo está permitido utilizar estas instrucciones y cuándo no, la mayoría de los computadores trabajan en dos modos de funcionamiento: **modo supervisor** o **núcleo** y **modo usuario**. Las instrucciones reservadas únicamente pueden emplearse en modo supervisor. El cambio de modo usuario a modo supervisor se lleva a cabo al usar uno de los siguientes mecanismos:

- Llamada al sistema: un proceso hace una llamada al sistema operativo que requiere ejecutar alguna función que emplea instrucciones reservadas.
- Interrupción: ha ocurrido un suceso externo al proceso.
- *Trap*: Se da una condición de error en un proceso.

La vuelta a modo usuario desde el modo supervisor también se lleva a cabo me
reservada.

1.2.4. Reloj de Tiempo Real

Para poder lleva a cabo políticas de planificación es esencial disponer de un reloj hardware que produzca interrupciones a intervalos fijos de tiempo real. Típicamente, un reloj se compone de un cristal de cuarzo, un registro contador y un registro de carga. Un cristal de cuarzo tiene la propiedad de que bajo cierta tensión puede oscilar a intervalos periódicos con mucha precisión.

El funcionamiento de un reloj consiste en decrementar el registro contador cada vez que se produce una oscilación del cristal de cuarzo, de forma que cuando el contador llega a cero se produce una interrupción. A continuación, se puede optar por dos modos de funcionamiento. En el primer modo, el registro contador toma como valor inicial el contenido del registro de carga y cuando se produce la interrupción el reloj se para hasta que es puesto en marcha de nuevo de forma explícita. En el segundo modo, cuando el registro contador llega a cero y se produce la interrupción, automáticamente vuelve a

tomar el valor del registro de carga y comienza de nuevo. Estas interrupciones periódicas se denominan *ticks* de reloj.

1.3. Núcleos Monolíticos y Micronúcleos (*Microkernels*)

Las funciones concretas del núcleo van a depender en gran medida de la arquitectura del sistema operativo. Tradicionalmente, los sistemas operativos poseen un núcleo monolítico que suministra la mayoría de los servicios directamente, a través de las llamadas al sistema. El término monolítico implica, no tanto que el núcleo del sistema operativo sea de gran tamaño, sino que esté codificado de forma no modular. Como consecuencia, la alteración de cualquier componente del mismo para modificar o añadir nuevos servicios es difícil. El típico ejemplo de sistema operativo con núcleo monolítico es UNIX.

En la actualidad, se tiende hacia arquitecturas basadas en el concepto de micronúcleo (*microkernel*), que consiste en un núcleo que proporciona, básicamente, un conjunto de servicios esenciales. El resto de los servicios se implementan como procesos servidores a nivel de usuario. Estos servidores presentan una interfaz bien definido y son accedidos por el resto de los procesos mediante paso de mensajes.

La filosofía de los sistemas operativos basados en micronúcleo es mantener el núcleo lo más pequeño posible. Las ventajas de este esquema son que el sistema operativo es muy modular y que es fácil implementar, instalar y depurar nuevos servicios, ya que añadir o modificar un servicio no requiere parar el sistema y arrancarlo de nuevo. Como consecuencia, estos sistemas son muy flexibles, hasta el punto de que los usuarios que no estén satisfechos con un servicio aportado por el sistema pueden crearse el suyo propio.

La principal ventaja de los núcleos monolíticos reside en la eficiencia, ya que todas las funciones se llevan a cabo dentro del espacio de direcciones del núcleo. Por el contrario, una llamada al sistema en un micronúcleo suele consistir en el paso de un mensaje entre un proceso cliente y un proceso servidor, que se ejecutan en espacios de direcciones diferentes (y, posiblemente, en máquinas diferentes).

1.3.1. Arquitectura de un Micronúcleo

No existe una definición que establezca las funciones que debe de realizar un
todos incluyen la gestión de los recursos básicos (procesos, memoria, entrada/salida básica e intercomunicación entre procesos).

Los micronúcleos se diseñan para ser portables entre máquinas de arquitecturas diferentes, por lo que suelen estar codificados por lenguajes de alto nivel, como C y C++, y están estructurados en capas, de forma que los componentes dependientes del hardware se reducen a un mínimo nivel inferior.

La arquitectura de un micronúcleo se muestra en la Figura 2. Los componentes principales son los siguientes:

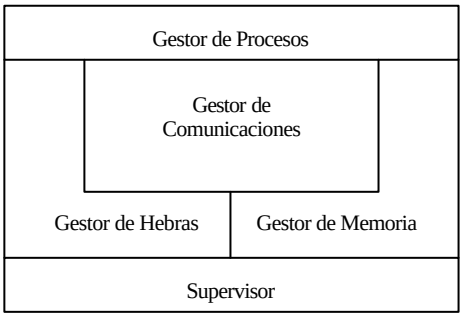

Figura 2.- *Arquitectura de un micronúcleo.*

- Gestor de Procesos: Gestiona todas las operaciones relativas a procesos, como creación, eliminación y planificación. Los servicios que ofrece suelen estar mejorados o aumentados mediante aplicaciones que se situan en un nivel superior, tales como emuladores de sistemas operativos.
- Gestor de hebras: Crea, planifica y sincroniza hebras. Una hebra es una actividad planificable que se asigna a un proceso. La política de planificación se puede definir en un módulo a nivel de usuario.
- Gestor de comunicaciones: Permite la comunicación entre las hebras de un procesos. En algunos casos, el núcleo incluye soporte para comunicaciones entre hebras de procesos remotos. En otros, el núcleo no es consciente de la presencia de otras máquinas, por lo que suele existir un determinado servicio en un nivel superior para comunicaciones externas.

- Gestor de memoria: Gestiona la memoria del sistema local. En algunos casos, el espacio de direcciones virtual está repartido entre varias máquinas, por lo que pueden existir gestores externos de memoria virtual.
- Supervisor: Es la parte que está justo por encima del hardware.

1.4. Procesos

El concepto de proceso es fundamental para la estructura de un sistema operativo, ya que que éste consiste básicamente en una colección de procesos. Estos se dividen entre aquellos que ejecutan el código del sistema y los que ejecutan el código de los usuarios.

Un proceso (también denominado tarea o trabajo) se suele definir como un programa en ejecución, en el cual las diferentes instrucciones se ejecutan de forma secuencial. Sin embargo, un proceso es algo más que un programa binario ejecutable. También se puede definir un proceso como un espacio de direcciones más una actividad. Un espacio de direcciones no es más que una colección de regiones de memoria virtual, cada una de las cuales tiene ciertas propiedades (tamaño, atributos - lectura, escritura o ejecución- y si puede crecer hacia arriba o hacia abajo del espacio de direcciones virtual). La actividad del proceso está representada por el contenido del registro contador de programa y los contenidos del resto de los registros de la CPU.

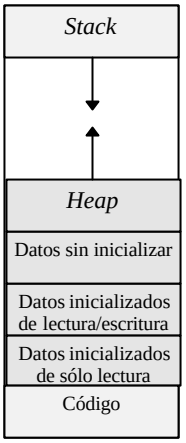

Figura 3.- Espacio de direcciones de un proceso en UNIX.

El espacio de direcciones de un proceso suele constar, al menos, de tres regiones: código, datos y pila (*stack*). La región de código contiene el código ejecutable del proceso, tiene tamaño fijo y es de sólo lectura. Esta última característica permite que una misma región de código pueda ser compartida por más de un proceso que sea instancia de un mismo programa. La región de datos contiene las variables globales del proceso y también se utiliza para asignar memoria a las estructuras de datos dinámicas, por lo que suele crecer hacia las direcciones virtuales más altas. Por último, la región de pila contiene datos temporales, tales como los parámetros, direcciones de retorno y variables locales de los procedimientos invocados. Crece hacia las direcciones virtuales inferiores.

Un proceso también es la unidad que puede poseer recursos. Los ficheros, memoria, canales de entrada/salida, etc. son asignados por el sistema operativo a los procesos.

1.4.1. Representación de los Procesos

El núcleo gestiona los procesos que se ejecutan en el sistema, por lo que debe de usar algún tipo de estructura de datos que constituya la representación física de todos los procesos. Cada proceso se puede representar mediante un **descriptor de proceso** (también conocido como bloque de control o vector de estado), que es una estructura que contiene información relevante acerca del proceso. La información que almacena un descriptor de proceso es la siguiente:

- Identificador de proceso, que constituye un nombre único por el que un proceso es reconocido de
- Estado del proceso.
- Información que hay que almacenar cuando se le quite la CPU al proceso y que es necesaria para poder reejecutar posteriormente dicho proceso:
  - Contador de programa.
  - Registros de la CPU (acumuladores, puntero de la pila, ...).
  - Información relativa a la gestión de memoria (registros base y límite, tabla de páginas, ...).
- Información para la planificación (prioridad del proceso, información requerida por el algoritmo de
- Información estadística (uso de CPU, ...).

Normalmente, existe una tabla en el núcleo llamada **tabla de procesos**, que permite acceder a los descriptores de los procesos.

1.4.2. Estados de un Proceso

Un proceso puede estar en tres estados básicos (Figura 4):

- **ejecución**: el proceso está siendo ejecutado por un procesador.
- **bloqueado**: un proceso no se puede ejecutar porque está esperando la ocurrencia de un determinado suceso (por ejemplo, la finalización de una operación de entrada/salida).
- **listo**: el proceso está listo para ser ejecutado por un procesador.

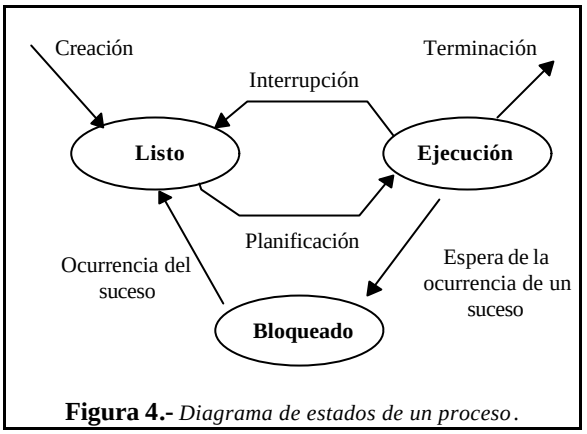

Figura 4.- *Diagrama de estados de un proceso*.

Algunos motivos por los que un proceso puede ser creado son los siguientes:

- Un nuevo usuario se conecta al sistema.
- El sistema operativo crea un proceso para que realice un determinado servicio.
- Un proceso puede crear otros procesos.
- Es el siguiente trabajo de un procesamiento por lotes (*batch*).

Las causas por las que un proceso termina pueden ser las siguientes:

- El proceso ejecuta una llamada al sistema indicándole que ha terminado.
- El proceso requiere más memoria de la que el sistema puede suministrarle.
- Se excede un límite de tiempo.
- Errores de protección, aritméticos, intento de ejecutar instrucciones no existentes o no permitidas en modo usuario, etc.
- El operador del sistema o el usuario decide eliminar el proceso.

Los motivos por las que un proceso se puede ser suspendido o bloqueado son:

- El sistema operativo lleva el proceso a disco (*swapping*) para permitir la ejecución de otros procesos.
- El sistema operativo suspende el proceso por varios motivos. Por ejemplo, el proceso puede estar involucrado en una situación de interbloqueo.
- El usuario puede suspender la ejecución de un proceso por motivos de depuración.
- Un proceso se puede ejecutar de forma periódica y debe esperar hasta que llega el siguiente intervalo.
- Un proceso hijo se puede suspender a causa de su proceso padre.

1.4.3. Planificación de Procesos

Una de las misiones del núcleo es la planificación de los procesos del sistema. Normalmente, existe un proceso dedicado a esa tarea denominado planificador de bajo nivel o *dispatcher*. Este proceso suele estar en estado bloqueado, y es despertado periódicamente, dependiendo del esquema de planificación, o cuando el proceso en ejecución no puede continuar y pasa a estado bloqueado.

1.5. Procesos y Hebras

En los sistemas operativos tradicionales, el concepto de proceso incluye dos características:

- Unidad a la que se le asigna recursos: espacio de direcciones, canales de entrada/salida, canales de comunicaciones, ficheros, etc.

Los procesos son entidades que son relativamente costosas de crear y manipular por parte del sistema operativo. En cambio, las hebras comparten los recursos de los procesos y su manipulación es mucho menos costosa que la de éstos. La principal ventaja de tener varias hebras de ejecución es permitir la ejecución concurrente de actividades relacionadas dentro de un proceso. De este modo, la organización de la Figura 5 está indicada cuando los tres procesos no presentan ningún tipo de relación. Por el contrario, el esquema de la Figura 6 es el idóneo cuando las tres hebras realizan actividades que forman

Como las hebras de un proceso comparten el mismo espacio de direcciones, no existe ningún tipo de protección por parte del sistema operativo ante el hecho de que una hebra pueda leer o escribir en la pila de otra. Sin embargo, esta protección no debería de ser necesaria, ya que mientras que los procesos pueden pertenecer a varios usuarios, las hebras de un proceso pertenecen a un mismo usuario, por lo que se asume que estará diseñadas para colaborar, no para competir. Esto no significa que exista la necesidad de proporcionar mecanismos para que las hebras de un proceso se puedan sincronizar. La programación de procesos multihebra se ajusta a la programación concurrente sobre sistemas que comparten memoria, por lo que suelen usar semáforos o monitores para sincronización, aunque también existen sistemas que incorporan primitivas de paso de mensajes.

1.6. Caso de Estudio: El Micronúcleo del Sistema Operativo Mach

Mach es un sistema operativo que presenta una arquitectura basada en el concepto de micronúcleo. Los principales objetivos de diseño de este sistema son los siguientes:

- Proporcionar una plataforma sobre la que construir otros sistemas operativos.
- Administrar un espacio de direcciones de gran tamaño.
- Permitir el acceso transparente a los recursos de red.
- Explotar el paralelismo, tanto a nivel de sistema como de aplicaciones.
- Portabilidad.

La idea básica consiste en poder explotar tanto los sistemas multiprocesadores como los distribuidos, a la vez que puedan emular sistemas operativos existentes, como es el caso de UNIX. Esta emulación se lleva a cabo mediante una capa software que se ejecuta fuera del núcleo, en modo usuario.

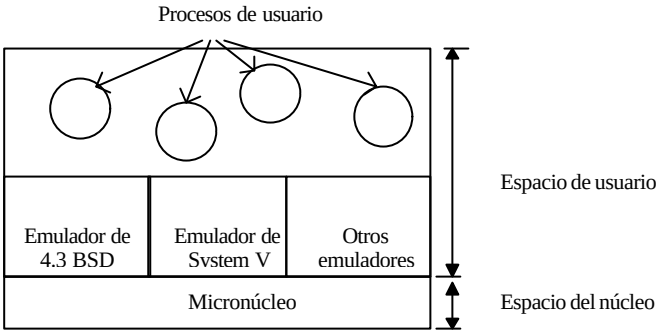

Figura 7.- *Esquema de la emulación de sistemas operativos en Mach.*

El núcleo de Mach proporciona los siguientes servicios:

- gestión de procesos,
- gestión de memoria,
- comunicaciones entre procesos, y
- servicios básicos de entrada/salida.

Los servicios relacionados con la gestión de ficheros, directorios y otras funcionalidades tradicionales de los sistemas operativos se realizan en el espacio de usuario. La idea es que el núcleo suministre los elementos básicos para que el sistema funcione, dejando el resto a procesos de usuario.

El núcleo de Mach maneja cinco abstracciones principales:

1. procesos (*tasks*),
2. hebras (*threads*),
3. objetos de memoria,
4. puertos, y
5. mensajes.

Un proceso es un entorno en el que se puede ejecutar una o más hebras. Los procesos son entidades pasivas y constan básicamente de un espacio de direcciones protegido. Son las unidades básicas para la asignación de recursos. Por ejemplo, un canal de comunicaciones es poseído por un único proceso.

Una hebra es una entidad ejecutable. Tiene un contador de programa y un conjunto de registros asociados. Cada hebra es parte de un único proceso. Un proceso con una hebra es similar a un proceso tradicional.

Un objeto de memoria es una estructura de datos que se puede asociar con el espacio de direcciones de un proceso. Los objetos de memoria ocupan una o más páginas de memoria y constituyen la base del sistema de memoria virtual de Mach. Cuando un proceso hace referencia a un objeto de memoria que no está presente en la memoria principal, se produce un fallo de página. Como en todos los sistemas operativos, el núcleo detecta el fallo de página, pero, a diferencia de otros sistemas, en Mach el núcleo puede enviar un mensaje a un servidor de memoria que se ejecuta en modo usuario para buscar la página.

La comunicación de procesos en Mach se basa en paso de mensajes. Para poder recibir mensajes, los procesos solicitan al núcleo un buzón, denominado puerto. El puerto se almacena dentro del núcleo y tendrá la estructura de una cola constituida por mensajes. Cuando un proceso quiere comunicarse con otro, deposita un mensaje en un puerto de éste. Cada puerto está protegido para garantizar que únicamente los procesos autorizados pueden escribir y leer de él.

Existe un puerto especial en cada proceso, conocido como puerto del proceso, que el proceso usa para la comunicación con el núcleo. La mayoría de las llamadas al sistema se realizan en realidad mediante la escritura de mensajes en este puerto. De forma análoga, cada hebra tiene su propio puerto para llevar a cabo las “llamadas al sistema” relacionadas con ella.

1.7. Referencias

- “Modern Operating Systems”. Tanenbaum, A.S. Prentice-Hall. 1992.
- "Fundamentos de Sistemas Operativos". Lister, A. M., Editorial Gustavo Gili, 1986.
- "Distributed Systems. Concepts and Design". Coulouris, G., Dollimore, J., Kindberg, T. Addison-Wesley, 1994.
- "Operating Systems Concepts". Silberschatz, A., Galvin, P. B. Addison-Wesley. 1994.

1.8. Apéndice: Comunicación y Sincronización entre Procesos

Los sistemas operativos se componen de un conjunto de procesos concurrentes, algunos de los cuales necesitan comunicarse y sincronizarse entre sí. El estudio de la comunicación y sincronización entre procesos se engloba en una disciplina conocida como **programación concurrente**, que tiene su origen en los sistemas operativos.

Los sistemas operativos primitivos usaban una técnica llamada **procesamiento por lotes** (*batch*) para el procesamiento de tareas. Esta técnica consiste en generar una cola de trabajos, los cuales serán ejecutados uno tras otro, según en el orden en el que aparezcan en la cola. Cuando un proceso necesita realizar una operación de entrada/salida, la CPU permanece inactiva hasta que dicha operación finaliza.

El problema del procesamiento por lotes radica en las diferencias de velocidad entre los dispositivos de entrada/salida y los componentes electrónicos de los computadores, que hacen que la ejecución en serie de procesos desaproveche la mayor parte del tiempo de CPU disponible. Por este motivo, aparece la **multiprogramación**, que consiste en la posibilidad de ejecutar un proceso de forma concurrente con las operaciones de entrada/salida de otro proceso. Con multiprogramación, en la memoria principal se cargan varios procesos, uno de los cuales se elige para su ejecución. Cuando este proceso realiza una operación de entrada/salida, se bloquea y se elige otro proceso para su ejecución. Si el número de procesos en memoria es lo suficientemente elevado, la CPU del sistema estará activa durante el 100% del tiempo.

Posteriormente, se generaliza el concepto de multiprogramación dando lugar al ***time-slicing*** o reparto del tiempo, que consiste en compartir el tiempo de procesador entre varios procesos, independientemente de que el proceso que está en ejecución esté realizando una operación de entrada/salida. La forma de realizar este reparto la determina la estrategia de planificación del sistema operativo. De esta forma, las instrucciones de varios procesos se ejecutan de forma entremezclada (***interleaving***). El *time-slicing* da lugar a sistemas interactivos de **tiempo compartido** (*time-sharing*), mediante el que cada usuario puede utilizar un terminal para utilizar un computador de forma interactiva.

Por otro lado, se dice que un sistema operativo es **multitarea** cuando es capaz de ejecutar más de un proceso de forma concurrente. Por consiguiente, todo sistema operativo que incorpore multiprogramación o reparto de tiempo es multitarea. Se distinguen dos tipos de multitarea:

- **Multitarea no apropiativa** (*non-preemptive multitasking*): los procesos se apropian de la CPU, de forma que el sistema operativo únicamente puede ejecutar otro proceso cuando el que tiene la CPU realiza una operación de entrada/salida o la libera de forma voluntaria. Un ejemplo de sistema con este tipo de multitarea es Windows 3.1.
- **Multitarea apropiativa** (*preemptive multitasking*): el sistema le puede quitar la CPU al proceso en ejecución para asignársela a otro. Ejemplos de sistemas operativos con multitarea apropiativa son UNIX y Windows NT.

En la actualidad, aparecen nuevos conceptos que se aplican a los sistemas operativos y que tienen relación con la concurrencia o programación concurrente, como son el multiprocesamiento y el procesamiento distribuido. El **multiprocesamiento** que permite la ejecución de varios procesos sobre diferentes procesadores. Aunque los computadores con varios procesadores se restringían hasta hace poco tiempo a supercomputadores y *mainframes*, en la actualidad es factible la construcción de computadores personales y estaciones de trabajo con varios procesadores y con un coste reducido. Windows NT es un sistema operativo que se ejecuta sobre estos computadores y tiene capacidad de multiprocesamiento. Por último, mediante **procesamiento distribuido** se pueden ejecutar varios procesos sobre un sistema distribuido, que es una colección de computadores independientes interconectados entre sí. Sobre sistemas distribuidos aparecen los sistemas operativos de red y los sistemas operativos distribuidos. La principal diferencia entre ambos es que en los primeros los usuarios son conscientes de la existencia de varios computadores, mientras que en los segundos todo el sistema distribuido aparece a los usuarios como un sistema monoprocesador tradicional.

1.8.1. Principios de Concurrencia

En los sistemas monoprocesador con multitarea los procesos se entremezclan o intercalan en el tiempo para dar la sensación de ejecución simultánea. En los sistemas basados en multiprocesamiento y procesamiento distribuido las instrucciones de los procesos no sólo se entremezclan, sino que también se

race conditions) debido a la intercalación y al solapamiento. Para evitar estos comportamientos no deseados se distinguen dentro de cada proceso aquellas zonas de código que usan recursos compartidos de las que no lo hacen. Estos trozos de código se denominan . Si se consigue que únicamente un proceso pueda estar a la vez en su sección crítica se evita el problema. Sin embargo, esto no es suficiente. Una solución satisfactoria debe de cumplir, al menos, las siguientes condiciones (**problema de la exclusión mutua**):

- Dos procesos no pueden estar a la vez dentro de sus secciones críticas (condición de exclusión mutua).
- Un proceso que está fuera de su sección crítica no debe interferir con otros procesos.
- Un proceso no puede permanecer siempre a la espera de entrar en su sección crítica (**indefinida** o **interbloqueo**).
- En ausencia de competición, un proceso que desee entrar en su sección crítica lo hará a un coste

- No se establecen condiciones acerca de las velocidades de los procesos o número de procesadores del sistema.
- Se asume que un proceso permanece en su sección crítica durante un tiempo finito.

El control de la competición entre procesos involucra al sistema operativo, ya que éste es la entidad que asigna recursos a los procesos. Además, los procesos deben de expresar sus requerimientos de exclusión mutua de alguna forma. Algunos de estos mecanismos son: semáforos, monitores, contadores de sucesos y paso de mensajes.

1.8.2. Sincronización entre Procesos: El Problema de la Exclusión Mutua

La sincronización entre procesos la podemos definir como la constatación por parte de los procesos implicados que un determinado suceso ha ocurrido. La sincronización se suele estudiar mediante el problema de la exclusión mutua. Se han propuesto muchas soluciones a este problema, pudiéndose clasificar entre las que no requieren ningún soporte por parte del lenguaje de programación, sistema operativo o hardware y las que si lo requieren. Entre las primeras están el algoritmo de Dekker, el algoritmo de la panadería y el algoritmo de Peterson. El problema de estas soluciones es que son poco adecuadas para ser llevadas a la práctica. Uno de los problemas que presentan es que requieren realizar esperas activas (*busy waiting*), lo que consume tiempo de CPU de forma innecesaria. A continuación se comentan algunas soluciones que pertenecen al segundo grupo.

1.8.2.1. Inhabilitación de Interrupciones

La inhabilitación de interrupciones (*disabling interrupts*) consiste en que un proceso inhabilite todas las interrupciones justo antes de entrar en su sección crítica y las rehabilite cuando salga de ella. De esta forma se impide que a un proceso se le pueda quitar la CPU (*preemption*). El inconveniente de esta técnica

es que deja en manos del los procesos de usuario el control de las interrupciones. Sin embargo, es útil *kernel*) del sistema operativo.

1.8.2.2. Soluciones con apoyo del Hardware

La solución al problema de la exclusión mutua se simplifica si se dispone de instrucciones hardware adecuadas. Estas instrucciones permiten hacer lecturas y escrituras de forma atómica. Ejemplos típicos son las instrucciones *TEST&SET* y *SWAP*. Sin embargo, las soluciones usando estas instrucciones no evita el problema de la espera activa.

1.8.2.3. Semáforos

Los semáforos son primitivas de más alto nivel que las instrucciones máquina y se suelen implementar como llamadas al sistema. Un semáforo `S` es una variable entera que no puede tomar valores negativos sobre la que hay definidas dos operaciones: `wait(S)` y `signal(S)`. La operación `wait(S)` decrementa en una unidad el valor de `S` si éste era mayor que cero. En caso contrario, el proceso que ejecuta la operación se suspende en el semáforo. La operación `signal(S)` aumenta el valor de `S` en una unidad, si no existe ningún proceso encolado. En caso contrario, despierta uno cualquiera de ellos. La característica fundamental de los semáforos es que las operaciones `wait()` y `signal()` son atómicas. Si un semáforo únicamente puede tomar los valores 0 y 1 se dice que es un semáforo binario.

Ejemplo: Solución al problema de la exclusión mutua usando semáforos

```
/* Problema de la exclusión mutua usando semáforos            */
typedef int semaforo ;
semaforo exmut = 1 ;     /* Semaforo para exclusion mutua     */

void proceso()
{
  while (1) {
     sección_no_crítica(); /* Sección no crítica del proceso */
     wait(&exmut) ;        /* Entrada en la sección crítica  */
     sección_crítica() ;
     signal(&exmut) ;      /* Salida de la sección crítica   */
  } /* while */
} /* proceso */
```

1.8.2.4. Monitores

El problema de los semáforos radica en que son primitivas de bajo nivel. Cualquier error en el uso de las operaciones `wait()` o `signal()` puede originar errores difíciles de descubrir (violación de la exclusión mutua, interbloqueos, etc.) y que presentan un comportamiento impredecible y no reproducible.

Un monitor es una primitiva estructurada para programación concurrente que encapsula dentro de un módulo un conjunto de datos y un conjunto de procedimientos. Estos procedimientos son invocados por procesos y operan sobre los datos, que permanecen ocultos dentro del módulo. La propiedad fundamental de los monitores es que los procedimientos se ejecutan en exclusión mutua. No es posible que existan dentro del monitor dos procesos activos al mismo tiempo.

La sincronización dentro de un monitor se realiza mediante unas estructuras de datos denominadas . Cada variable de condición tiene definida tres operaciones:

- `Wait(C)`: El proceso que invoca al procedimiento del monitor que contiene esta operación se suspende en un cola asociada a la variable de condición `C`. A partir de este momento, el monitor está disponible para otros procesos.
- `Signal(C)`: Si la cola de `C` no está vacía despierta un proceso de la cola. Si la cola está vacía, no ocurre nada.
- `Non_Empty(C)`: Es una función lógica que informa sobre si la cola asociada a `C` está vacía.

Los monitores son una construcción lingüística, por lo que será misión de un compilador la implementación de la exclusión mutua. Al realizarse la exclusión mutua de forma automática, la programación concurrente es menos propensa a errores que si se usan semáforos.

Los monitores tienen el inconveniente de que no son implementados por los lenguajes usados para la programación de sistemas (como es el caso del lenguaje C). Otro inconveniente, que también afecta a los semáforos, es que no fueron diseñados para sistemas distribuidos, que se componen de varias CPUs interconectadas por una red de área local, cada una de las cuales tiene su propia memoria local.

Ejemplo: Solución al problema del productor-consumidor usando monitores

```
(* Ejemplo del uso de monitores para el problema del productor-
consumidor *)

monitor ProductorConsumidor ;
  condition lleno, vacio ;
  integer contador ;

  procedure introducir ;
  begin
    if contador = N then wait(lleno) ;
    introducir_elemento ;
    contador := contador + 1 ;
    if contador = 1 then signal(vacio) ;
  end ;

  procedure extraer ;
  begin
    if contador = 0 then wait(vacio) ;
    extraer_elemento ;
    contador := contador - 1 ;
    if contador = N - 1 then signal(lleno) ;
  end ;
begin
  contador := 0 ;
end ProductorConsumidor ;

procedure productor;
begin
  while true do
  begin
    producir_elemento ;
    ProductorConsumidor.introducir ;
  end ;
end ;

procedure consumidor;
begin
  while true do
  begin
    ProductorConsumidor.extraer ;
    consumir_elemento ;
  end ;
end ;
```

1.8.3. Comunicación entre Procesos: Paso de Mensajes

Cuando dos o más procesos interactúan entre sí necesitan mecanismos que les permitan comunicarse y sincronizarse. En el caso del problema de la exclusión mutua los procesos se comunican mediante variables compartidas y se sincronizan mediante semáforos, monitores, etc. El principal inconveniente de estas soluciones es que son adecuadas para sistemas que comparten memoria (con uno o varios procesadores), pero no son válidas para sistemas distribuidos, en los que los distintos computadores que lo componen no comparten memoria.

Existen dos tipos de tuberías: las anónimas y las que tienen nombre. Las primeras las pueden utilizar únicamente procesos relacionados, mientras que las segundas puede ser usadas por procesos cualesquiera.

1.8.4.2. Mensajes

Un mensaje es un bloque de texto que tiene asociado un tipo. Existen dos llamadas al sistema, *msgsnd* y *msgrcv*. Cada proceso tiene asociada una cola de mensajes, que funciona como un buzón. El emisor de

un mensaje especifica el tipo del mensaje que va a enviar, lo que puede ser utilizado por el receptor para seleccionar el mensaje que quiere aceptar. El receptor puede, además, obtener los mensajes en orden *FIFO*. Cuando la cola de mensajes está llena o vacía, el emisor o el receptor, respectivamente, se

1.8.4.3. Memoria Compartida

La forma más rápida de comunicar procesos en UNIX es mediante memoria compartida, que es un bloque de memoria virtual compartido por varios procesos. Estos procesos escriben y leen la memoria compartida usando las mismas instrucciones máquina que usan para acceder a cualquier porción de su espacio de direcciones de memoria virtual. Un proceso puede tener permisos de sólo lectura o de lectura y escritura.

Los mecanismos necesarios para garantizar la exclusión mutua no forman parte del sistema de memoria compartida y deben ser proporcionados por los procesos que acceden a ella.

1.8.4.4. Semáforos

UNIX System V proporciona llamadas al sistema sobre semáforos que son una generalización de las primitivas `wait` y `signal` clásicas. El núcleo del sistema operativo garantiza que estas operaciones se

Un semáforo consta de los siguientes elementos:

- Valor del semáforo.
- Identificador del último proceso que realizó una operación sobre el semáforo.
- Número de procesos que esperan que el valor del semáforo sea mayor que su valor actual.
- Número de procesos que esperan que el valor del semáforo sea cero.

Cada semáforo tiene asociadas varias colas asociadas con los procesos
semáforos se crean en conjuntos, teniendo un conjunto uno o más semáforos. Existe una llamada al sistema, `semctl`, que permite dar un valor a todos los miembros de un conjunto a la vez. Además, otra llamada, `semop`, permite realizar de una sola vez una secuencia de operaciones definida sobre los semáforos de un conjunto.

1.8.4.5. Señales

Una señal es un mecanismo software que informa a un proceso la ocurrencia de un suceso Son similares a las interrupciones hardware, pero no emplean prioridades, todas son tratadas por igual. Las señales pueden ser enviadas por el núcleo o por los procesos.

Una señal se envía mediante la modificación de un bit en la tabla de procesos del proceso al que se le envía la señal. Como se emplea un bit, no se pueden encolar varias señales de un mismo tipo. La señal se procesa cuando el proceso es planificado para ejecución o cuando va a pasar a modo usuario cuando termina de ejecutar una llamada al sistema. Un proceso puede responder de varias formas ante una señal (ignorarla, aceptarla o dejar que el sistema la trate).

1.8.4.6. *Sockets*

Aunque el concepto de *socket* fue introducido por las versiones de UNIX de Berkeley, se incluye en la actualidad en todas las versiones de UNIX. Un *socket* es un punto por el cual se pueden comunicar dos procesos, análogo a un buzón de correos o a un teléfono. A diferencia de los mecanismos mencionados anteriormente, mediante *sockets* es posible comunicar y sincronizar procesos que se ejecutan en sistemas diferentes.

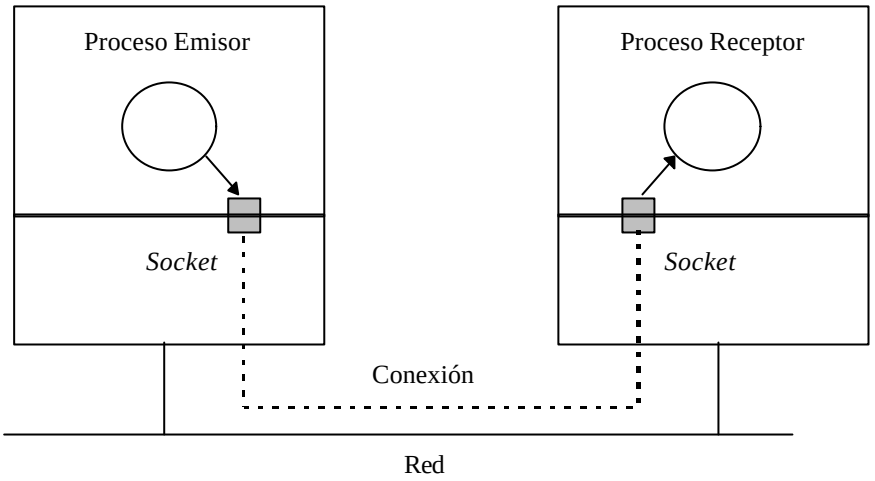

Figura 8.- *Comunicación mediante sockets.*.

Los *sockets* se pueden crear y destruir de forma dinámica. La llamada al sistema que crea un socket devuelve un descriptor de fichero, que puede ser usado dentro de otras llamadas al sistema para leer y escribir a través de él. Antes de usar un *socket*, éste debe tener asignada una dirección de red. Una vez que se han creado los *sockets* en las máquinas origen y destino, se establece una conexión, tras la cual los procesos se comunican de forma similar a las tuberías.

1.8.5. Referencias

- “Modern Operating Systems”. A.S. Tanenbaum. Prentice-Hall. 1992.
- "Operating Systems Concepts". Silberschatz, Galvin. Addison-Wesley, 1994.
- "Principles of Concurrent and Distributed Programming". Ben-Ari, M. Prentice-Hall, 1990.
- "Operating Systems. Second Edition". W. Stallings. Prentice-Hall. 1995.