SISTEMAS OPERATIVOS – PARCIAL 2

CONCURRENCIA

EXCLUSIÓN MUTUA Y SINCRONIZACIÓN

Los temas centrales del diseño de sistemas operativos están todos relacionados con la gestión de procesos e hilos.

- **Multiprogramación**: gestión de múltiples procesos dentro de un sistema monoprocesador.
- **Multiprocesamiento**: gestión de múltiples procesos dentro de un multiprocesador.
- **Procesamiento distribuido**: gestión de múltiples procesos que ejecutan sobre múltiples sistemas de cómputo distribuidos.

La concurrencia es fundamental en todas las áreas mencionadas y en el diseño del sistema operativo.

La concurrencia abarca varios aspectos, entre los cuales están la comunicación entre procesos y la compartición de recursos, la sincronización de actividades de múltiples procesos y la reserva de tiempo de procesador para los procesos. La concurrencia aparece en tres contextos diferentes:

- **Múltiples aplicaciones**: la multiprogramación fue ideada para permitir compartir dinámicamente el tiempo de procesamiento entre varias aplicaciones activas.
- **Aplicaciones estructuradas**: como extensión de los principios del diseño modular y de la programación estructurada, algunas aplicaciones pueden ser programadas eficazmente como un conjunto de procesos concurrentes.
- **Estructura del sistema operativo**: las mismas ventajas constructivas son aplicables a la programación de sistemas y, de hecho, los sistemas operativos son a menudo implementados en sí mismos como un conjunto de procesos o hilos.

El requisito básico para conseguir ofrecer procesos concurrentes es la capacidad de hacer imperar la exclusión mutua, esto es, la capacidad de impedir a cualquier proceso realizar una acción mientras se le haya permitido a otro.

Algunos términos clave relacionados con la concurrencia:

- **Sección crítica**: sección de código dentro de un proceso que requiere acceso a recursos compartidos y que no puede ser ejecutada mientras otro proceso esté en una sección de código correspondiente.
- **Interbloqueo**: situación en la cual dos o más procesos son incapaces de actuar porque cada uno está esperando que alguno de los otros haga algo.
- **Círculo vicioso**: situación en la cual dos procesos o más cambian continuamente su estado en respuesta a cambios en los otros procesos, sin realizar ningún trabajo útil.

- **Exclusión mutua**: requisito por el cual cuando un proceso se encuentra en una sección crítica que accede a recursos compartidos, ningún otro proceso pueda estar en una sección crítica que acceda a ninguno de esos recursos compartidos.
- **Condición de carrera**: situación en la cual múltiples hilos o procesos leen y escriben un dato compartido y el resultado final depende de la coordinación relativa de sus ejecuciones.
- **Inanición**: situación en la cual un proceso preparado para avanzar es soslayado indefinidamente por el planificador, aunque es capaz de avanzar, nunca se le escoge.

PRINCIPIOS DE LA CONCURRENCIA

En un sistema multiprogramado de procesador único, los procesos se entrelazan en el tiempo para ofrecer la apariencia de ejecución simultánea.

Aunque no se consigue procesamiento paralelo real, e ir cambiando de un proceso a otro supone cierta sobrecarga, la ejecución entrelazada proporciona importantes beneficios en la eficiencia del procesamiento y en la estructuración de los programas.

En un sistema de múltiples procesadores no sólo es posible entrelazar la ejecución de múltiples procesos sino también solaparlas.

A primera vista, puede parecer que el **entrelazado** y el **solapamiento** representan modos de ejecución diferentes y que presentan diferentes problemas, pero en realidad, ambas técnicas pueden verse como ejemplos de procesamiento concurrente y ambas presentan los mismos problemas. En ambos casos se plantean los siguientes peligros:

- **La compartición de recursos globales está cargada de peligros**: por ejemplo, si dos procesos utilizan la misma variable global y realizan lecturas y escrituras sobre ellas, el orden en que se ejecutan las lecturas y escrituras es crítico.
- **Para el sistema operativo es complicado gestionar la asignación de recursos de manera óptima**: por ejemplo, un determinado proceso solicita que se le conceda un canal de E/S, se le es concedido y justo después, el proceso es suspendido. Puede no ser deseable que el sistema operativo bloquee simplemente el canal e impida su utilización por parte de otros procesos, ya que esto conllevaría a una condición de **interbloqueo**.
- Puede llegar a ser muy complicado localizar errores de programación porque los resultados son típicamente no deterministas y no reproducibles.

Como hemos dicho antes, todas estas dificultades se presentan tanto en sistemas monoprocesador como multiprocesador, aunque hay que añadir que un sistema multiprocesador cuenta también con problemas derivados de la ejecución simultánea de múltiples procesos.

CONDICIÓN DE CARRERA

Una **condición de carrera** sucede cuando múltiples procesos o hilos leen y escriben datos de manera que el resultado final depende del orden de ejecución de las instrucciones en los múltiples procesos.

Un ejemplo de condición de carrera es el siguiente:

Supónganse dos procesos P1 y P2 que comparten la variable global ‘a’. En algún punto de su ejecución, P1 actualiza el valor de ‘a’ a 1 y en el mismo punto de su ejecución, P2 actualiza el valor de ‘a’ a 2. Así, los dos procedimientos compiten en una carrera por

escribir la variable 'a'. En este ejemplo, el "perdedor" de la carrera (el proceso que actualiza el último) determina el valor de la variable 'a'.

PREOCUPACIONES DEL SISTEMA OPERATIVO

A continuación se muestran aspectos de diseño y gestión que surgen de la concurrencia:

1. El sistema operativo debe ser capaz de seguir la pista de varios procesos.
2. El sistema operativo debe ubicar y desubicar varios recursos para cada proceso activo. Estos recursos incluyen:
    - **Tiempo de procesador**: esta es la misión de la planificación.
    - **Memoria**: la mayoría de los sistemas operativos usan un esquema de memoria virtual.
    - **Ficheros**.
    - **Dispositivos de E/S**.
3. El sistema operativo debe proteger los datos y recursos físicos de cada proceso frente a interferencias involuntarias de otros procesos. Esto involucra técnicas que relacionan memoria, ficheros y dispositivos de E/S.
4. El funcionamiento de un proceso y el resultado que produzca, debe ser independiente de la velocidad a la que suceda su ejecución en relación con la velocidad de otros procesos concurrentes.

Para entender cómo puede abordarse la cuestión de la independencia de la velocidad, necesitamos ver las formas en que los procesos pueden interaccionar.

INTERACCIÓN DE PROCESOS

Podemos clasificar las formas en que los procesos interaccionan en base al grado en que perciben la existencia de cada uno de los otros:

- **Procesos que no se perciben entre sí**: son procesos independientes que no se pretende que trabajen juntos. El mejor ejemplo de esta situación es la multiprogramación de múltiples procesos independientes. Aunque los procesos no estén trabajando juntos, el sistema operativo necesita preocuparse de la **competencia** por recursos.
- **Procesos que se perciben indirectamente entre sí**: procesos que no está al tanto de los demás mediante sus ids, pero que comparten accesos a algún objeto, como un *buffer* de E/S. Estos procesos exhiben **cooperación (por compartición)** en la compartición del objeto común.
- **Procesos que se perciben directamente entre sí**: procesos que son capaces de comunicarse entre sí mediante sus ids y son diseñados para trabajar conjuntamente en una actividad. De nuevo, estos procesos exhiben **cooperación (por comunicación)**

Competencia entre procesos por recursos

Los procesos concurrentes entran en conflicto entre ellos cuando compiten por el uso del mismo recurso.

No hay intercambio de información entre procesos en competencia. No obstante, la ejecución de un proceso, puede afectar al comportamiento de los procesos en competencia.

En el caso de procesos que se encuentran en competencia, deben afrontarse tres problemas de control:

- Es necesario que exista una **exclusión mutua**. Dos o más procesos pueden querer acceder a un recurso común, al que llamaremos **recurso crítico**, y a la porción de programa que lo utiliza, **sección crítica**. Es importante que solo un proceso pueda acceder a este recurso, y no debemos delegar la responsabilidad de elegir a que proceso asignar el recurso al sistema operativo ya que los detalles de los requisitos pueden no ser obvios. La aplicación de la exclusión mutua conlleva los siguientes dos problemas de control.
- **Interbloqueo**. Es decir, dos procesos quieren un recurso que está asignado al otro proceso, así, se bloquean mutuamente y ninguno puede seguir con su ejecución.
- Por último, tenemos el problema de la **inanición**. Puede darse el caso que debido a la planificación de asignación de recursos que tiene el sistema operativo, un proceso quede bloqueado indefinidamente a la espera de un recurso.

El control de la competencia involucra inevitablemente al sistema operativo ya que es quien ubica los recursos. Además, los procesos tienen que ser capaces de expresar de alguna manera el requisito de exclusión mutua, como, por ejemplo, bloqueando el recurso antes de usarlo.

Cooperación entre procesos vía compartición

Como vimos antes, el caso de cooperación vía compartición cubre procesos que interaccionan con otros procesos sin tener conocimiento explícito de ellos.

Un ejemplo de esto es que múltiples procesos pueden tener acceso a variables compartidas o ficheros que pueden usar y modificar sin referenciar a otros procesos, pero saben que otros procesos pueden tener acceso a los mismos datos. Es decir, los procesos deben cooperar para asegurar que los datos que comparten son manipulados adecuadamente.

Dado que los datos están contenidos en recursos, nos vuelven a surgir los tres problemas de control que mencionamos para el caso anterior. La única diferencia es que solo se requiere exclusión mutua en la escritura de datos, no en la lectura.

Sin embargo, en este caso, surge un nuevo requisito: **coherencia de datos**. Para garantizar la coherencia, a veces es necesario declarar en cada proceso una secuencia de instrucciones como sección crítica, esto garantiza la consistencia y sincronización entre los procesos.

Cooperación entre procesos vía comunicación

En este caso de cooperación, los procesos involucrados participan en un esfuerzo común que los vincula. La comunicación proporciona una manera de sincronizar o coordinar actividades varias.

Normalmente, la comunicación se fundamenta en mensajes de algún tipo. Las primitivas de envío y recepción de mensajes deben ser proporcionadas como parte del lenguaje de programación o por el núcleo del sistema operativo.

En el caso de la comunicación, no es necesario el requisito de exclusión mutua, ya que, en el acto de pasar mensajes, los procesos no comparten nada. Sin embargo, los procesos de interbloqueo e inanición sí que están presentes.

REQUISITOS PARA LA EXCLUSIÓN MUTUA

Cualquier mecanismo o técnica que vaya a proporcionar exclusión mutua debería cumplimentar los siguientes requisitos:

1. La exclusión mutua debe hacerse cumplir.
2. Un proceso que se pare en su sección no crítica debe hacerlo sin inferir en otros procesos.
3. No debe ser posible que un proceso que solicite acceso a una sección crítica sea postergado indefinidamente: ni interbloqueo ni inanición.
4. Cuando ningún proceso esté en una sección crítica, a cualquier proceso que solicite entrar en su sección crítica, se le debe permitir entrar sin demora.
5. No se hacen suposiciones sobre las velocidades relativas de los procesos ni sobre el número de procesadores.
6. Un proceso permanece dentro de su sección crítica sólo por un tiempo finito.

Hay varias maneras de satisfacer los requisitos de exclusión mutua mencionados. Una manera es delegar la responsabilidad en los procesos que desean ejecutar concurrentemente. Estos procesos estarían obligados a coordinarse entre sí, sin apoyo del lenguaje de programación ni del sistema operativo. Podemos referirnos a esto como **soluciones software**. Otro enfoque es proporcionar un cierto nivel de soporte dentro del sistema operativo o del lenguaje de programación.

EXCLUSIÓN MUTUA: SOPORTE HARDWARE

Se han desarrollado un cierto número de algoritmos software para conseguir exclusión mutua, de los cuales, el más conocido es el **algoritmo de Dekker**.

Las soluciones software es fácil que tengan una alta sobrecarga de procesamiento y es significativo el riesgo de errores lógicos.

A continuación, se muestran varias soluciones hardware para la exclusión mutua.

DESHABILITAR INTERRUPCIONES

En una máquina monoprocesador, los procesos concurrentes no pueden solaparse, solo pueden entrelazarse. Un proceso continuará ejecutándose hasta que invoque un servicio del sistema operativo o hasta que sea interrumpido.

Por lo tanto, una forma de garantizar la exclusión mutua es impidiendo que el proceso sea interrumpido.

Por medio de primitivas del núcleo del sistema operativo, se desactivan las interrupciones antes de entrar en la sección crítica y se activan de nuevo justo después de salir.

El precio de esta solución es alto, ya que limita la capacidad del procesador para entrelazar la ejecución de procesos. Además, esta solución no funciona en sistemas multiprocesador

INSTRUCCIONES MÁQUINA ESPECIALES

En una arquitectura multiprocesador, varios procesadores comparten acceso a una memoria principal común.

Los procesadores se comportan independientemente en una relación de igualdad. No hay mecanismo de interrupción entre procesadores en el que pueda basarse la exclusión mutua.

A nivel hardware, el acceso a una posición de memoria excluye cualquier otro acceso a la misma posición.

Basándose en ese fundamento, los diseñadores de procesadores han desarrollado varias instrucciones máquina que llevan a cabo dos acciones atómicamente, como leer y escribir o leer y comprobar, sobre una única posición de memoria con un único ciclo de búsqueda de instrucción. Durante la ejecución de la instrucción, el acceso a la posición de memoria se le bloquea a toda otra instrucción que referencie esa posición. Típicamente, estas acciones se realizan en un único ciclo de instrucción.

Ejemplos de este tipo de instrucciones son la **instrucción test and set** (comprueba y establece) y la **instrucción exchange** (intercambio entre el contenido de un registro con el de una posición de memoria).

Para entender el funcionamiento de las instrucciones máquina de las que hablamos, debemos conocer lo que es un cerrojo:

Una variable compartida **cerrojo** se inicializa a 0. El único proceso que puede entrar en su sección crítica es aquél que encuentra la variable cerrojo a 0. El resto de los procesos que intentan entrar en su sección crítica caen en un modo de **espera activa**.

La espera activa se refiere a una técnica en la cual un proceso no puede hacer nada hasta obtener permiso para entrar en su sección crítica, pero continúa ejecutando una instrucción o conjunto de instrucciones que comprueban la variable apropiada para conseguir entrar. Cuando un proceso abandona su sección crítica, restablece el cerrojo a 0, para que otro proceso pueda entrar en su sección crítica. La elección del proceso depende de cuál de los procesos es el siguiente que ejecuta la instrucción test and set.

Propiedades de la solución instrucción máquina

El uso de una instrucción máquina especial para conseguir exclusión mutua tiene ciertas ventajas:

- Es aplicable a cualquier número de procesos sobre un procesador único o multiprocesador de memoria principal compartida.
- Es simple y, por tanto, fácil de verificar.
- Puede ser utilizado para dar soporte a múltiples secciones críticas: cada sección crítica puede ser definida por su propia variable cerrojo.

Sin embargo, este tipo de soluciones también conlleva una serie de desventajas:

- **Se emplea espera activa**: así, mientras un proceso está esperando para acceder a una sección crítica, continúa consumiendo tiempo de procesador.
- **Es posible la inanición**: cuando un proceso abandona su sección crítica y hay más de un proceso esperando para acceder a la suya, la selección del proceso en espera es arbitraria. Así, a algún proceso podría denegársele indefinidamente el acceso.
- **Es posible el interbloqueo**: dependiendo de la prioridad de procesos y del orden en el que ejecuten las instrucciones máquina especiales, se pueden dar escenarios de interbloqueo.

Dados los inconvenientes que nos presentan las soluciones software y hardware que hemos estado describiendo, es necesario que encontremos otros mecanismos.

SEMÁFOROS

A continuación, pasamos a describir mecanismos del lenguaje de programación y del sistema operativo que se utilizan para proporcionar concurrencia.

El principio fundamental es este: dos procesos o más pueden cooperar por medio de simples señales, tales que un proceso pueda ser obligado a parar en un lugar específico hasta que haya recibido una señal específica.

Cualquier requisito complejo de coordinación puede ser satisfecho con la estructura de señales apropiada. Para la señalización, se utilizan unas variables especiales llamadas **semáforos**. Para transmitir una señal vía el semáforo 's', el proceso ejecutará la primitiva ***semSignal(s)***. Para recibir una una señal vía el semáforo 's', el proceso ejecutará la primitiva ***semWait(s)***. Si la correspondiente señal no se ha transmitido todavía, el proceso se suspenderá hasta que la transmisión tenga lugar.

Para conseguir el efecto deseado, un semáforo puede ser visto como una variable que contiene un valor entero sobre el cual solo están definidas tres operaciones:

- Un semáforo puede ser inicializado a un valor no negativo.
- La operación *semWait* decrementa el valor del semáforo. Si el valor pasa a ser negativo, entonces el proceso que está ejecutando *semWait* se bloquea. En otro caso, el proceso continúa su ejecución.
- La operación *semSignal* incrementa el valor del semáforo. Si el valor es menor o igual a cero, entonces se desbloquea uno de los procesos bloqueados en la operación *semWait*.

SEMÁFORO BINARIO O MUTEX

Un semáforo binario es aquel semáforo que solo puede tomar los valores 0 o 1 y se puede definir con las siguientes tres operaciones:

- Un semáforo binario puede ser inicializado a 0 o a 1.
- La operación *semWaitB* comprueba el valor del semáforo. Si el valor es 0, el proceso se bloquea. Si el valor es 1, entonces se cambia el valor a 0 y el proceso sigue con su ejecución.
- La operación *semSignalB* comprueba si hay algún proceso bloqueado en el semáforo. Si lo hay, entonces se desbloquea uno de los procesos. Si no hay procesos bloqueados, entonces el valor del semáforo se pone a 1.

Además de los semáforos binarios, existe otro tipo de semáforo llamado **semáforo con contador** o **semáforo general**.

En ambos semáforos se utiliza una cola para mantener los procesos esperando en el semáforo. La política más favorable de extracción de procesos de la cola es la política FIFO. Los semáforos que utilizan esta política se llaman **semáforos fuertes**. El semáforo que no especifica el orden en el que los procesos son extraídos se denomina **semáforo débil**.

Los semáforos fuertes garantizan que ningún proceso sufrirá inanición, mientras que los semáforos débiles no lo garantizan.

IMPLEMENTACIÓN DE SEMÁFOROS

Como se mencionó antes, es necesario que las funciones **semWait** y **semSignal** sean implementadas como primitivas atómicas. Una forma obvia de hacer esto es implementarlas en hardware.

A falta de esto, se han propuesto variedad de esquemas. La esencia del problema es la de la exclusión mutua: sólo un proceso al mismo tiempo puede manipular un semáforo. Así, cualquiera de los esquemas software, tales como el **algoritmo de Dekker** o el de **Peterson** pueden usarse, aunque esto supone una substancial sobrecarga de procesamiento. Otra alternativa es utilizar uno de los esquemas soportados por hardware (instrucciones máquina especiales) para la exclusión mutua.

En un sistema de procesador único, es posible inhibir las interrupciones durante las operaciones *semWait* y *semSignal*. La relativa corta duración de estas operaciones significa que esta solución es razonable.

MONITORES

Sabemos que los semáforos proporcionan una herramienta potente y flexible para conseguir la exclusión mutua y para la coordinación de procesos. Sin embargo, puede ser difícil producir un programa correcto utilizando semáforos. La dificultad reside en que las operaciones *semWait* y *semSignal* pueden estar dispersas a través de un programa y no resulta fácil ver el efecto global de esas operaciones sobre los semáforos a los que afectan.

El monitor es una construcción del lenguaje de programación que proporciona una funcionalidad equivalente a la de los semáforos, pero es más fácil de controlar.

MONITOR CON SEÑAL

Un monitor es un módulo software consistente en uno o más procedimientos, una secuencia de inicialización y datos locales. Las principales características de un monitor son las siguientes:

- Las variables locales de datos son sólo accesibles por los procedimientos del monitor y no por ningún procedimiento externo.
- Un proceso entra en el monitor invocando uno de sus procedimientos.
- Sólo un proceso puede estar ejecutando dentro del monitor al tiempo: cualquier otro proceso que haya invocado al monitor se bloquea, en espera de que el monitor quede disponible.

Como cumple la condición de sólo un proceso al mismo tiempo, el monitor es capaz de proporcionar exclusión mutua fácilmente.

Las variables de datos en el monitor sólo pueden ser accedidas por un proceso a la vez. Así, una estructura de datos compartida puede ser protegida colocándola dentro de un monitor.

Para ser útil para la programación concurrente, el monitor debe incluir herramientas de sincronización.

Un monitor soporta la sincronización mediante el uso de **variables condición** que están contenidas dentro del monitor y son accesibles sólo desde el monitor. Las variables condición son un tipo de datos especial que se manipula mediante dos funciones:

cwait(c) (suspende la ejecución del proceso llamante en la condición 'c') y *csignal(c)* (retoma la ejecución de algún proceso bloqueado por un *cwait* en la misma condición).

PASO DE MENSAJES

Cuando los procesos interaccionan entre sí, deben satisfacerse dos requisitos fundamentales: sincronización y comunicación.

Los procesos necesitan ser sincronizados para conseguir exclusión mutua. Los procesos cooperantes pueden necesitar intercambiar información.

Un enfoque que proporciona ambas funciones es el **paso de mensajes**.

El paso de mensajes tiene la ventaja de que se presta a ser implementado tanto en sistemas distribuidos como en multiprocesadores de memoria compartida y sistemas monoprocesador.

La funcionalidad del paso de mensajes se presenta mediante las dos siguientes primitivas:

send (destino, mensaje)

receive (origen, mensaje)

SINCRONIZACIÓN

La comunicación de un mensaje entre dos procesos implica cierto nivel de sincronización entre los dos:

El emisor y el receptor del mensaje pueden ser bloqueantes o no bloqueantes, con esto, tenemos las siguientes combinaciones:

- **Envío bloqueante, recepción bloqueante**: emisor y receptor se bloquean hasta que el mensaje se entrega. A esto se le conoce como *rendezvous*.
- **Envío no bloqueante, recepción bloqueante**: aunque el emisor pueda continuar, el receptor se bloquea hasta que el mensaje solicitado llegue.
- **Envío no bloqueante, recepción no bloqueante**: ni emisor ni receptor tienen que esperar.

Para muchas tareas de programación concurrente, es más natural el *send* no bloqueante. Para la primitiva *receive*, la versión bloqueante es más habitual.

DIRECCIONAMIENTO

Existen dos formas de especificar que procesos son destinatarios u origen de mensajes:

- **Direccionamiento directo**: la primitiva *send* incluye un identificador específico del proceso destinatario. En cuanto a la primitiva *receive* hay dos maneras: se debe especificar el proceso emisor o se usa un direccionamiento implícito.
- **Direccionamiento indirecto**: en este caso, los mensajes no se envían directamente por un emisor a un receptor, sino que son enviados a una estructura de datos compartida que consiste en colas que pueden contener mensajes temporalmente. A estas colas se las conoce como **buzones**.

EXCLUSIÓN MUTUA

Para conseguir exclusión mutua usamos una primitiva *receive* bloqueante y una primitiva *send* no bloqueante. El procedimiento es el siguiente:

Un grupo de procesos comparte un buzón en el que hay un único mensaje. Cuando el proceso quiere entrar en su sección crítica, llama a *receive*, si en el buzón no hay ningún mensaje, el proceso se bloquea. En caso contrario, el proceso entra en su sección crítica y al salir, vuelve a poner enviar el mensaje al buzón.