¿Qué es una interrupción?

Una interrupcion es una señal enviada al procesador por un dispositivo de hardware o software indicando que requiere atencion inmediata. Este mecanismo funciona deteniendo la ejeucion del proceso actual en el procesador, guardando su estado y transfiriendo el control a una rutina de manejo de interrupcion especifica que gestione la causa de la interrupcion y finalmente restaurando el estado guardado para continuar el proceso interrumpido. En general permiten al sistema operativo gestionar eventos asincronos sin necedidad de consultar constantemente el estado de los procesos.

¿El modo usuario y modo kernel, son modos de trabajo del hardware o del sistema? Explique

El **modo usuario** y el **modo kernel** son modos de trabajo del sistema que se utilizan para distinguir qué tareas son ejecutadas por parte del usuario, y cuáles por parte del sistema operativo. Dado que los usuarios en un sistema comparten recursos con el SO, el propósito de estos modos es proteger a otros programas, e incluso al sistema operativo en sí mismo, de la ejecución de código que les pueda causar daño y/o impedir su correcto funcionamiento. Si bien a este tipo de código se lo suele llamar "malicioso", no necesariamente se trata únicamente de programas que intencionalmente intenten impedir el correcto funcionamiento de otros, sino que también puede provenir de programas "incorrectos". Por ejemplo, un programa que acceda fuera de su espacio de memoria asignado por un error en su programación puede provocar daños a otros programas en la memoria.

Para lograr esta protección, los modos restringen qué instrucciones puede ejecutar el procesador dependiendo de aquel en que se encuentra el sistema en un momento dado. A aquellas instrucciones que pueden causar daño a otros programas si se las usa incorrectamente, se las llama **instrucciones privilegiadas**, y sólo se pueden ejecutar cuando el procesador se encuentra en modo kernel.

La implementación más sencilla de los modos es mediante hardware, a partir de establecer un bit en un registro del procesador que indica el modo actual, conocido como el **bit de modo** (*mode bit*, en inglés). Así, si el sistema se encuentra en modo kernel, se lo suele representar con un `0` en el bit de modo, y si se encuentra en modo usuario, se lo representa con el bit `1`.

De esta forma, cuando arranca el sistema, éste se encuentra en modo kernel y se carga el sistema operativo, el cual luego inicia las aplicaciones a nivel usuario en el modo con el mismo nombre. Cuando alguna aplicación requiere de algún recurso de parte del SO, lo hacen a través de una **llamada al sistema** (*syscall*, en inglés) que ejecuta un *trap*[1] para establecer el bit de modo en 0, o sea, en modo kernel. En este modo, el SO determina si la operación solicitada por la aplicación es válida y "legal" (es decir, que se encuentra permitida dentro del contexto del programa que la solicitó), y en ese caso la ejecuta, seguido de devolverle el control, habiendo ya revertido el bit de modo a `1` (modo usuario).

Existen implementaciones más complejas donde el bit de modo puede tomar más de dos valores, conocidos como **anillos de protección** (*protection rings*, en inglés), que permiten distinguir entre modos adicionales, como aquellos destinados a servicios del sistema operativo, o para virtualización. El uso de estos anillos implementados en el procesador depende del sistema operativo, siendo a veces poco común el uso de algunos, como es el caso de los anillos 1 y 2 en procesadores Intel con 4 rings, donde los únicos normalmente utilizados son el anillo 0 (modo kernel) y el 3 (modo usuario).

En el caso en el que algún programa intente ejecutar una instrucción privilegiada en modo usuario, el hardware impide su ejecución y le informa al SO mediante un "hardware trap" (una "excepción" por hardware síncrona) para que que éste lo maneje. Típicamente, el resultado de esto es que el programa sea finalizado.

Dicho todo esto, el modo usuario y el modo kernel son, al fin y al cabo, modos conceptuales de trabajo del sistema, en particular utilizados por el SO, cuyos detalles de implementación varían según el sistema. Casi siempre su funcionalidad se implementa por hardware (en particular, en el procesador), y permiten proteger al sistema operativo y al resto de programas, de otros programas que intenten ejecutar instrucciones que puedan impedir el correcto funcionamiento de los primeros.

¿Qué es el software libre?

El Software Libre es un término que se refiere a un tipo de software el cual le da la libertad a los usuarios de:

- Usarlo
- Estudiarlo y modificarlo (ie. tenemos acceso al codigo fuente)
- Redistribuir copias

Algunos ejemplos de Software Libre populares son: GNU/LINUX, LibreOffice, Firefox, GIMP, etc.

Es importante no confundir Software Libre con Open-Source, si bien ambos garantizan acceso al codigo fuente, solo el Software libre nos garantiza que podamos usarlo sin costo, ser redistribuido y modificado; y Open-Source no necesariamente te permite eso

Podemos pensarlo como que:

- Todo Software Libre es Open Source $(SL \subset OS)$
- Pero no todo Software Open Source es Software Libre $(OS \not\subset SL)$

¿Cúales de las siguientes operaciones deben ser privilegiadas?

```
a) Setear un valor del timer.
b) Leer el clock.
c) Limpiar la memoria.
d) Apagar las interrupciones.
e) Modificar la entrada de la table de estados de los dispositivos.
f) Cambiar de modo usuario a modo kernel.
g) Acceder a un dispositivo de entrada y salida.
```

[....]

Algunas CPUs proveen más de 2 modos de operación. ¿Cúales son los dos usos posibles de esos múltiples modos?

[....]

Dar dos razones de por qué los caches son útiles. ¿Qué problemas resuleven? ¿Qué problemas causan?

La memoria caché consiste en un área de almacenamiento de información de alta velocidad. Dentro de la jerarquía de memoria, es más rápida que el almacenamiento primario, aunque más lenta que los registros, y suele descomponerse en varios niveles sucesivamente más eficientes, pequeños y cercanos al procesador.

Dado que tienen el potencial de optimizar los accesos a memoria, las cachés son útiles por varios motivos. Por un lado, permiten acceder rápidamente a datos actualmente en uso por un programa, aprovechando el principio de localidad. La idea básica es que, al momento de buscar un dato, primero se busca en la caché, y si se encuentra, ocurre un *cache hit* y el dato puede obtenerse directamente. En caso contrario, ocurre un *cache miss* y se busca en el almacenamiento primario un bloque que contenga el dato requerido. Esto permite que futuras referencias a este dato o datos cercanos puedan extraerse eficientemente de la caché. Para la mayor parte de los usos cotidianos, el *hit rate* suele ser alto, por lo que ofrece una gran optimización.

Por otro lado, las cachés también se emplean para almacenar instrucciones. Sin esta caché, cada *fetch* de instrucción requeriría leer de la memoria principal, lo que introduciría una latencia de varios ciclos por cada instrucción. Esto provocaría un cuello de botella debido a la velocidad reducida que ofrece la memoria principal en comparación con la caché.

En resumen, las cachés ofrecen una gran ventaja al ocultar la latencia de la memoria principal, en la medida en que sea posible.

Pese a que resuelven múltiples problemas, las cachés también introducen complejidades que deben ser abordadas apropiadamente. En un entorno multihilo o multiproceso, se debe garantizar que las cachés accesibles por cada hilo o proceso sean consistentes entre sí. Las inconsistencias pueden surgir, por ejemplo, si un hilo o proceso escribe en un área de memoria compartida, y el cambio queda oculto porque el otro hilo o proceso lee una caché desactualizada. Este tipo de cuestiones pueden manifestarse en procesadores multinúcleo, en los cuales cada núcleo tiene uno o más niveles de caché privados. Para evitar este tipo de inconvenientes, es esencial el diseño de un protocolo de *coherencia de caché*, que permita mantener la consistencia entre cada caché privada. Esto por lo general suele resolverse a nivel de hardware, por debajo del sistema operativo. Sin embargo, la coherencia debe ser solicitada explícitamente a través de algún mecanismo de sincronización.

Describir el mecanismo por el cúal se refuerza la protección para prevenir que un programa modifique la memoria asociada a otro programa

[....]

¿Qué es un cambio de contexto?

Cuando ocurre una interrupción, el sistema tiene que guardar el contexto actual del proceso que estaba ejecutando en el núcleo de la CPU, para que pueda continuar desde ese punto cuando termine de atender la interrupción. El contexto se representa en la PCB(Process Status Block/ Bloque de estado del proceso) que es una estructura de datos que el Sistema operativo utiliza para almacenar toda la información sobre un proceso como por ejemplo la información de la gestion de la memoria y el valor de los registros de la CPU. Para cambiar el núcleo de la CPU a otro proceso es necesario guardar el estado del proceso actual y restaurar el estado de un proceso diferente. Esto es conocido como un cambio de contexto, cuando esto ocurre el Kernel guarda el contexto del proceso en su PCB y carga el contexto guardado del nuevo proceso para ejecutarse. La velocidad del cambio de contexto cambia según la máquina, según la velocidad de memoria que esta tenga, la cantidad de registros que deban copiarse y la existencia de instrucciones especiales, una velocidad comun es de algunos microsegundos.

¿Cúal es la diferencia entre threads de nivel de usuario y de nivel kernel? ¿En cúales circuntancias uno es mejor que otro?

Threads de nivel de usuario

- Gestión: Son gestionados por bibliotecas de threading en el espacio de usuario, sin interacción directa con el kernel.
- Cambio de contexto: Los cambios entre threads de nivel de usuario son más rápidos, ya que no requieren intervención del kernel.

Ventajas:

- Más eficientes y veloces en sistemas con una sola CPU.
- Ideales cuando la aplicación no requiere utilizar varias CPUs o cuando la interacción con el kernel es limitada.

Desventajas:

- Si un thread realiza una llamada bloqueante al sistema, todos los threads de ese proceso se quedan bloqueados, ya que el kernel no tiene conocimiento de los threads individuales.
- No aprovechan múltiples núcleos de la CPU, dado que el kernel considera el proceso como un único hilo.

Threads de nivel kernel

- Gestión: Son gestionados directamente por el kernel del sistema operativo.
- Cambio de contexto: Los cambios entre threads de nivel kernel son más costosos, ya que requieren intervención del kernel.

Ventajas:

- Pueden aprovechar múltiples CPUs en sistemas con soporte para multiprocesamiento.
- Si un thread se bloquea, otros threads del mismo proceso pueden seguir ejecutándose.

Desventajas:

- Son menos eficientes en sistemas con una sola CPU debido al mayor costo de manejo por parte del kernel.

Circunstancias ideales para cada uno:

Threads de nivel de usuario:

- Aplicaciones que no requieren aprovechar múltiples núcleos de CPU.
- Sistemas donde el rendimiento es crítico y las llamadas al sistema son mínimas.
- Entornos con recursos limitados, donde reducir la carga del kernel es importante.

Threads de nivel kernel:

- Aplicaciones intensivas que necesitan ejecutar en paralelo en múltiples núcleos.
- Programas que realizan muchas llamadas al sistema o interactúan extensamente con el hardware.
- Escenarios donde la robustez y la capacidad de manejar bloqueos son prioritarios.

Describir las acciones que son tomadas por el kernel para hacer un cambio de contexto.

Al realizar el cambio de contexto, el kernel realiza 3 acciones principales

1) Se almacena lo necesario en una estructura llamada PCB (Process control block)

Esto incluye todo lo necesario para retomar la ejecución más adelante y se guarda protegido en memoria del sistema.
Los campos de la estrucutura PCB son los siguientes:

- Estado del proceso
- Contador de programa
- Registros del CPU
- Información de planificación (scheduling) *(Información que ayuda al SO para planificar los procesos como la prioridad, la cola en que esta agendado etc)*
- Información de administración de memoria *(Paginas o segmentos, incluye la pila)*
- Información de contabilidad *(Informacion de la utilización de recursos que ha tenido este proceso (tiempo total empleado, uso acumulado de memoria y dispositivos, etc))*
- Estado de I/O *(dispositivos y archivos que abrió el proceso)*

2) Se elige el siguiente proceso a ejecutar El planificador analiza los procesos listos y decide cuál ejecutará según la política del sistema (por prioridad, tiempo de espera, etc.).

3) Se restaura el estado del nuevo proceso a ejecutrar El kernel carga en el CPU la información del PCB del proceso elegido, restaurando registros, contador de programa, y en algunos casos la memoria virtual. Luego con esta información puede continuar su ejecución como si nunca hubiera sido interrumpido.

¿Qué recursos son usados cuando se crean threads? ¿Cómo difiere de los recursos usados cuando se crea un proceso?

[....]

¿Qué es una llamada al sistema? ¿Cúales son su propósito?

[....]

¿Para qué sirve un intérprete de comandos? ¿Por qué usualmente están separados del kernel?

El intérprete de comandos es una de las formas en que los usuarios pueden interactuar con el sistema operativo.
Permite a los usuarios entrar comandos directamente que sean ejecutados por el SO.
La función principal de un intérprete de comandos es recibir y ejecutar el nuevo comando especificado por el usuario.
Muchos comandos en este nivel manipulan archivos, por ejemplo para crear, eliminar, enlistar, ejecutar, etc.

Hay dos formas generales para implementar estos comandos:

1. **El intérprete contiene el código del comando**
   Por ejemplo, un comando para eliminar un archivo puede hacer que el intérprete salte a una sección de su código que setea los parámetros y hace la llamada al sistema correspondiente.
   En este caso, el número de comandos determina el tamaño del intérprete, ya que cada comando requiere implementar su propio código.

2. **Los comandos se implementan como programas del sistema**
   Esta forma, usada por UNIX entre otros, implementa la mayoría de comandos mediante programas externos.
   El intérprete no entiende los comandos, simplemente usa el nombre del comando para identificar un archivo ejecutable, lo carga en memoria y lo ejecuta.
   Por ejemplo, el comando `rm file.txt` busca un archivo llamado `rm`, lo carga en memoria y lo ejecuta con el parámetro `file.txt`.
   La lógica asociada al comando está completamente definida por el código del archivo `rm`. Una ventaja de este

enfoque es que los programadores pueden añadir nuevos comandos fácilmente, simplemente creando nuevos archivos con la lógica que se necesite. Así, el intérprete de comandos puede mantenerse pequeño y no necesita modificarse para agregar nuevos comandos.

El intérprete de comandos está separado del kernel porque está pensado para poder cambiarse o actualizarse sin que afecte la estabilidad del sistema operativo.

¿Qué es un módulo del kernel?

[....]

¿Qué es un API?

Un API (Application Programming Interface) es un conjunto de funciones que un programador puede usar para desarrollar aplicaciones. El API especifica qué funciones están disponibles, qué parámetros reciben y qué valores devuelven. En lugar de usar directamente las llamadas al sistema, que suelen ser más complejas, se utiliza el API, que es más sencillo de entender y programar. Además, el uso del API mejora la portabilidad del programa entre distintos sistemas, ya que un programa que utiliza un API puede funcionar en diferentes sistemas que lo soporten, sin necesidad de modificar el código.

Detrás del uso del API, el Runtime Environment (RTE) se encarga de servir como intermediario entre la aplicación y el sistema operativo. El RTE traduce las funciones del API en llamadas al sistema específicas del sistema operativo. Estas llamadas están asociadas a números y se gestionan mediante una tabla interna. El programador no necesita saber cómo se implementan las llamadas al sistema, solo debe seguir las reglas del API y entender lo que va a hacer el sistema operativo en respuesta a las llamadas del sistema. Para pasar parámetros al sistema operativo, se pueden usar registros, bloques en memoria o la pila, dependiendo del sistema y de la cantidad de parámetros involucrados.

Ejemplo: #include <fcntl.h> #include <unistd.h>

int fd = open("archivo.txt", O_RDONLY);

open() es una function API POSIX la cual sirve para abrir archivos. El programador no tiene que preocuparse por cómo el sistema operativo realmente abre el archivo. Solo usa la función open() con los parámetros correctos. Internamente, esa función hace una llamada al sistema que el RTE se encarga de manejar.

¿Qué es el process control block (PCB)?

Grupo 11 - Eliseo Vallejos, Jerónimo Delorenzi

El Process Control Block (PCB), o bloque de control de procesos, es una estructura que representa cada proceso dentro del sistema operativo. Contiene información necesaria para iniciar o reiniciar un proceso, junto con algunos datos de contabilidad.
El PCB contiene:

•Estado del proceso: indica el estado del proceso, el cual puede ser: nuevo, en ejecución, en espera, listo, finalizado o zombie.

•Contador de programa: contiene la dirección de la próxima instrucción que debe ejecutarse para el proceso.

•Registros del procesador: contienen acumuladores, registros de índice, punteros de pila y registros de propósito general, además de cualquier código de condición. Esta información se guarda cuando ocurre una interrupción para permitir que el proceso pueda continuar correctamente cuando se reprograma para ejecutarse.

•Información de programación: incluye la prioridad del proceso, punteros a colas de planificación y cualquier otro parámetro necesario para la planificación por parte del sistema operativo.

•Información de gestión de memoria: incluye los valores de registros base y límite, tablas de páginas o tablas de segmentos, dependiendo del sistema de memoria utilizado por el sistema operativo.

•Información de contabilidad: incluye la cantidad de CPU y tiempo real utilizados, límites de tiempo, números de cuenta y números de trabajo o proceso.

•Información de dispositivos de E/S: incluye la lista de dispositivos de entrada/salida asignados al proceso, lista de archivos abiertos y otros recursos de E/S utilizados por el proceso.

¿Cuáles son los mecanismos de comunicación entre procesos?

Si el proceso es cooperativo, luego este requiere un mecanismo de comunicación entre procesos (IPC), de los cuales hay 2 modelos fundamentales, memoria compartida y envío de mensajes. En el modelo de memoria compartida o shared memory, una región en la memoria es establecida, luego los procesos pueden intercambiar información, escribiendo y leyendo lo que esté en la región compartida.En general el sistema operativo evita que los procesos puedan acceder a la información de otros procesos, pero si 2 o más procesos aceptan remover esta restricción se creará una región compartida, notar que este método no está bajo el control del sistema operativo y hace responsable a los procesos de asegurarse de no estar escribiendo en la misma ubicación simultáneamente. En cambio en el modelo de pasaje de mensajes o message passing la comunicación toma forma de mensajes entre los procesos cooperativos.Este modelo provee un mecanismo que permite a los procesos comunicarse y sincronizar sus acciones sin ningún espacio compartido en memoria, por ejemplo en el uso de comunicación entre diferentes computadoras conectadas a la red. Este tipo de modelo tiene al menos 2 funciones:

```
send(message)

receive(message)
```

Los mensajes enviados entre procesos pueden ser fijos de tamaño o variables, con la complicación que cada uno conlleva. Si un proceso P y Q se quieren comunicar, luego se deberá crear un link de comunicacion (communication link), este link se puede implementar de muchas formas: -Comunicación directa o indirecta -Comunicación sincrónica o asincrónica -Buffer automático o específico En el caso de comunicación directa, ambos procesos se quieren comunicar de forma explícita, es decir saben los nombres de los procesos, esto denota simetría, hay variantes simétricas, donde un proceso manda un mensaje a un proceso en específico y otros procesos escuchan sin especificar a quién están escuchando, la desventaja de ambos esquemas es el hecho de que saber los nombres de los procesos requiere hard-

coding. Cuando se habla de comunicación indirecta, los mensajes se envían y se reciben por los llamados puertos o mailbox, cada uno de estos tiene una forma única de identificarse.Dos procesos sólo se pueden comunicar su ambos tiene un puerto compartido, este esquema trae un problema de concurrencia, donde uno debería revisar qué pasa si más de un proceso quiere recibir información del mismo puerto. Los procesos pueden ser los creadores de los mailbox tanto como el sistema operativo, una vez que el proceso dueño de un puerto termine, su puerto terminará con el, y se deberá notificar a los usuarios de ese puerto que el mismo ya no existe. Sincronización, hay 2 tipos de send() y receive() los bloqueantes y no bloqueantes, también conocidos como sincrónico y asincrónicos.Básicamente si se ejecuta alguna función (send o receive) y esta es bloqueante, el proceso frenada hasta poder recibir u/o escribir donde sea pertinente. El hecho de que sea bloqueante puede ayudar a resolver muchos problemas, por ejemplo el de consumidor y productor, pero a su vez generar otros, ya que bloquear procesos no es algo que siempre sea factible. Por último Buffering.Tanto como directa como indirecta, la comunicación entre procesos se puede definir como una cola, y estas pueden ser implementadas de 3 formas -Capacidad cero:La cola no tiene capacidad, por lo que el link no puede tener mensajes esperando dentro, y el enviador se deberá bloquear a la hora de mandar un mensaje. -Capacidad limitada:Esta cola tiene un tamaño finito n, luego puede contener hasta n mensajes. Si a la hora de enviar un mensaje la cola no está llena, luego este mensaje se une a la misma, y el enviador puede seguir con su proceso, caso contrario donde la cola está llena., el enviador se quedará bloqueado esperando a que un recibidor tomé un mensaje y haga lugar en la cola. -Capacidad ilimitada:La capacidad de la cola es prácticamente infinita luego entran una cantidad infinita de mensajes y el proceso que envía nunca se bloquea.

Ambos modelos mencionados son comunes en los sistemas operativos y generalmente ambos están implementados en los mismo, ya que tiene cualidades y beneficios diferentes.Por ejemplo la comunicación a través de mensajes es útil a la hora de transferir poca información, ya que hay menos factores a tener en cuenta. La memoria compartida por otro lado puede ser mas rapida, ya que los mensajes enviados por los procesos son escritos y leídos vía llamadas a sistema, lo cual hace que sea más costosa, la memoria compartida solo necesita establecer una región compartida, una vez establecida todos los procesos que participen de esta memoria, podrán acceder, sin asistencia del kernel.

¿Cúal es la diferencia entre hilos y procesos en Linux?

Las principales diferencias entre los procesos y los threads son:

- Los Hilos dentro de un mismo proceso comparten su espacio de direcciones, memoria global y otros atributos como el PID, lo cual hace que sea más fácil compartir información entre estos. Basta con que copien los datos a compartir en variables compartidas (en el segmento de heap o el de data). En cambio con los procesos se dificulta, ya que al realizar un fork, el padre y el hijo no comparten memoria. En estos casos se debe recurrir a mecanismos de memoria compartida (shm).
- La inicialización de un nuevo Proceso mediante fork() es mucho más costosa que la creación de un nuevo Hilo, ya que hay que duplicar varios de sus atributos como sus páginas de tablas y file descriptors (aún utilizando la técnica de copy-on-write), mientras que al crear un Hilo estos son simplemente compartidos entre ellos.
- Como los hilos comparten su memoria, un error en alguno de ellos puede propagarse al resto de los hilos sobre el mismo Proceso, en el peor de los casos terminando con el mismo. En procesos, esta situación se evita ya que cada uno tiene su propia memoria.
- Al programar con Hilos hay que asegurarse que las funciones sean `Thread-Safe` , mediante el uso de Mutex y otros métodos para evitar condiciones de carrera, Deadlocks y otros.

- Los Hilos de un mismo proceso compiten por el uso de una única memoria virtual, lo cual puede causar problemas en casos con una gran cantidad de Hilos o cuando estos necesiten abundante memoria. Mientras, en cambio, los Procesos tienen la totalidad de su memoria virtual disponible.
- Todos los Hilos deben correr sobre el mismo programa, mientras que con los procesos estos pueden ser resultado de correr diferentes programas.
- Realizar un cambio de contexto entre hilos suele ser más rápido que un cambio entre procesos, ya que los hilos comparten gran parte de su entorno. En particular, como comparten memoria al realizar un CC no se requiere un intercambio de páginas virtuales, una de las operaciones más costosas.

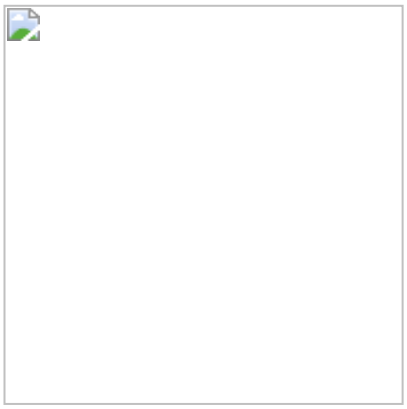
¿Es posible tener concurrencia pero no paralelismo?

Sí, es posible. Los conceptos de concurrencia y paralelismo están relacionados, pero son diferentes. Que un sistema sea concurrente no implica necesariamente que también sea (o no sea) paralelo, y viceversa.

La **concurrencia** es la capacidad de un sistema para manejar múltiples tareas de forma "simultánea". Esto puede lograrse incluso en un sistema con un solo núcleo, intercalando la ejecución de varias tareas en el tiempo, lo que da la *ilusión* de simultaneidad.

El **paralelismo**, por su parte, implica la ejecución *real y simultánea* de múltiples tareas. Para esto se necesitan múltiples núcleos de CPU, de modo que cada uno pueda ejecutar una tarea diferente al mismo tiempo.

Queda claro entonces que puede haber sistemas que sean concurrentes y paralelos a la vez, solo concurrentes, solo paralelos o ninguno de los dos:



¿Qué es en programación concurrente región crítica?

En programación concurrente, una **región crítica** (o **sección crítica**) es un segmento de código dentro de un programa donde se accede o modifica un recurso compartido (como variables, estructuras de datos, dispositivos de E/S, etc.) que debe ser protegido para evitar condiciones de carrera (*race conditions*). La característica principal de una región crítica es que solo puede ser ejecutada por un único hilo o proceso en un momento dado, garantizando así la **exclusión mutua** (*mutual exclusion*). La intercalación de instrucciones en esas secciones críticas provocan condiciones de carrera que pueden generar resultados erróneos dependiendo de la secuencia de ejecución.

¿Qué es race condition?

Race condition (en español condición de carrera) es una categoría de errores de programación en la cual varios procesos acceden a y manipulan los mismos datos de manera concurrente. El error surge de la propia ejecución, ya que como los procesos compiten por acceder y modificar los recursos compartidos, se pueden dar diferentes resultados que dependen del orden de dichos accesos

¿Qué es exclusión mutual?

La Exclusión Mutua, o Mutual Exclusion (Mutex), es un mecanismo dentro de la programación concurrente que busca garantizar que dos o más procesos no puedan acceder dentro de su región crítica al mismo tiempo.
Esto resulta esencial para prevenir condiciones de carrera y asegurar la integridad de los datos compartidos.
Un buen algoritmo de Exclusión Mutua también debe garantizar la ausencia de interbloqueo (deadlock) y inanición (starvation o lockout), lo que significa que ningún proceso debe esperar indefinidamente para entrar o salir de su región crítica.
Ejemplos conocidos de estos algoritmos son el Algoritmo de Peterson o el Algoritmo de la panadería de Lamport.

¿Qué mecanismos para lograr exclusión mutua podemos usar?

Si bien hay muchos métodos que se pueden usar para exclusión mutua vamos a ver los que se cubrieron en la materia:

Deshabilitar interrupciones

Una primera opción basada en hardware, se basa en que cada proceso deshabilite las interrupciones antes de entrar a la zona crítica y las vuelva a habilitar al salir; Eso funciona porque el CPU solo cambia de un proceso a otro cuando ocurre una interrupción. Si bien este método funciona, no es bueno, pues es mala idea darle control a procesos con nivel de usuario la posibilidad de deshabilitar las interrupciones, junto con esto en un sistema con múltiples núcleos hacer esto solo afecta al núcleo que ejecutó la instrucción, los otros continúan funcionando normalmente.

Variables de lock

Como segunda posibilidad podemos pensar en exclusión basada en software, en particular en usar una bandera. Esta se almacena en una variable compartida, accesible a todos los threads, con un valor inicial de 0. Cuando un proceso quiere ingresar a la zona crítica revisa si la bandera es 0; Si lo es le cambia el valor a 1 e ingresa a la zona crítica, de lo contrario espera a que se vuelva 0. El problema que tiene este método es que la comparación y asignación no es una operación atómica. Es decir puede ocurrir que un proceso lea el valor de la bandera y vea que es 0; Justo después ocurre un cambio de contexto y un segundo proceso lee el valor de la bandera, que sigue siendo 0, pues no fue actualizado. Como consecuencia de esto los dos procesos entran a la zona crítica.

C

```C while (bandera); bandera= 1; /* Zona critica */ bandera= 0; ```

x86-assembly

```
LOOP:
    movl    FLAG(%rip), %eax
    testl   %eax, %eax
    jne     LOOP
    movl    $1, FLAG(%rip)
    movl    $0, FLAG(%rip)
```
```

En el ejemplo de arriba vemos cómo revisar el valor de la bandera y poner su valor en 1 se traduce a 4 instrucciones de x86, lo que causa que la operación no sea atómica.

CAS

Una forma de implementar exclusión con una variable de lock, que funcione correctamente, es usando **CAS**(Compare and swap). Esta es una instrucción atómica provista por varias arquitecturas, tiene la forma `CAS(l,a,b)`; Lo que hace es leer el valor en la dirección de memoria `l`, si es igual a `a` se almacena `b` en la dirección a la que apunta `l`, de lo contrario no hace nada. Al mismo tiempo devuelve un booleano indicando si ocurre el cambio o no. En el caso de x86-64 **CAS** se puede implementar usando la instrucción `cmpxchg`. Esto nos permite usar una variable de lock, ya que podemos hacer la comparación y la asignación a la bandera de forma atómica, lo que resuelve el problema del punto anterior. Presentamos una implementación de este metodo de exclusion usando C y `cmpxchg`:

Definición de CAS

Uso de CAS

```C int CAS(volatile int *ptr, int expected, int new_val) { unsigned char success; __asm__ volatile ( "lock cmpxchg %2, %1\n\t" "sete %0" : "=q" (success), "+m" (*ptr), "+r" (new_val) : "a" (expected) : "memory" );

```
return success;
```

}

```
</td>
<td>

```C
//La bandera debe ser una variable global
int flag = 0;

void *funcionThread(void* arg) {
    //Lock de zona crítica. Al ingresar causa flag = 1.
    while(!CAS(&flag, 0, 1));

    /*
        Zona critica
    */

    flag = 0;

    return NULL;
}
```

Alternancia estricta

Otro método posible de exclusión basado en software se basa en usar una variable de turno, esta indica cuál de los procesos puede acceder a la zona crítica. En forma general un proceso entra a la zona crítica cuando el número de la variable de turno es igual a su número de proceso, al salir de la zona crítica este incrementa la variable de turno en 1, permitiendo que otro proceso acceda a la zona crítica. Para mayor claridad vemos un ejemplo con solo dos procesos:

Codigo del proceso 0

```C while (1) { while (turno != 0); /* Region critica */ turno = 1; } ```

Codigo del proceso 1

```
while (1) {
    while (turno != 1);
    /*
        Region critica
    */
    turno = 0;
}
```

Si bien este método evita accesos simultáneos a la zona critica requiere que todos los procesos utilizándolo se alternan de forma estricta, es decir: $P_{1} \implies P_{2} \implies P_{3} \implies ... \implies P_{1}$. De lo contrario un proceso cederá su turno pero no habrá quien se lo dé nuevamente; Usando el código anterior como ejemplo, digamos que el proceso 0 termina, cuando el proceso 1 sale de la región crítica le cede su turno al proceso 0. Sin embargo como el proceso 0 ya no se está ejecutando nunca le devuelve el turno, causando que el proceso 1 se quede esperando, violando la propiedad de *liveness*.

Mutexes

Todos los métodos de exclusión mutua que explicamos hasta ahora comparten una característica importante, que los hace poco eficientes, el uso de *busy waiting*. Así se llama cuando un proceso se queda esperando sin ceder el uso del procesador, es decir el proceso se sigue ejecutando verificando constantemente si se cumple una condición; En los métodos vistos esto se hacía con un loop . Si bien hacer esto funciona es muy ineficiente, pues se desperdicia mucho tiempo de ejecución.

La solución que vimos en clase es utilizar un *mutex*; Un *mutex* es una variable compartida que tiene dos estados, bloqueada o desbloqueada. Cuando un proceso quiere ingresar a la zona crítica esté consulta el estado del *mutex*, si está desbloqueado lo bloquea e ingresa a la zona crítica, de lo contrario el proceso deja de ejecutarse, liberando el CPU. La próxima vez que el proceso se ejecute volverá a revisar el estado del *mutex*.

¿Qué es un deadlock?

Un **DEADLOCK** es una situación que puede ocurrir en entornos de programación múltiple, donde varios hilos *(threads)* compiten por un número limitado de recursos. Si un hilo solicita un recurso que está siendo usado por otro hilo, o que no está disponible en ese momento, el hilo entra en un estado de espera. Este estado se denomina **DEADLOCK**, y ocurre al tener varios hilos esperando indefinidamente ya que éstos no pueden cambiar de estado al tener el recurso pedido, que está tomado por otro hilo en espera. Es decir, ninguno puede continuar su ejecución.

Para que se produzca un deadlock, se deben cumplir simultáneamente las siguientes condiciones **necesarias**:

- ${\color{lightblue}Exclusión \space mutua}$, los recursos solo pueden ser utilizados un hilo a la vez.
- ${\color{lightblue}Hold-and-Wait \space (Retención \space y \space espera)}$, un hilo que retiene recursos puede solicitar otros adicionales.
- ${\color{lightblue}No \space Preemption \space (No \space expropiación)}$, los recursos deben ser liberados voluntariamente.
- ${\color{lightblue}Circular \space wait \space (Espera \space circular)}$, cadena de hilos en donde cada uno espera un recurso que posee el siguiente.
```

\({\color{red} IMPORTANTE: \space Estas \space condiciones \space son \space necesarias, \space NO \space suficiente s\space por \space sí \space solas \space para \space que \space se \space genere \space un \space deadlock.}\)

Para asegurarse que nunca ocurra un deadlock, el sistema puede usar una **prevención contra deadlock** o un **esquema de evasión de deadlock**. El primero provee una serie de métodos que aseguran que al menos una condición necesaria no se cumpla. El segundo necesita que se le dé al sistema operativo información adicional por adelantado acerca de que recursos solicitará y usará el hilo mientras funcione.

De todas las condiciones necesarias, la de *Circular wait* es la más práctica para evitar, se suele imponer un orden fijo para la solicitud de recursos, para romper el ciclo de espera.

¿Qué es un livelock?

Livelock es otra forma de ausencia de liveness. Es similar a deadlock; ambos no permiten que dos o mas hilos procedan, pero los hilos son incapaces de proceder por distintas razones. Mientras deadlock ocurre cuando todos los hilos en un conjunto se bloquean esperando por un evento que pueda ser causado solo por otro hilo en el conjunto, livelock ocurre cuando un hilo continuamente intenta realizar una acción que falla. Esto sucede típicamente cuando los hilos reintentan operaciones fallidas al mismo tiempo. Generalmente se puede evitar haciendo que cada hilo reintente la operación fallida en momentos aleatorios.

Un ejemplo de livelock es cuando dos hilos detectan al mismo tiempo un deadlock entre ellos y liberan el lock para que el otro lo adquiera, sin cuidado terminarán atrapados en un bucle en el que ambos liberan continuamente el lock y nunca logran avanzar.

Livelock es menos cómun que deadlock, aún asi es un problema desafiante en el diseño de aplicaciones concurrentes y, al igual que deadlock, solo puede ocurrir bajo circunstancias específicas de scheduling.

---

1. Cabe aclarar que la definición de *trap* es ambigüa y puede variar según la bibliografía o su definición en una arquitectura, pudiéndose referir de forma general a una interrupción, como a interrupciones por software, o incluso aquellas por software que específicamente son síncronas. En este caso, se lo usa en el contexto de una definición común para la arquitectura x86, donde un *trap* es una "excepción" por software, comúnmente utilizada para implementar llamadas al sistema.↩