Índice

Índice de Preguntas

P-1: OTRO

P0: INTRODUCCIÓN

P1: GENERAL

P2: THREADS - PROCESOS

P3: ERRORES Y +

P4: MÁS INFO

P-1: OTRO

¿Qué es una interrupción? (+)

Una interrupción es una señal generada por un dispositivo de hardware/software que requiere atención inmediata. Cuando sucede esto, el PCB guarda todo lo relacionado con el proceso interrumpido para que si re reprograma su ejecución, pueda continuar con normalidad. Permiten al

sistema operativo gestionar eventos asincronos sin necedidad de consultar constantemente el estado de los procesos.

¿Qué es el software libre?

El software Libre es un tipo de software que permite el uso, modificación y distrubución del mismo. No hay que confundirse con SOftware de código abierto, pues este no permite la libre desitribución y modificación del mismo sin algún coste.

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. Es más rápida que el almacenamiento primario, aunque más lenta que los registros. Optimizan el acceso a memoria.

- Permiten acceder rápidamente a datos 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.
- 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.

Aunque también problemas como la consistencia entre procesos-hilos de un sistema con varios núcleos. 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

¿Qué es un cambio de contexto (CC)? (+)

Un cambio de contexto es el proceso mediante el cual un sistema operativo detiene la ejecución de un proceso o hilo y transfiere el control a otro.

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

Al haber un CC, el sistema realiza 3 acciones:

1. Guarda en el PCB la información relacionada al proceso que se está ejecutando
2. Detiene al proceso ejecutándose y carga el contexto del otro proceso elegido entre los que están en la tabla de procesos - según la política del programa - , para así ejecutarlo
3. Al terminar la ejecución, debido al contexto ya cargado, puede reanudar la ejecución del proceso detenido.

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

Un intérprete de comandos sirve como intermediarios entre el usuario y el kernel (SO).

- Si el comando solicitado por el usuario se encuentra incluido en el intérprete, el kernel lo buscará y ejecutará
- Si el comando está en un programa externo, el kernel llamará a ese programa para ejecutarlo y luevo devolverá su retorno.

Está usualmente separado del kernel para poder modificarlo o actualizarlo sin involucar al SO.

¿Qué es un API? (+/-)

Una API (Application Programming Interface) es una ayuda para los programadores. Permite obtener funcionalidades ya modularizadas sin necesidad de conocer exactamente cómo funcionan internamente, solamente debemos de conocer sus argumentos y el valor de retorno para poder utilizarlas. Un ejemplo, las funciones de threads provienen de la API POSIX.

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.

P0: INTRODUCCIÓN.

¿Qué es un Sistema Operativo (SO)? (+)

Un Sistema Operativo(SO) es un programa que administra el hardware de una pc, además de proporcionar las basea para los programas de aplicación y ser utilizado como un intermediario entre el usuario y el hardware ya que nos permite trabajar con los componentes del mismo.

¿Cuáles son sus 3 funciones principales?

- **ABSTRACCIÓN:** Permite esconder la complejidad del hardware, para ser más accesible a nivel usuario.
- **GESTIÓN DE RECURSOS / SCHEDLING:** Gestiona los recursos para que los procesos no compitan por ellos.
- **AISLAMIENTO:** Al ser multiusuario y multitarea, el SO permite que un usuario utilice el Sistema individualmente sin preococuparse por otros que usen el mismo. (Lo aisla)

¿Qué es la multiprogramación?

La multiprogramación se trata de programación en varios núcleos que permiten ejecutar procesos/hilos en ellos y estas ejecuciones son al mismo tiempo. Por ello, son mayormente más rápidas puesto que maximizan el uso de la PC y la CPU. (Mayor trabajo, en menos tiempo)

¿Qué es programación concurrente? (+)

La programación concurrente son técnicas de programación como mutex, semáforos y sincronización permite a las tareas ejecutarse "simultaneamente", gestionandolas con hilos y procesos. En computadoras de un núcleo esto se simula mediante un orden de ejecución.

¿Es posible tener concurrencia pero no paralelismo?

Sí, es posible. Al igual que al revés o también que pase las 2/ninguna.

P1: GENERAL.

¿Qué es un proceso y cómo se crea?

Un proceso es una entidad dinámica ejecutada por un programa.

1. Fork(): Permite a un proceso, el padre, crear otro llamado hijo. El hijo obtiene las copias del stack, data, heap e instrucciones del padre
2. exec(): el sistema carga un nuevo programa en la memoria de un proceso.
3. wait(NULL): El proceso padre espera a que el hijo termine

Presenta los sig estados:

- INICIO: El SO recibio la solicitud de creación y está creando sus estructuras y recursos
- LISTO: El proceso esta listo para ser ejecutado pero aun no se le asigno ningun procesador
- EJECUCIÓN
- BLOQUEADO: El proceso se encuentra en espera de que suceda algún evento para continuar su ejecusión. No consume procesador
- TERMINADO: El proceso termino su ejecución y se borraran sus estructuras.
- ZOMBIE: El proceso termino su ejecución pero, el SO necesita algunos mecanismos de limpieza para eliminarlo por completo.

¿Qué es el PCB (Process Control Block)? (+)

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:

- 1 **Estado del proceso:** indica el estado del proceso, el cual puede ser: nuevo, en ejecución, en espera, listo, finalizado o zombie.
- 2 **Contador de programa:** contiene la dirección de la próxima instrucción que debe ejecutarse para el proceso.
- 3 **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.
- 4 **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.
- 5 **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.
- 6 **Información de contabilidad:** incluye la cantidad de CPU y tiempo real utilizados, límites de tiempo, números de cuenta y números de trabajo o proceso.
- 7 **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.

¿Qué es un módulo del kernel?

Un módulo del kernel (kernel module) es una parte del código del núcleo que puede ser cargada o descargada dinámicamente sin necesidad de reiniciar el sistema operativo.

Características:

Permite agregar funcionalidades al kernel en tiempo de ejecución (por ejemplo, nuevos drivers de dispositivos).

Se maneja con comandos como insmod, rmmod, lsmod en Linux.

Ejemplos: drivers de red, módulos de sistemas de archivos, extensiones de seguridad.

¿Por qué son útiles?

El kernel puede mantenerse pequeño y modular.

Solo se cargan los módulos necesarios.

¿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.

¿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.
```

Las operaciones que afectan al sistema completo o a otros procesos deben ser privilegiadas (solo ejecutables en modo kernel).

a) Setear un valor del timer → ✅ Sí, privilegiada (controla interrupciones de tiempo)

b) Leer el clock → ❌ No, no es privilegiada (solo lectura de datos)

c) Limpiar la memoria → ✅ Sí, privilegiada (puede afectar otros procesos)

d) Apagar las interrupciones → ✅ Sí, privilegiada (afecta control de CPU)

e) Modificar la entrada de la tabla de estados de los dispositivos → ✅ Sí, privilegiada (cambia dispositivos del sistema)

f) Cambiar de modo usuario a modo kernel → ❌ No directamente. El cambio a kernel se produce solo mediante un trap controlado (no puede forzarlo un programa).

g) Acceder a un dispositivo de entrada y salida → ✅ Sí, privilegiada (los I/O devices se manejan solo desde 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)[1] que ejecuta un *trap*[2] 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).

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

Cuando una CPU ofrece más de 2 modos (más allá de "usuario" y "kernel"), estos se usan para:

Separar niveles de privilegio en servicios del sistema operativo Ejemplo: drivers de hardware pueden correr en un modo intermedio entre kernel y usuario.

Soportar virtualización segura Ejemplo: hipervisores o máquinas virtuales corriendo en un "modo de virtualización" diferente del sistema operativo principal.

(📍 Los famosos protection rings: anillo 0 = kernel, anillo 3 = usuario; anillos 1 y 2 poco usados)

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.

Una llamada al sistema (system call) es el mecanismo controlado por el cual un programa en modo usuario solicita un servicio al sistema operativo (en modo kernel).

Propósito:

Permitir que los programas usen funcionalidades críticas del sistema (leer archivos, usar red, asignar memoria, etc.)

Proteger el sistema: solo a través de system calls controladas un proceso puede acceder a recursos.

Ejemplos de llamadas al sistema:

- read(), write() → acceso a archivos
- fork(), exec() → creación de procesos
- exit() → terminación de proceso
- kill() → enviar señales

P2: THREADS - PROCESOS

¿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.

¿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 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.

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

Los recursos que son usados cuando se crean threads son:

1- Espacio de direcciones : Comparte el mismo espacio de direcciones (memoria) que su proceso padre y los demás threads.

2- Segmento de texto, heap, datos globales y archivos abiertos(files descriptors): Compartido por todos los threads.

3- Stack: Es individual de cada thread.

4- Registros y contador de programa: Individuales de cada thread.

Cuando se crea un proceso esos recursos son usados de la siguiente forma:

1- Espacio de direcciones : Cada proceso tiene su propia memoria.

2- Segmento de texto, heap, datos globales y archivos abiertos(files descriptors): Cada proceso hace una copia.

3- Stack: Es individual de cada proceso.

4- Registros y contador de programa: Individuales de proceso

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

El mecanismo que refuerza la protección para evitar que un programa modifique la memoria de otro programa se basa en la colaboración entre el sistema operativo y el hardware, utilizando los siguientes elementos:

- **Registros base y límite**: Cada proceso cuenta con un registro base que señala el inicio de su área de memoria permitida y un registro límite que define el tamaño autorizado. El hardware verifica automáticamente que todas las direcciones de memoria accedidas se encuentren dentro del rango entre el registro base y el límite.
- **Memoria paginada o segmentada**: La Unidad de Gestión de Memoria (MMU, por sus siglas en inglés) traduce las direcciones virtuales a direcciones físicas, permitiendo que cada proceso perciba su propia memoria virtual, completamente aislada de otros.
- **Permisos de página**: Cada página de memoria tiene asignados permisos específicos (lectura, escritura, ejecución). Si un proceso intenta un acceso indebido, el hardware genera una excepción que permite al sistema operativo finalizar el proceso infractor.

En conjunto, estos mecanismos garantizan que la memoria de cada proceso esté protegida frente a accesos no autorizados, evitando daños o lectura indebida por parte de otros procesos.

¿Qué es exclusión mutua? (+/-)

La exclusión mutua (MUTEX) es un API POSIX que permite manejar los errores como race condition y deadlock a través de asegurar que los hilos sean THREAD-SAFE. Busca garantizar que dos o más procesos no puedan acceder dentro de su región crítica al mismo tiempo.

¿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	x86-assembly

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

`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;`
`}`
	`//La bandera debe ser una var`
	`int flag = 0;`
	
	`void *funcionThread(void* arg`
	`  //Lock de zona crítica. A`
	`  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	Codigo del proceso 1

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

Algoritmo de Peterson

Este algoritmo es una mejora de el propuesto por T. Dekker (Matemático de los Países Bajos), siendo significativamente más simple. Se basa en una variable de turno y una lista que indica la intención de ejecutarse de un proceso. Veremos la versión para dos procesos, sin embargo es posible modificar el algoritmo para trabajar con `n` procesos, introduciendo `n` niveles de espera. En la práctica antes de ingresar a la zona crítica cada proceso indica su intención de ejecutarse, poniendo en `true` su posición en la lista de interés, luego le pasa el turno al otro proceso. Habiendo hecho esto espera a que el proceso al que le pasó el turno salga de la zona crítica, momento en el que le pasa el turno, permitiéndole al proceso entrar en la zona crítica. A continuación se da una implementación de este algoritmo en C:

Codigo del proceso 1	Codigo del proceso 2

`//Lock`
`flag[0] = 1; //El proceso 1 quiere el turno.`
`turn = 2; //Le pasa el turno al proceso 2.`
`while (flag[1] == 1 && turn == 2);`
 	
`/*`
`  Zona critica`
`*/`
 	
`//El proceso 1 ya no quiere el turno.`
`flag[0] = 0;`
	`//Lock`
	`flag[1] = 1; //El proceso 2 quiere el tu`
	`turn = 1; //Le pasa el turno al proceso`
	`while (flag[1] == 1 && turn == 2);`
	
	`/*`
	`  Zona critica`
	`*/`
	
	`//El proceso 2 ya no quiere el turno.`
	`flag[1] = 0;`

Algoritmo de Lamport

Este algoritmo fue inventado por L. Lamport, la idea es simular el sistema de tickets de una panadería, usando una analogía: Un cliente(Proceso) toma un número y se pone en la fila, esperando que sea su turno de que lo atiendan(Ejecutarse), donde el panadero(El algoritmo) va atendiendo y llamando los números.

En la práctica este algoritmo usa dos listas, una que indica el "número de ticket" de cada proceso y otra que los procesos usan para indicar si están eligiendo un “número de ticket”. Cuando un proceso quiere ingresar a la sección crítica, primero se le asigna su número de ticket, el más alto de todos los procesos. Cómo calcular el número de ticket no es una operación atómica, el proceso indica que está eligiendo usando la lista de elección. Esto es para prevenir que otro proceso revise el número de ticket que aún no se terminó de calcular. Finalmente el proceso verifica que su número de ticket sea el menor, si es así entonces "es su turno", puede entrar a la zona crítica, de lo contrario espera hasta

tener el menor número de ticket. Al salir de la zona crítica el número de ticket del proceso se vuelve cero, indicando que no tiene interés en acceder a la misma.

Presentamos una implementación del algoritmo en C, donde `NUM` y `CHOOSING` son variables globales:

```
void lock(const int index) {
    //El proceso indica que está obteniendo un número.
    CHOOSING[index] = true;
    //El proceso obtiene su número.
    int max = NUM[0];
    for (int i = 1; i < N; i++)
        if (NUM[i] > max)
            max = NUM[i];
    NUM[index] = max + 1;
    //El proceso obtuvo un número.
    CHOOSING[index] = false;

    for (int j = 0; j < N; j++)
    {
        //El proceso espera a que nadie esté eligiendo un número.
        while (CHOOSING[j]);

        //El proceso espera a tener el menor número.
        while ((NUM[j] != 0) && ((NUM[j] < NUM[index]) || ((NUM[j] == NUM[index]) && (j < inde
    }
}

void unlock(const int index) {
    NUM[index] = 0;
}
```

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*. En C el uso de mutexes requiere la librería *pthread.h*, además se debe compilar usando las flags `-pthread` y `-lrt` , en ese orden. Esta librería incluye las funciones `pthread_mutex_lock(mutex)` y `pthread_mutex_unlock(mutex)` que bloquean y desbloquean el mutex

dado; Para usarlas se requiere de una variable compartida de tipo `pthread_mutex_t`, que almacena el *mutex*. Se presenta un ejemplo de uso en C:

```
//Variable global con el mutex, hay que inicializarla.
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *funcionThread(void* arg) {
    pthread_mutex_lock(&mutex);
    /*
        Zona critica
    */
    pthread_mutex_unlock(&mutex);

    return NULL;
}
```

Cuando definimos el problema Mutex lo definimos en base a una propiedad que tiene que ser garantizada: la región crítica es accedida por lo sumo un proceso. Las propiedades de 'Safety' son las que garantizan que nada malo puede pasar. Esto lo lograremos insertando mecanismos de sincronización antes y después de la sección crítica.

¿Cuáles son los mecanismos de comunicación entre procesos?

Cuando un proceso es cooperativo, necesita un mecanismo de comunicación entre procesos (IPC). Existen dos modelos fundamentales:

- **Memoria compartida (shared memory):** En este modelo, se establece una región de memoria común. Los procesos pueden intercambiar información escribiendo y leyendo en esta región compartida. Normalmente, el sistema operativo impide que los procesos accedan a la memoria de otros, pero si dos o más procesos acuerdan eliminar esta restricción, se crea una región compartida. Es importante destacar que este método no está supervisado por el sistema operativo, por lo que los procesos deben gestionar la concurrencia y evitar escribir simultáneamente en el mismo lugar.
- **Paso de mensajes (message passing):** Este modelo utiliza el envío y recepción de mensajes entre procesos cooperativos. A diferencia del modelo de memoria compartida, no requiere un espacio de memoria común y permite que los procesos se comuniquen y sincronicen sus acciones, incluso entre computadoras conectadas a una red. Las funciones básicas que proporciona este modelo son:

```
send(message)
receive(message)
```

Los mensajes pueden ser de tamaño fijo o variable, cada uno con sus propias complejidades. Para establecer comunicación entre dos procesos (por ejemplo, P y Q), es necesario crear un enlace de comunicación (**communication link**) que puede implementarse de diversas maneras:

- Comunicación directa o indirecta
- Comunicación síncrona o asíncrona
- Buffer automático o específico

Tipos de comunicación:

1. **Comunicación directa:** Los procesos involucrados conocen explícitamente los nombres del otro, lo que implica simetría. Existe una variante asimétrica donde un proceso envía mensajes a un proceso específico mientras otros escuchan sin especificar a quién. La principal desventaja de este enfoque es la dependencia de nombres codificados (hard-coding).

2. **Comunicación indirecta:** Los mensajes se envían y reciben mediante puertos o "mailbox", identificados de forma única. Dos procesos solo pueden comunicarse si comparten un puerto. Esto puede generar problemas de concurrencia cuando varios procesos intentan acceder al mismo puerto. Los puertos pueden ser creados tanto por los procesos como por el sistema operativo, y cuando un proceso propietario de un puerto finaliza, el puerto también desaparece, notificando a los usuarios del mismo.

Sincronización:

- Las funciones `send()` y `receive()` pueden ser bloqueantes o no bloqueantes (síncronas o asíncronas). Si son bloqueantes, el proceso queda en espera hasta completar la operación, lo cual puede resolver problemas como el del productor-consumidor, pero también presenta inconvenientes en situaciones donde bloquear procesos no es viable.

Buffering:

La comunicación entre procesos, tanto directa como indirecta, puede manejarse mediante colas que pueden implementarse de tres formas:

- **Capacidad cero:** No se permite almacenar mensajes en espera. El emisor queda bloqueado hasta que el receptor tome el mensaje.
- **Capacidad limitada:** La cola tiene un tamaño finito (n). Si no está llena, los mensajes se encolan; en caso contrario, el emisor se bloquea.
- **Capacidad ilimitada:** Permite almacenar una cantidad infinita de mensajes, y el emisor nunca se bloquea.

Ambos modelos son comunes en los sistemas operativos y suelen estar implementados simultáneamente debido a sus diferentes ventajas. Por ejemplo:

- La comunicación a través de mensajes es ideal para transferir pequeñas cantidades de información, ya que implica menos complejidades.

- La memoria compartida suele ser más rápida, ya que los mensajes se envían y leen mediante llamadas al sistema, mientras que en la memoria compartida, una vez establecida la región común, los procesos acceden directamente sin intervención del kernel.

P3: ERRORES Y +

¿Qué es en programación concurrente región crítica?

La región crítica es un área en donde los procesos intentan acceder o modificar al mismo recurso, y según el orden de ejecución esto puede ocasionar distintos resultados y es por ello que resulta importante protegerla (Riesgo de RACE CONDITION). La finalidad es que sólo un proceso pueda ejecutarla al mismo tiempo.

¿Qué es race condition?

Race Condition 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 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**:

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

IMPORTANTE: Estas condiciones son necesarias, **NO** suficientes por sí solas para que se genere un 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? (-)

Propiedad Liveness

- Ausencia de Deadlock
- Ausencia de Inanición: Siempre que un proceso quiera tomar un lock, eventualmente lo hará.

Ausencia de Inanición implica Ausencia de Deadlock, pero no funciona al revés.

Un livelock es otra forma de ausencia de liveness, en donde al igual que el deadlock un proceso queda en espera pero este sigue tratando se ejecutarse fallidamente en bucle. Para prevenir esto, se suele poner un orden para que primero uno intente fallar en la ejecución y luego, el otro.

P4 : MÁS INFO (+)

Sincronización de Procesos

La sincronización se produce cuando uno o más procesos dependen del comportamiento de otro proceso, y puede darse de dos formas:

● Competencia: procesos compiten por un recurso (Ej: Acceso a una Impresora)

● Cooperación : procesos cooperan por un objetivo común. Barreras, Productor/Consumidor, etc.

En general sincronización es el conjunto de reglas y mecanismos que permiten la especificación e implementación de propiedades secuenciales de cada proceso que garantizan la correcta ejecución de un programa concurrente.

Operación Atómica

Manipulación de datos que requiere la garantía de que se ejecute como una sola unidad de ejecución, o fallará completamente, sin resultados o estados parciales observables por otro proceso o en el entorno.

Consistencia Secuencial

Asumimos

1. que las operaciones de cada procesador se realizan en el orden especificado
2. que los stores (escrituras a memoria) son inmediatamente visibles al otro procesador.

Fence (barrera de memoria)

Un Fence (instrucción mfence) causa que la CPU:

1. No reordene instrucciones a través del mismo
2. Garantice que todas las escrituras previas al fence son visibles a todos los procesadores antes de continuar

Spooling

El uso más común del spooling es la impresión: los documentos formateados para impresión se almacenan en una cola a la velocidad de la computadora, y luego se recuperan e imprimen a la velocidad de la impresora. Múltiples procesos pueden escribir documentos en el spool sin espera y realizar otras tareas, mientras el proceso de "spooler" opera la impresora.

Usa shm_open si necesitas crear un segmento de memoria compartida para que varios procesos trabajen con él y mmap si quieres mapear un archivo o descriptor de memoria al espacio de direcciones de un proceso

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

   

2. 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.