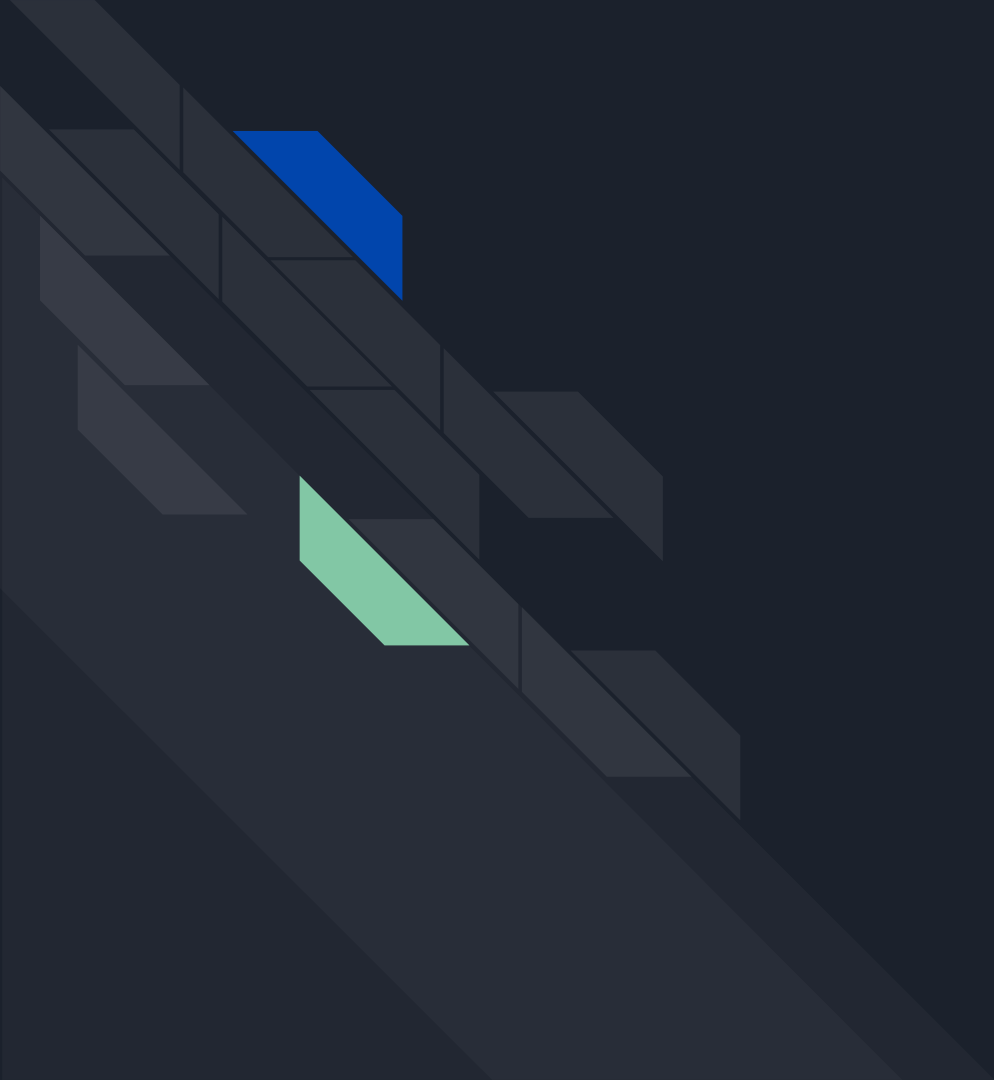




ARQUITECTURA Y SISTEMAS OPERATIVOS

CLASE 6

SINCRONIZACIÓN DE PROCESOS



REPASO



PLANIFICACIÓN APROPIATIVA Y NO APROPIATIVA

Un *kernel* no apropiativo esperará a que se complete una llamada al sistema o a que un proceso se bloquee mientras espera a que se complete la E/S, antes de realizar un cambio de contexto.

Un *kernel* apropiativo requiere mecanismos como bloqueos *mutex* para evitar condiciones de carrera al acceder a las estructuras de datos compartidas del *kernel*.



CRITERIOS DE PLANIFICACIÓN

Se han sugerido muchos criterios para comparar algoritmos de planificación, los cuales incluyen:

- Utilización de la CPU. Mantener la CPU lo más ocupada posible.
- Throughput. Una medida del trabajo realizado es el número de procesos que se completan por unidad de tiempo, se denomina *throughput*.
- Tiempo de retorno. Es el intervalo de tiempo desde el momento en que se despacha un proceso, hasta su finalización. Es la suma de los tiempos de espera en la cola de listos, la ejecución en la CPU y las operaciones de E/S.
- Tiempo de espera. Es importante destacar que el algoritmo de planificación de la CPU solo afecta el tiempo que un proceso pasa esperando en la cola de listos. El tiempo de espera es la **suma de los períodos de espera** en la cola de listos.
- Tiempo de respuesta. Es el tiempo que tarda un proceso en comenzar a responder, no el tiempo que tarda en generar la respuesta.

Lo deseable: maximizar la utilización y el rendimiento de la CPU, y minimizar los tiempos de retorno, de respuesta, y de espera. En la mayoría de los casos, se busca optimizar la medida promedio.



ALGORITMOS DE PLANIFICACIÓN

La planificación se ocupa del problema de decidir a qué proceso de la cola de listos se le asignará un núcleo de la CPU.

- First-Come First-Served (FCFS)
- Shortest-Job-First (SJF)
- Round-Robin (RR)
- Priority
- Priority + Round-Robin
- Multilevel Queues
- Multilevel Queues with feedback

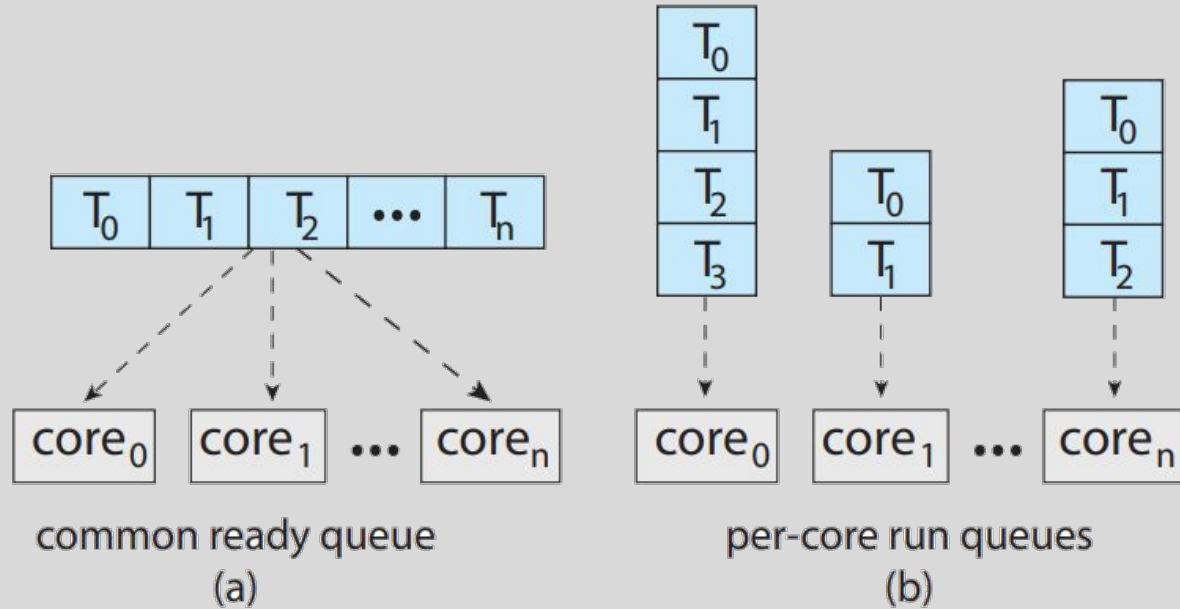
ENFOQUE ESTANDAR

Multiprocesamiento simétrico (SMP), donde cada procesador se auto planifica.

1. Todos los hilos pueden estar en una cola de hilos listos común.
2. Cada procesador puede tener su propia cola privada de hilos.

En la primera opción, existe una posible condición de carrera en la cola de listos compartida.

La segunda opción permite que cada procesador planifique hilos, y así evita los posibles problemas de rendimiento asociados a una cola compartida.

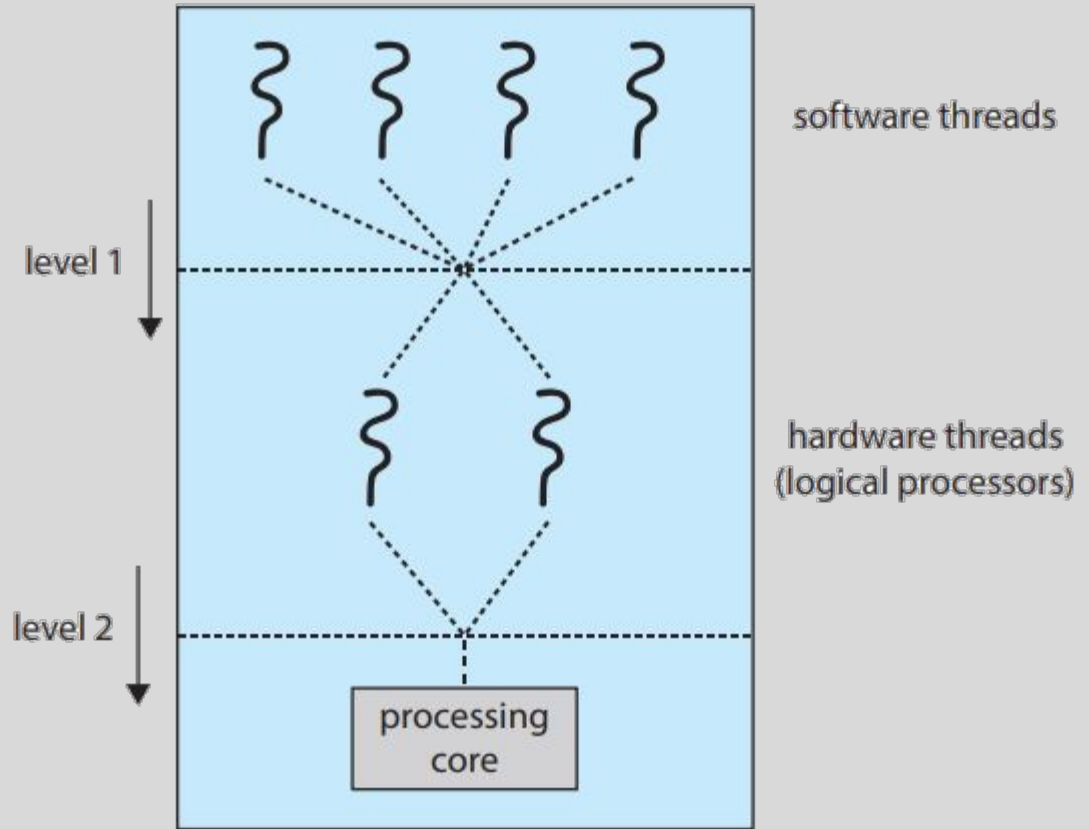


Organization of ready queues.

MULTITHREADED PROCESSING

En un primer nivel se encuentran las decisiones de planificación que debe tomar el sistema operativo al elegir qué hilo de software ejecutar en cada hilo de hardware (CPU lógica). Donde el sistema operativo puede elegir cualquier algoritmo de planificación mencionados.

Un segundo nivel de planificación especifica cómo cada núcleo decide qué hilo de hardware ejecutar. Existen varias estrategias para esta situación. Un enfoque tradicional consiste en utilizar un algoritmo simple de round-robin para planificar un hilo de hardware en el core de procesamiento.



Two levels of scheduling



BIBLIOGRAFIA

- Operating System Concepts. By Abraham, Silberschatz.
 - Capítulo VI
- Modern Operating Systems. By Andrew S. Tanenbaum.
 - Capítulo II

TEMAS DE LA CLASE

- Background
- Race Conditions
- Critical Regions
- Mutual Exclusion with Busy Waiting
 - Disabling Interrupts, Lock Variables and Spin Lock
 - Peterson's Solution
 - The TSL Instruction
- Mutual Exclusion with Sleep and Wake Up
 - The Producer-Consumer Problem
- Semaphores
- Mutex



BACKGROUND





BACKGROUND

Un proceso que puede afectar o verse afectado por otros procesos que se ejecutan en el sistema, se dice que es cooperativo. Estos procesos pueden compartir directamente un espacio de direcciones lógicas (código y datos), o mediante memoria compartida o paso de mensajes, solo compartir datos. Pero viendo cómo es la función del planificador de procesos y de la CPU, donde se alterna rápidamente entre procesos para proporcionar una ejecución concurrente, este acceso concurrente a datos compartidos puede generar inconsistencias.

Esto significa que un proceso puede ejecutarse parcialmente antes de que se programe otro, y puede interrumpirse en cualquier punto de su flujo (*stream*) de instrucciones. Más aún, en un sistema multinúcleo con ejecución paralela, donde existen más de un *stream* de instrucciones que se ejecutan simultáneamente en núcleos de procesamiento separados.



PRODUTOR - CONSUMIDOR EJEMPLO DEL REPOSITORIO

Productor

```
// Initialize shared memory
shared_mem->count = 0;
while (fgets(input, BUFFER_SIZE, stdin) != NULL) {
    // Wait if buffer is full
    while (shared_mem->count > 0) {
        sleep(1);
    }
    // Copy input to shared memory
    strncpy(shared_mem->buffer, input, BUFFER_SIZE);
    shared_mem->count++;
}
```

Consumidor

```
// Wait for producer to write data
while (1) {
    // Check if there's data to consume
    if (shared_mem->count > 0) {
        printf("Consumed: %s", shared_mem->buffer);
        shared_mem->count = 0; // Mark as consumed
    }
    sleep(1); // Wait before checking again
}
```

PRODUTOR - CONSUMIDOR

El ejemplo anterior es solo ilustrativo y puede resultar útil para los tiempos de interacción humana, pero tiene demasiadas deficiencias.

- No se controla el acceso a la memoria compartida.
- Cada proceso duerme al menos 1 segundo siempre.

Si queremos que los procesos se comuniquen entre sí sin la intervención humana, debemos realizar algunas modificaciones.

- `#define BUFFER_SIZE 10;`
- `count = 0;`

```
while (true) {  
    /* produce an item in next_produced */  
  
    while (count == BUFFER_SIZE)  
        ; /* do nothing */  
  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    count++;  
}
```

PRODUTOR

```
while (true) {  
    while (count == 0)  
        ; /* do nothing */  
  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    count--;  
  
    /* consume the item in next_consumed */  
}
```

CONSUMIDOR

PRODUTOR - CONSUMIDOR

Aunque los programas son correctos por separado, podrían no funcionar correctamente al ejecutarse simultáneamente. Por ejemplo, supongamos que el valor de la variable `count` es actualmente 5 y que los procesos ejecutan simultáneamente "`count++`" y "`count--`". Luego de la ejecución de estas dos instrucciones, el valor de la variable `count` puede ser 4, 5 o 6. Claramente, el único resultado correcto es `count == 5`, que se generaría correctamente si el productor y el consumidor se ejecutan por separado.

```
while (true) {  
    /* produce an item in next_produced */  
  
    while (count == BUFFER_SIZE)  
        ; /* do nothing */  
  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    count++;  
}
```

PRODUTOR

```
while (true) {  
    while (count == 0)  
        ; /* do nothing */  
  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    count--;  
  
    /* consume the item in next_consumed */  
}
```

CONSUMIDOR

RACE CONDITION

Las instrucciones “count++;” y “count--;” se podrían implementar en lenguaje máquina de la siguiente manera. Donde los registros 1 y 2 son registros de la CPU.

La ejecución concurrente de estas instrucciones equivale a una ejecución secuencial, donde las sentencias de nivel inferior se intercalan en un orden arbitrario, conservando el orden de nivel superior. Un ejemplo de este intercalado es el siguiente:

$register_1 = count$
 $register_1 = register_1 + 1$
 $count = register_1$

$register_2 = count$
 $register_2 = register_2 - 1$
 $count = register_2$

T_0 :	<i>producer</i>	execute	$register_1 = count$	{ $register_1 = 5$ }
T_1 :	<i>producer</i>	execute	$register_1 = register_1 + 1$	{ $register_1 = 6$ }
T_2 :	<i>consumer</i>	execute	$register_2 = count$	{ $register_2 = 5$ }
T_3 :	<i>consumer</i>	execute	$register_2 = register_2 - 1$	{ $register_2 = 4$ }
T_4 :	<i>producer</i>	execute	$count = register_1$	{ $count = 6$ }
T_5 :	<i>consumer</i>	execute	$count = register_2$	{ $count = 4$ }

RACE CONDITION

Notar que de esta forma, se llega a un estado incorrecto de “`count == 4`”, indicando que cuatro ítems del búffer están llenos, cuando en realidad son cinco. Si se invierte el orden de las instrucciones en T4 y T5, se llegaría al estado incorrecto de “`count == 6`”.

$register_1 = count$
 $register_1 = register_1 + 1$
 $count = register_1$

$register_2 = count$
 $register_2 = register_2 - 1$
 $count = register_2$

T_0 :	<i>producer</i>	execute	$register_1 = count$	{ $register_1 = 5$ }
T_1 :	<i>producer</i>	execute	$register_1 = register_1 + 1$	{ $register_1 = 6$ }
T_2 :	<i>consumer</i>	execute	$register_2 = count$	{ $register_2 = 5$ }
T_3 :	<i>consumer</i>	execute	$register_2 = register_2 - 1$	{ $register_2 = 4$ }
T_4 :	<i>producer</i>	execute	$count = register_1$	{ $count = 6$ }
T_5 :	<i>consumer</i>	execute	$count = register_2$	{ $count = 4$ }

RACE CONDITION

Estos estados incorrectos se generan por permitir que ambos procesos manipulen la variable `count` simultáneamente. Esta situación, donde varios procesos acceden y manipulan los mismos datos simultáneamente y el resultado de la ejecución depende del orden particular en que se realiza el acceso, se denomina condición de carrera (*race condition*). Para evitarlo, es necesario asegurarse de que solo un proceso a la vez pueda manipular la variable `count`.

Para esto, es necesario sincronizar los procesos de alguna manera.

```
register1 = count  
register1 = register1 + 1  
count = register1
```

```
register2 = count  
register2 = register2 - 1  
count = register2
```

T_0 :	<i>producer</i>	execute	<code>register₁ = count</code>	{ <code>register₁ = 5</code> }
T_1 :	<i>producer</i>	execute	<code>register₁ = register₁ + 1</code>	{ <code>register₁ = 6</code> }
T_2 :	<i>consumer</i>	execute	<code>register₂ = count</code>	{ <code>register₂ = 5</code> }
T_3 :	<i>consumer</i>	execute	<code>register₂ = register₂ - 1</code>	{ <code>register₂ = 4</code> }
T_4 :	<i>producer</i>	execute	<code>count = register₁</code>	{ <code>count = 6</code> }
T_5 :	<i>consumer</i>	execute	<code>count = register₂</code>	{ <code>count = 4</code> }



REGIONES CRÍTICAS



REGIONES CRÍTICAS

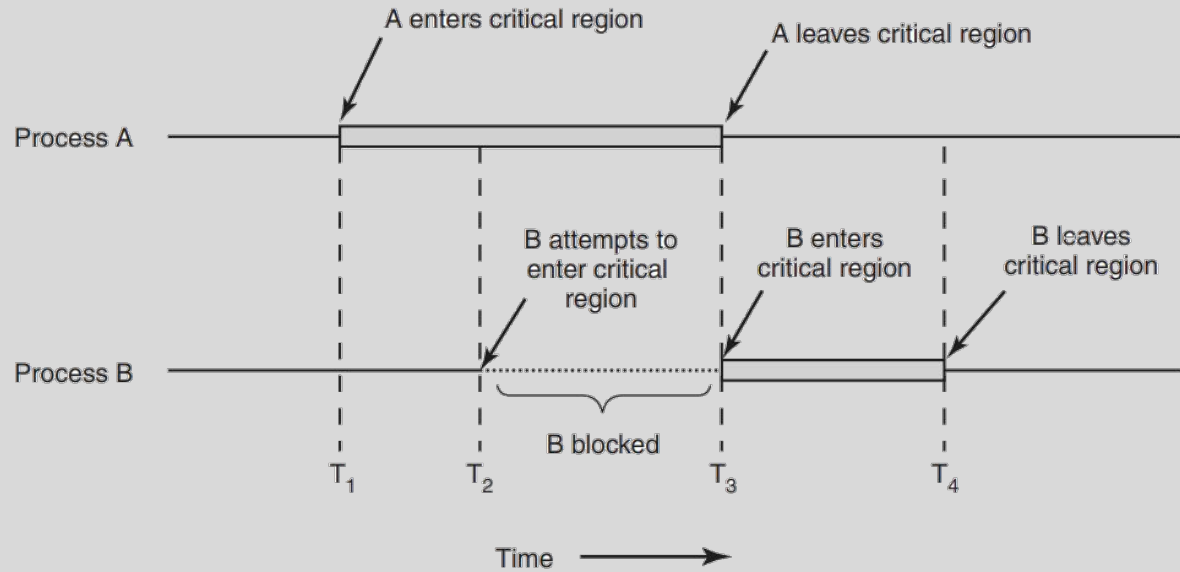
Para evitar problemas en situaciones relacionadas con memoria compartida, archivos compartidos, etc. hay que encontrar la manera de impedir que más de un proceso lea y escriba los datos compartidos simultáneamente. Es decir, se necesita exclusión mutua; esto es, una forma de garantizar que si un proceso usa una variable o archivo compartido, los demás procesos no puedan hacer lo mismo. Como vimos, los problemas mencionados anteriormente se producen porque el proceso 2 comenzó a usar una de las variables compartidas antes de que el proceso 1 terminara de usarla. Esa parte del programa donde se accede a la memoria compartida se denomina región crítica o sección crítica.

Para obtener una solución completa, necesitan cumplirse cuatro condiciones:

1. No se pueden hacer suposiciones sobre la velocidad ni el número de CPU. *
2. No pueden haber dos procesos simultáneamente dentro de sus regiones críticas.
3. Ningún proceso que se ejecute fuera de su región crítica puede bloquear a otro proceso.
4. Ningún proceso debería tener que esperar infinitamente para entrar en su región crítica.

CRITICAL REGION

En abstracto, el comportamiento deseado se muestra en la figura. El proceso A entra en su región crítica en el tiempo T1. En el tiempo T2, el proceso B intenta entrar en su región crítica, pero falla porque otro proceso ya está en su región crítica y solo se permite uno a la vez. En consecuencia, B se suspende temporalmente hasta el tiempo T3, cuando A sale de su región crítica, lo que permite a B entrar inmediatamente. Finalmente, en el tiempo T4, B sale de su región crítica y se presenta a la situación original, sin procesos en sus regiones críticas.

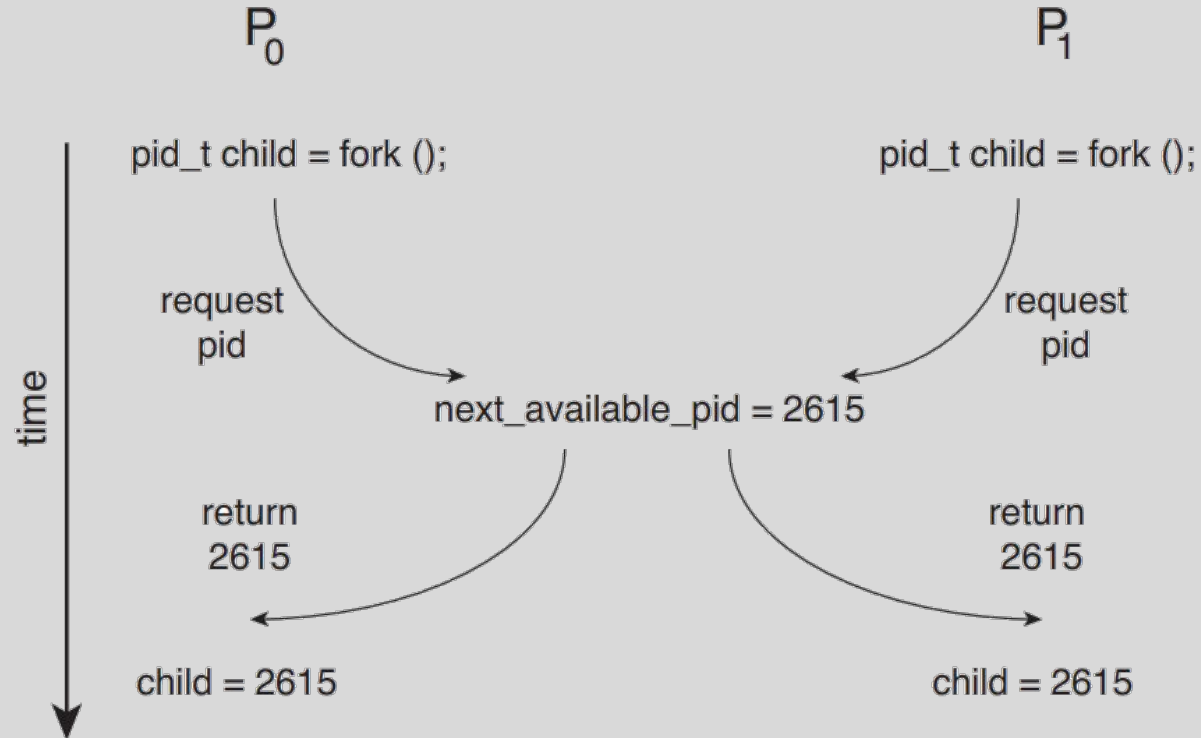


Mutual exclusion using critical regions.

```
while (true) {  
    entry section  
    critical section  
    exit section  
    remainder section  
}
```

EJEMPLO

Dos procesos, P0 y P1, crean procesos hijos mediante la `syscall fork()`, que devuelve el id del proceso recién creado al proceso padre. Aquí, existe una condición de carrera en la variable del `kernel next_available_pid` que representa el valor del siguiente id de proceso disponible. A menos que se proporcione exclusión mutua, es posible asignar el mismo número de id a dos procesos distintos.



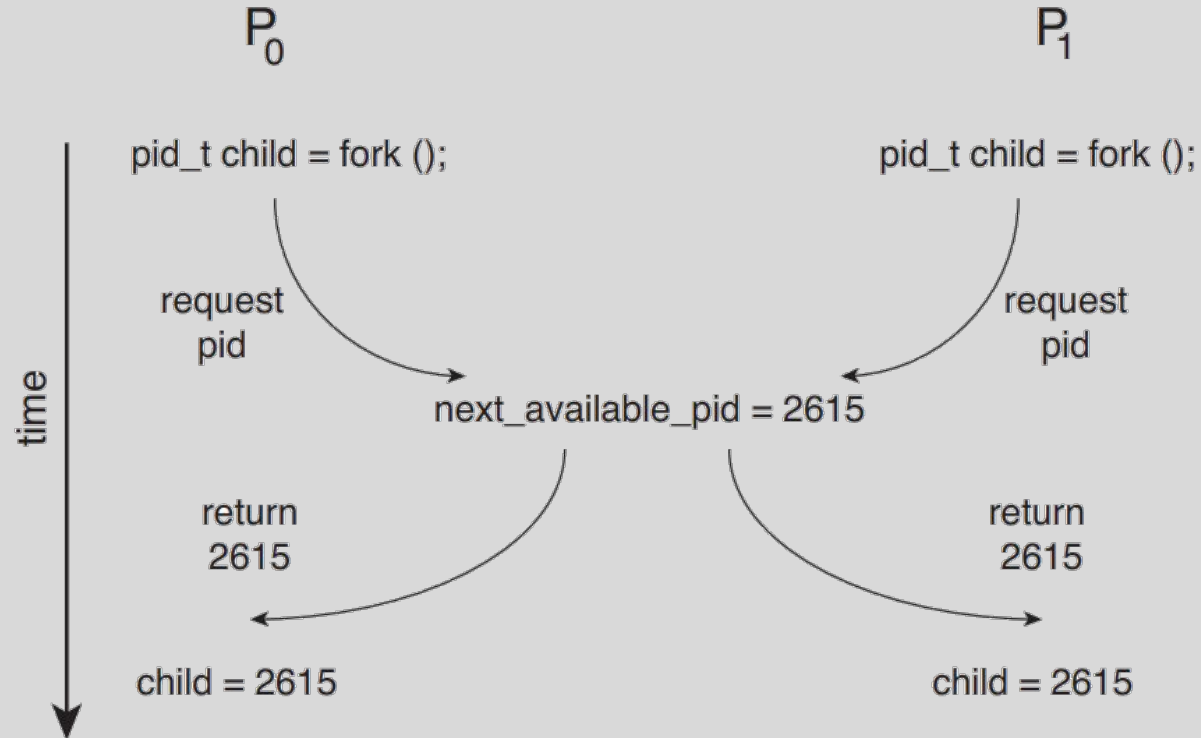
Race condition when assigning a pid.

EJEMPLO

Otras estructuras de datos del *kernel* propensas a posibles condiciones de carrera incluyen:

- las estructuras para mantener la asignación de memoria
- las listas de procesos
- listas de archivos abiertos
- la gestión de interrupciones
- etc

Es responsabilidad de los desarrolladores del *kernel* garantizar que el sistema operativo esté libre de estas condiciones de carrera.



Race condition when assigning a pid.



EXCLUSION MUTUA CON BUSY WAITING



EXCLUSION MUTUA

Diferentes propuestas para lograr la exclusión mutua, de manera tal que mientras un proceso está ocupado actualizando la memoria compartida en su región crítica, ningún otro proceso ingrese a su región crítica y cause problemas.

- **Deshabilitar interrupciones**

En un sistema de un solo procesador, la solución más sencilla es que cada proceso deshabilite las interrupciones justo después de entrar en su región crítica y las vuelva a habilitar justo antes de salir. De esta forma, no pueden ocurrir interrupciones y la CPU no cambiará a otro proceso. Así, el proceso puede examinar y actualizar la memoria compartida sin temor a que intervenga ningún otro proceso. De todas formas, este enfoque no es atractivo porque no es prudente dar a los procesos de usuario la capacidad de desactivar las interrupciones. ¿Qué pasaría si un proceso las desactiva y nunca las volviera a habilitar? Además, en un sistema multinúcleo, desactivar las interrupciones de una CPU no impide que otras interfieran con las operaciones que realiza la primera. En conclusión, desactivar las interrupciones no es apropiado como mecanismo general de exclusión mutua para los procesos de usuario.



EXCLUSION MUTUA

- **Variables de bloqueo**

Una posible solución de software. Una única variable compartida (de bloqueo), inicialmente 0. Cuando un proceso quiere entrar en su región crítica, primero chequea el bloqueo. Si es 0, el proceso lo cambia a 1 y entra en la región crítica. Si el bloqueo ya es 1, el proceso espera hasta que se convierta en 0. Por lo tanto, un 0 significa que ningún proceso se encuentra en su región crítica, y un 1 significa que algún proceso sí lo está.

Desafortunadamente, esta idea contiene exactamente la misma falla que la solución del Productor y Consumidor. Por ejemplo, un proceso lee el bloqueo y ve que es 0. Antes de que pueda cambiar el bloqueo a 1, se planifica otro proceso, se ejecuta y este lo cambia a 1. Cuando el primer proceso se vuelva a ejecutar, también cambiará el bloqueo a 1, y dos procesos estarán en sus regiones críticas al mismo tiempo.

EXCLUSION MUTUA

- **Busy waiting**

Un tercer intento, también de SW, para lograr la exclusión mutua fue la siguiente. La variable "turn", inicialmente en 0, registra a quién le toca entrar en la región crítica. Al inicio, el proceso 0 inspecciona "turn", y cómo es 0 entra en su región crítica. El proceso 1 al inspeccionar, también obtiene 0, por lo tanto permanece en un bucle (*loop*) inspeccionando continuamente "turn" para ver cuándo se convierte en 1. Verificar continuamente una variable hasta que aparezca un valor se denomina "espera ocupada" (*busy waiting*), y en lo posible debe evitarse, ya que desperdicia tiempo de CPU, solo en casos donde exista una expectativa razonable de que la espera será corta, es posible utilizar espera ocupada. Entonces, un bloqueo que utiliza la espera ocupada se denomina "bloqueo de giro" (*spin lock*).

```
int turn = 0;

while (TRUE) {
    while (turn != 0)    /* loop */ ;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(a) Process 0.

```
while (TRUE) {
    while (turn != 1)    /* loop */ ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

(b) Process 1.

EXCLUSION MUTUA

- **Busy waiting**

Cuando el proceso 0 abandona la región crítica, establece el turno en 1 para permitirle al proceso 1 entrar en ella. Supongamos que el proceso 1 finaliza su región crítica rápidamente, y ambos procesos se encuentran en sus regiones no críticas, con el turno en 0. El proceso 0 termina rápido su sección no crítica y vuelve al inicio del bucle, ingresa a su región crítica, cambia el turno a 1 y continúa. En este punto, el turno es 1 y ambos procesos se ejecutan en sus regiones no críticas. El proceso 0 finaliza su región no crítica y regresa al principio del bucle. Como el turno es 1, no puede entrar en su región crítica, pero el proceso 1 está ocupado con su región no crítica. Permanece inactivo en su bucle while hasta que el proceso 1 establezca el turno en 0. Conclusión, no es recomendable esta solución cuando uno de los procesos es mucho más lento que el otro.

```
int turn = 0;

while (TRUE) {
    while (turn != 0)    /* loop */ ;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(a) Process 0.

```
while (TRUE) {
    while (turn != 1)    /* loop */ ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

(b) Process 1.

EXCLUSION MUTUA - *PETERSON'S SOLUTION*

La solución de Peterson se limita a dos procesos que alternan la ejecución entre sus secciones críticas y las secciones restantes. Requiere que ambos procesos compartan dos datos:

```
int turn;  
boolean flag[2];
```

La variable `turn` indica a quién le toca entrar a su región crítica, si `turn == i`, el proceso P_i puede ejecutar su región crítica. El arreglo `flag` se utiliza para indicar si un proceso está listo para entrar a su sección crítica, si `flag[i] == true`, P_i está listo para entrar en su sección crítica.

```
#include <stdbool.h>  
  
volatile bool flag[2] = {false, false};  
volatile int turn;  
  
void process(int i)  
{  
    int j = 1 - i;  
    while (true)  
    {  
        flag[i] = true;  
        turn = j;  
        while (flag[j] && turn == j)  
        {  
            // Busy wait  
        }  
        // Critical section  
        // ...  
        flag[i] = false;  
        // Remainder section  
        // ...  
    }  
}  
  
// Peterson's solution is a classic algorithm for mutual exclusion in concurrent  
// programming. It allows two processes to share a single-use resource without conflict.
```

EXCLUSION MUTUA - *PETERSON'S SOLUTION*

Para entrar en la sección crítica, el proceso P_i primero establece `flag[i]` como verdadero y luego `turn` en el valor j , lo que garantiza que si el otro proceso desea entrar en la sección crítica, puede hacerlo. Si ambos procesos intentan entrar al mismo tiempo, `turn` se establecerá en i y j , y solo una de estas asignaciones permanecerá activa; la otra se ejecutará, pero se sobrescribirá inmediatamente. El valor final de `turn` determina cuál de los dos procesos puede ingresar primero a su sección crítica.

```
#include <stdbool.h>

volatile bool flag[2] = {false, false};
volatile int turn;

void process(int i)
{
    int j = 1 - i;
    while (true)
    {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j)
        {
            // Busy wait
        }
        // Critical section
        // ...
        flag[i] = false;
        // Remainder section
        // ...
    }
}

// Peterson's solution is a classic algorithm for mutual exclusion in concurrent
// programming. It allows two processes to share a single-use resource without conflict.
```

EXCLUSION MUTUA - *PETERSON'S SOLUTION*

Está demostrado que esta solución es correcta, en cuanto a:

- No permite dos procesos simultáneamente dentro de sus regiones críticas.
- Asegura progreso. Un proceso que se ejecute fuera de su región crítica no impide el ingreso de otro a su región crítica.
- Ningún proceso espera infinitamente para entrar en su región crítica.

```
#include <stdbool.h>

volatile bool flag[2] = {false, false};
volatile int turn;

void process(int i)
{
    int j = 1 - i;
    while (true)
    {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j)
        {
            // Busy wait
        }
        // Critical section
        // ...
        flag[i] = false;
        // Remainder section
        // ...
    }
}

// Peterson's solution is a classic algorithm for mutual exclusion in concurrent
// programming. It allows two processes to share a single-use resource without conflict.
```



EXCLUSION MUTUA - SOPORTE DE HARDWARE

- Test and Set Lock

Las computadoras diseñadas para múltiples procesadores, tienen una instrucción de hardware:

`TSL RX, LOCK`

Test and Set Lock, lee el contenido de la palabra en memoria `lock` en el registro `RX` y almacena un valor distinto de cero en la dirección de memoria `lock`. Esta instrucción garantiza que las operaciones de lectura y almacenamiento de la palabra en memoria son indivisibles: ningún otro procesador puede acceder a la palabra de memoria hasta que la instrucción finalice. La CPU que ejecuta la instrucción `TSL` bloquea el bus de memoria para impedir que otras CPU accedan a la memoria hasta que finalice.

`enter_region:`

```
TSL REGISTER, LOCK
CMP REGISTER, #0
JNE enter_region
RET
```

```
| copy lock to register and set lock to 1
| was lock zero?
| if it was not zero, lock was set, so loop
| return to caller; critical region entered
```

`leave_region:`

```
MOVE LOCK, #0
RET
```

```
| store a 0 in lock
| return to caller
```

Entering and leaving a critical region using the `TSL` instruction.



EXCLUSION MUTUA - SOPORTE DE HARDWARE

- Swap

Esta instrucción intercambia el contenido de dos ubicaciones atómicamente. Por ejemplo, un registro y una palabra de memoria.

```
XCHG RX, LOCK
```

enter_region:

MOVE REGISTER,#1	put a 1 in the register
XCHG REGISTER,LOCK	swap the contents of the register and lock variable
CMP REGISTER,#0	was lock zero?
JNE enter_region	if it was non zero, lock was set, so loop
RET	return to caller; critical region entered


leave_region:

MOVE LOCK,#0	store a 0 in lock
RET	return to caller

Entering and leaving a critical region using the XCHG instruction.



EXCLUSION MUTUA SIN BUSY WAITING



EXCLUSION MUTUA - *SLEEP AND WAKE UP*

Tanto la solución de Peterson como las que utilizan TSL o XCHG son correctas, pero ambas requieren espera activa (*busy waiting*). Es decir, cuando un proceso desea entrar en su región crítica, comprueba constantemente si la entrada está permitida, hasta que lo esté. Este enfoque desperdicia tiempo de CPU.

Ahora veremos algunas primitivas de comunicación entre procesos que bloquean en lugar de desperdiciar tiempo de CPU cuando no se les permite acceder a sus regiones críticas. Una de las más simples es el par de suspensión y activación. La suspensión es una llamada al sistema que bloquea al invocador, es decir, lo suspende hasta que otro proceso lo despierte. Y la llamada de activación tiene un parámetro: el proceso que se va a despertar.



EXCLUSION MUTUA - *SEMAPHORES*

En base a lo anterior, se propuso utilizar una variable entera para contar el número de reactivaciones guardadas para uso futuro. Este nuevo tipo de variable se denominó semáforo. Un semáforo podía tener el valor 0, lo que indicaba que no se guardaban reactivaciones, o un valor positivo si una o más reactivaciones están pendientes.

Esta propuesta consta de dos operaciones en los semáforos, "*down*" y "*up*" (generalizaciones de "*sleep*" y "*wake up*", respectivamente).

- La operación "*down*" en un semáforo comprueba si el valor es mayor que 0. Si es así, lo decrementa (es decir, consume una reactivación). Si es 0, el proceso se pone en reposo sin completar la operación "*down*". Comprobar el valor, modificarlo y, posiblemente, entrar en reposo se realizan como una única acción atómica indivisible. Se garantiza que, una vez iniciada una operación de semáforo, ningún otro proceso podrá acceder a él hasta que se complete o se bloquee. Esta atomicidad es absolutamente esencial para resolver problemas de sincronización y evitar condiciones de carrera.



EXCLUSION MUTUA - *SEMAPHORES*

- La operación “*up*” incrementa el valor del semáforo. Si uno o más procesos estaban inactivos en ese semáforo, sin poder completar una operación “*down*”, el sistema elige uno de ellos (por ejemplo, al azar) y se le permite completar su operación “*down*”. Por lo tanto, tras un “*up*” en un semáforo con procesos inactivos, el semáforo seguirá siendo 0, pero habrá un proceso menos inactivo. Estas acciones de incrementar el semáforo y de despertar un proceso también son indivisibles, ningún proceso puede bloquearlas.

De esta manera, una solución del problema productor-consumidor mediante la utilización de semáforos sería la siguiente:

SEMAPHORES

Esta solución utiliza tres semáforos:

- uno llamado "full" para contar el número de *slots* ocupados
- otro llamado "empty" para contar el número *slots* disponibles
- y otro llamado "mutex" para garantizar que el productor y el consumidor no accedan al búfer simultáneamente.

Inicialmente `full` es 0, `empty` es igual al tamaño del búfer y `mutex` es 1.

Los semáforos que se inicializan en 1 y son utilizados por dos o más procesos para garantizar que solo uno de ellos pueda entrar en su región crítica simultáneamente se denominan semáforos binarios.

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        /* infinite loop */
        /* decrement full count */
        /* enter critical region */
        /* take item from buffer */
        /* leave critical region */
        /* increment count of empty slots */
        /* do something with the item */
    }
}
```

The producer-consumer problem using semaphores.

SEMAPHORES

En este ejemplo se han utilizado semáforos de dos maneras diferentes. El semáforo `mutex` se utiliza para la exclusión mutua. Está diseñado para garantizar que solo un proceso a la vez lea o escriba en el búfer.

El otro uso de los semáforos es la sincronización. Los semáforos `full` y `empty` son necesarios para garantizar que el productor deje de ejecutarse cuando el búfer está lleno y que el consumidor deje de ejecutarse cuando está vacío. Este uso es diferente de la exclusión mutua.

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }

    /* TRUE is the constant 1 */
    /* generate something to put in buffer */
    /* decrement empty count */
    /* enter critical region */
    /* put new item in buffer */
    /* leave critical region */
    /* increment count of full slots */
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }

    /* infinite loop */
    /* decrement full count */
    /* enter critical region */
    /* take item from buffer */
    /* leave critical region */
    /* increment count of empty slots */
    /* do something with the item */
}
```

The producer-consumer problem using semaphores.



EXCLUSION MUTUA - *MUTEX*

Cuando no se requiere la capacidad de conteo del semáforo, se puede utilizar una versión simplificada, llamada mutex. La misma solo sirven para gestionar la exclusión mutua en algún recurso compartido. Son especialmente útiles en librerías de subprocesos (*threads*) implementadas completamente en el espacio de usuario.

Un mutex es una variable compartida que puede estar en uno de dos estados: desbloqueado o bloqueado, donde 0 significa desbloqueado y todos los demás valores significan bloqueado.

Cuando un *thread* (o proceso) necesita acceder a una región crítica, invoca `mutex_lock`. Si el mutex está desbloqueado (ie. la región crítica está disponible), la llamada se realiza correctamente y el *thread* que lo invoca puede acceder a la región crítica.

Por otro lado, si el mutex está bloqueado, el *thread* que lo invoca se bloquea hasta que el *thread* en la región crítica finalice y ejecute `mutex_unlock`. Si varios *thread* están bloqueados en el mutex, se elige uno de ellos al azar y se le permite adquirir el *lock*.

MUTEX IN PTHREADS

Pthreads proporciona varias funciones sobre mutex para sincronizar *threads*.

- `pthread_mutex_init` crea un mutex.
- `pthread_mutex_destroy` destruye un mutex.
- `pthread_mutex_lock` intenta obtener el bloqueo y si no puede, se bloquea.
- `pthread_mutex_trylock` intentar obtener el bloqueo y si no puede, falla con un código de error. Esta llamada permite que un hilo realice una espera activa.
- `pthread_mutex_unlock` desbloquea un mutex y libera exactamente un hilo si hay uno o más esperando en él.

Thread call	Description
<code>Pthread_mutex_init</code>	Create a mutex
<code>Pthread_mutex_destroy</code>	Destroy an existing mutex
<code>Pthread_mutex_lock</code>	Acquire a lock or block
<code>Pthread_mutex_trylock</code>	Acquire a lock or fail
<code>Pthread_mutex_unlock</code>	Release a lock

Some of the Pthreads calls relating to mutexes.

MUTEX IN PTHREADS

Además de mutex, *Pthreads* ofrece un segundo mecanismo de sincronización: las variables de condición. Los mutex sirven para permitir o bloquear el acceso a una región crítica. Las variables de condición permiten que los hilos se bloqueen si no se cumple alguna condición. Casi siempre se utilizan ambos métodos juntos.

- `pthread_cond_init` crea una variable de condición.
- `pthread_cond_destroy` destruye una variable.
- `pthread_cond_wait` bloquea al hilo que realiza la llamada hasta que otro hilo le envíe una señal.

Thread call	Description
<code>Pthread_cond_init</code>	Create a condition variable
<code>Pthread_cond_destroy</code>	Destroy a condition variable
<code>Pthread_cond_wait</code>	Block waiting for a signal
<code>Pthread_cond_signal</code>	Signal another thread and wake it up
<code>Pthread_cond_broadcast</code>	Signal multiple threads and wake all of them

Some of the Pthreads calls relating to condition variables.

MUTEX IN PTHREADS

- `pthread_cond_signal` envía una señal para desbloquear un hilo bloqueado.
- `pthread_cond_broadcast` se utiliza cuando hay varios hilos potencialmente bloqueados y esperando la misma señal.

El hilo que bloquea en general espera a que el hilo que realiza la señalización realice algún trabajo, libere algún recurso o realice alguna otra actividad. Sólo entonces puede continuar el hilo que bloquea. Las variables de condición permiten que esta espera y bloqueo se realicen de forma automática.

Thread call	Description
<code>Pthread_cond_init</code>	Create a condition variable
<code>Pthread_cond_destroy</code>	Destroy a condition variable
<code>Pthread_cond_wait</code>	Block waiting for a signal
<code>Pthread_cond_signal</code>	Signal another thread and wake it up
<code>Pthread_cond_broadcast</code>	Signal multiple threads and wake all of them

Some of the Pthreads calls relating to condition variables.

PRODUTOR - CONSUMIDOR

```
#include <stdio.h>
#include <pthread.h>

#define MAX 10 /* how many numbers to produce */

pthread_mutex_t the_mutex; /* used for mutual exclusion */
pthread_cond_t condc, condp; /* used for signaling */

int buffer = 0; /* buffer used between producer and consumer */

/* main function */
int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);

    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);

    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

```
/* producer and consumer are the two threads */
void *producer(void *ptr) /* produce data */
{
    int i;
    for (i = 1; i <= MAX; i++)
    {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0)
            pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

/* consumer and producer are the two threads */
void *consumer(void *ptr) /* consume data */
{
    int i;
    for (i = 1; i <= MAX; i++)
    {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0)
            pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}
```

Muchas Gracias

Jeremías Fassi

Javier E. Kinter

