Sistemas Embebidos Avanzados II

DSI

Ultima actualización: August 26, 2021.
Actualizaciones del documento y fuentes:
`https://`
Correcciones, sugerenias, contribuciones y traducciones son bienvenidas!

Derechos de copia

Document sources: `https://github.com/bootlin/training-materials/`

Concurrencia

DSI

Procesos

Introducción a Procesos

- Todas las computadoras moderas realizan varias tareas al mismo tiempo
- En cada instante la CPU ejecuta un único programa
- Existen programas con un único hilo de control
- Los sistemas de tiempo real son inherentemente concurrentes

Un único procesador puede compartirse entre varios procesos utilizando un algoritmo de planificación que determine cuándo hay que detener el trabajo sobre un proceso y pasar a atender a otro diferente.

El modelo de los procesos secuenciales

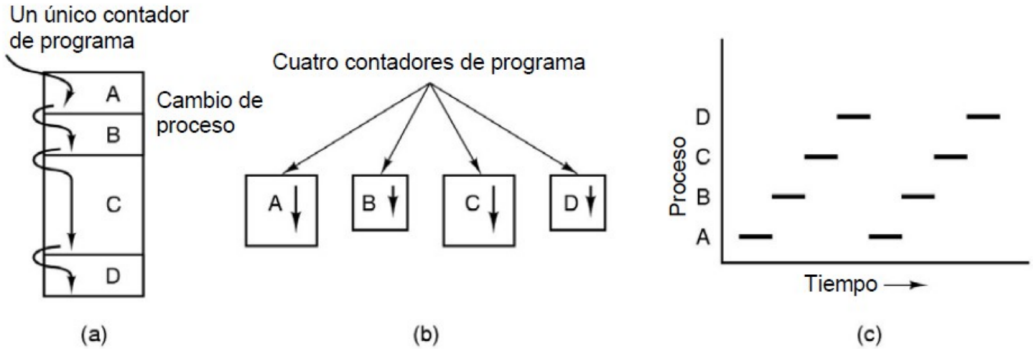


Figura 2-1. (a) Multiprogramación de cuatro programas. (b) Modelo conceptual de cuatro procesos secuenciales independientes. (c) Sólo un programa está activo en cada momento.

El modelo de los procesos secuenciales

- La CPU conmuta de un proceso a otro
- La velocidad a la cual un proceso realiza su cómputo no es uniforme y probablemente ni siquiera es reproducible
- Los procesos no deben programarse bajo suposiciones preconcebidas sobre su velocidad de ejecución
- Cuando un proceso tiene requerimientos de tiempo real críticos es necesario tomar medidas especiales para asegurar que efectivamente los sucesos ocurran dentro de ciertos límites de tiempo

Procesos concurrentes

- El trabajar con procesos concurrentes añade complejidad a la tarea de programar
- ¿Cuáles son entonces los beneficios que aporta la programación concurrente?

Beneficios de la programación concurrente

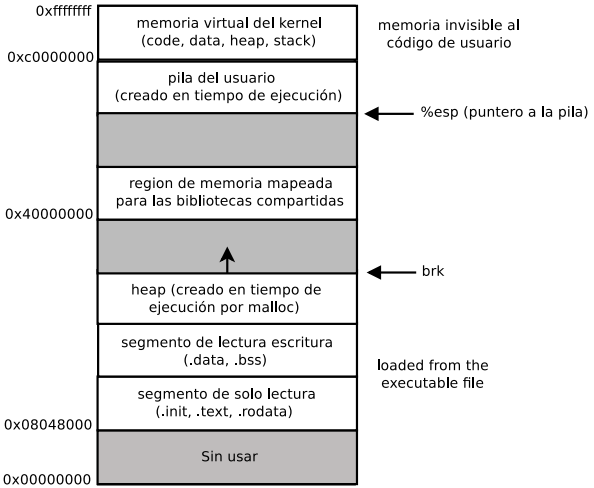
- Mejor aprovechamiento de la CPU
- Velocidad de ejecución
- Solución de problemas de naturaleza concurrente:
  - Sistemas de control
  - Tecnologías web
  - Aplicaciones basadas en interfaces de usuarios
  - Simulación
  - SGDB

- Es una instancia de ejecución de un programa
- Además del código ejecutable (.text) incluye:
  - un contador de programa
  - contenido de registros del procesador
  - pila (stack): con datos temporales como parámetros de funciones, variables locales, etc.
  - sección de datos: variables globales
  - heap: memoria alocada dinámicamente
- Tiene un ciclo de vida, es decir pasa por distintos estados

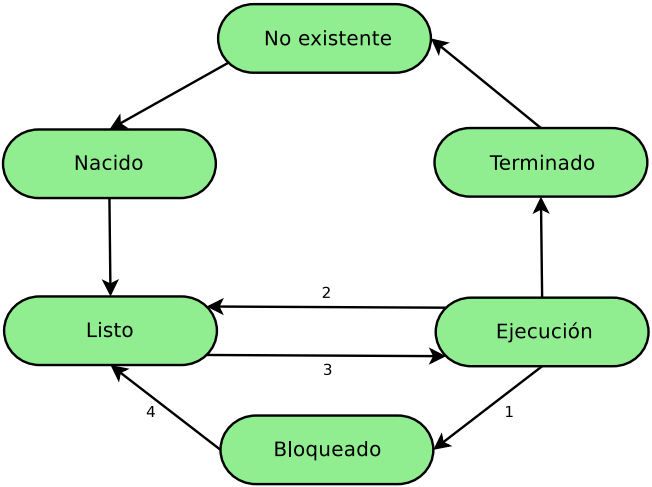
Diseño de la memoria

- A todo proceso se le asigna un espacio de direccionamiento que representa las zonas de memoria asignadas al proceso.
- Este espacio incluye:
  - El código del proceso
  - Los datos del proceso. Se divide en variables inicializadas y no inicializadas (.bss)
  - Código y datos de bibliotecas.
  - La pila

Diseño de memoria



Ciclo de vida de un proceso



1 El proceso se bloquea esperando un dato

2 El planificador selecciona otro proceso

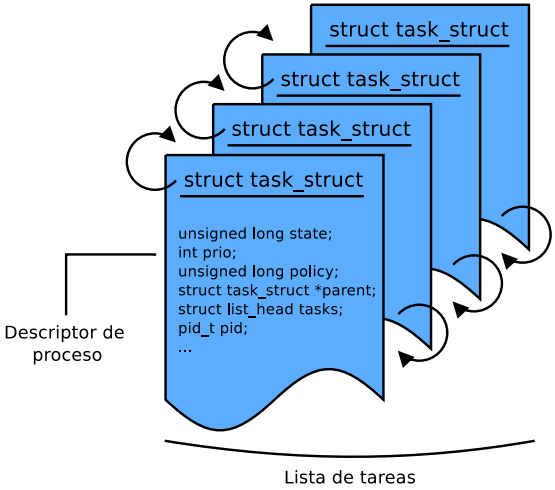
3 El planificador selecciona este proceso

4 El dato está displonible

Atributos de un proceso

- Estado
- Identificadores (proceso, usuario, grupo, etc.)
- Valor de registros, incluyendo PC
- Identidad del usuario
- Prioridad (política y mecanismos de planificación)
- Espacio de almacenamiento
- Descriptores de archivos

Descriptor de un proceso



Creación de un proceso

La creación de un proceso se da en los siguientes momentos:

1. La inicialización del sistema
2. La ejecución por parte de un proceso (en ejecución) de una llamada al sistema de creación de un nuevo proceso.
3. La petición por parte del usuario de la creación de un nuevo proceso.
4. El inicio de un trabajo en batch.

Finalización de un proceso

La finalización de un proceso se puede dar en los siguientes casos:

1. Finaliza la ejecución de su cuerpo
2. Ejecución de alguna sentencia de auto finalización
3. Condición de error sin tratar
4. Aborto por medio de la intervención de otro proceso
5. Nunca: procesos que se ejecutan en bloques que no terminan
6. Cuando ya no son necesarios

Cambio de contexto de un proceso

- Guardar el contexto del proceso (registros, contador, direccionamiento, etc)
- Cambiar el estado y guardar información acerca de la interrupción
- Seleccionar un programa para su ejecución
- Restaurar el contexto del nuevo programa
- Ejecutar el programa seleccionado

Ejecución de los procesos

- Todos los SO tienen formas de crear procesos
- Cada proceso se ejecuta en su propia máquina virtual
- `fork()` crea un nuevo proceso duplicando al proceso que lo invoca

Creación de procesos en Linux

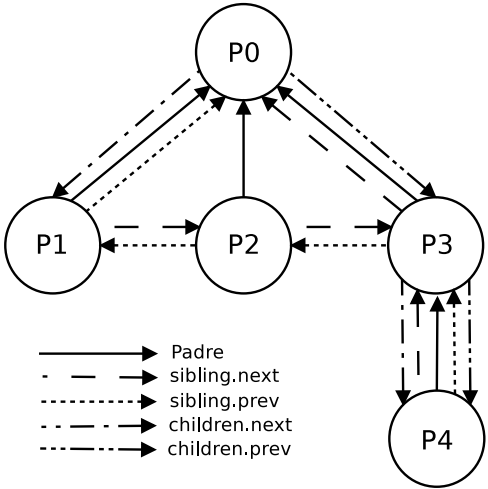
```c
#include <wait.h>
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int value = 5;

int main(void)
{
    pid_t pid;

    pid = fork(); /* Retorna 0 en el hijo y el pid del hijo en el padre */

    if (pid == 0) {
        /* proceso hijo */
        value += 15;
    } else if (pid > 0) {
        /* proceso padre */
        wait(NULL);
        printf("Padre: valor = %d\n", value);
        exit(EXIT_SUCCESS);
    }
}
```

Creación de procesos en Linux

- El nuevo proceso es llamado proceso *hijo* y es una copia exacta del proceso *padre*
- `fork()` retorna 0 al hijo y el PID del hijo al padre
- La utilización de recursos es limpiada a 0 en el hijo.
- El espacio de direcciones es replicado (se copian todas las variables y archivos abiertos).
- Ejemplo: intérpretes de comandos.

Jerarquía de procesos

Relaciones de parentesco entre cinco procesos



Comunicación entre procesos: IPC

- Memoria compartida
- Tuberías (pipes)
- Paso de mensajes
- Semáforos

Esta comunicación es costosa

Clonado de procesos

- Existe en Linux la llamada al sistema `clone()` que crea un proceso por duplicación de su padre
- Puede compartir sólo una parte de su contexto con su padre
- Espacio de direccionamiento (segmentos de código y datos)
- Información de control del sistema de archivos (directorios raíz y actual)
- Premite configurar que compartir a través de una lista de argumentos

Clonado de procesos

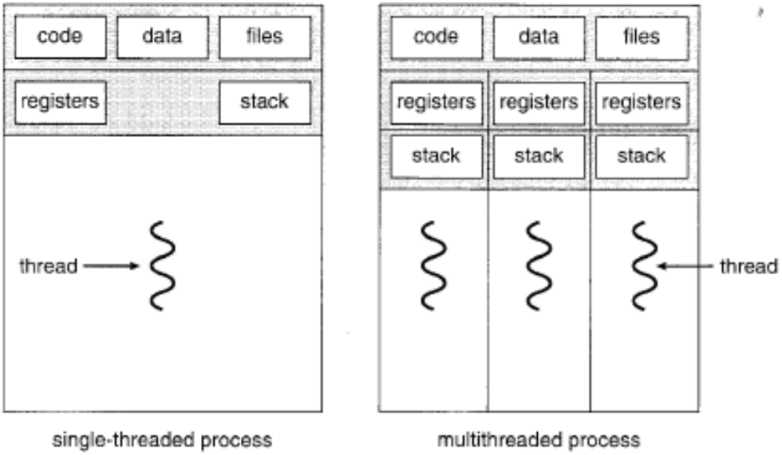
- Descriptores de archivos abiertos
- Gestores de señales
- Identificador de procesos (ambos procesos comparten el mismo número)
- Esto permite ejecutar varias actividades sin necesidad de IPC, compartiendo simplemente datos
- Los SO modernos permiten crear hilos (threads), también llamados procesos ligeros, dentro de la misma máquina virtual

Hilos

Introducción a los Hilos

- Hay situaciones en las que es deseable contar con múltiples hilos de control (threads) en el mismo espacio de direcciones ejecutándose quasi-paralelamente, como si fueran procesos independientes
- Tener múltiples hilos ejecutándose en paralelo dentro de un proceso es análogo a tener múltiples procesos ejecutándose en paralelo dentro de un ordenador
- Comprenden un ID, un contador de programa, un conjunto de registros y una pila
- Comparten con otros hilos pertenecientes al mismo proceso: la sección de código, la sección de datos y otros recursos del SO tales como archivos abiertos y señales
- No son jerárquicos
- Se los suele llamar **procesos ligeros** (*lightweight process*).
- También se utiliza el término de **multihilo** (*multithreaded*) para describir la situación en la cual se permite que haya múltiples threads en el mismo proceso

Hilos


code
data
files
registers
stack
thread
single-threaded process
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded process

Ciclo de vida de los Hilos

- Un hilo puede estar en cualquiera de los estados de un proceso tradicional
  - Un hilo en **ejecución** tiene actualmente la CPU y está activo
  - Un hilo **bloqueado** está esperando que algún suceso lo desbloquee. Un hilo puede bloquearse esperando que tenga lugar algún suceso externo o que algún otro hilo lo desbloquee
  - Un hilo **listo** está planificado para ejecutarse y lo hace tan pronto como le llega su turno
- Las transiciones entre los estados de un hilo son las mismas que las transiciones entre los estados de un proceso
- Los cambios de contexto son mucho más rápidos

Utilización de Hilos

- Mejora el tiempo de respuesta
  - Por ejemplo un Navegador multihilo puede permitir interacción con un usuario mientras se carga una imagen
- Son más fáciles de crear y destruir que los procesos
  - En numerosos sistemas, la creación de un hilo puede realizarse 100 veces más rápido que la creación de un proceso
- Memoria compartida $=>$ Comunicación

Ejemplo

- Un procesador de texto
  - La mayoría de los procesadores de texto visualizan en la pantalla el documento que se está creando formateado exactamente como aparecería una vez impreso
  - Todos los saltos de línea y de página aparecen en su posición correcta final

Ejemplo

- Supongamos que el usuario está escribiendo un libro
  - Desde el punto de vista del autor es más cómodo meter el libro entero en un único archivo (búsqueda por temas, sustituciones globales, etc)
  - Alternativamente, puede ponerse cada capítulo en un archivo separado
- Se borra una frase de la página 1 de un documento de 800 páginas
- Se busca la página 600. El procesador de texto se ve forzado a reformatear inmediatamente todo el libro hasta la página 600
- Espera considerable

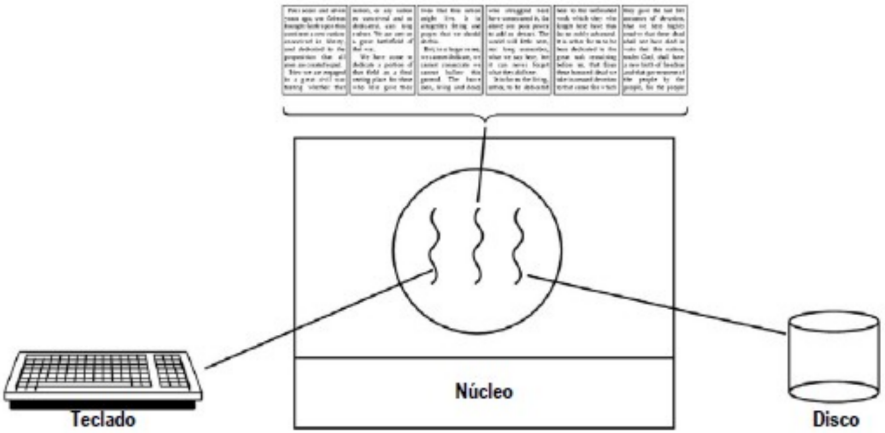
Ejemplo

- Los hilos pueden ayudarnos
- Un hilo interactúa con el usuario (atiende teclado, mouse, scrolls).
- Otro realiza el reformateo como una actividad de fondo (tan pronto como se borra la frase de la página 1)
- Con un poco de suerte, el reformateo se completa antes de que el usuario pida ver la página 600, de forma que en ese momento puede visualizarse instantáneamente

Ejemplo

- Muchos procesadores de texto ofrecen la posibilidad de guardar automáticamente todo el archivo en el disco cada pocos minutos
- El tercer hilo puede ocuparse de las copias de seguridad en el disco sin interferir con los otros dos

Ejemplo



Un procesador de texto con tres threads.

¿Cuando utilizar Hilos?

- Desventajas de trabajar con hilos:
  - Protección de variables compartidas (concurrencia)
  - Deadlocks, inversión de prioridades
  - Dificultad para depurar
- Se recomienda la utilización cuando:
  - Se realizan cálculos intensivos en paralelo
  - Existen actividades asíncronas
  - Sistemas de tiempo real
  - Sistemas distribuidos

Programación concurrente

Los lenguajes concurrentes tienen elementos para:

- Crear procesos
- Sincronizar procesos
- Comunicar procesos

Comportamiento de procesos

- Independientes: no se sincronizan ni comunican (son muy raros).
- Cooperativos: se comunican y sincronizan sus actividades.
- Competitivos: compiten por recursos del sistema

Sincronizar y Comunicar

- Sincronizar: Satisfacer las restricciones en el enlazado de las acciones de los distintos procesos
- Comunicar: pasar información de un proceso a otro.
- Variables compartidas: objetos a los que puede acceder más de un proceso
- Paso de mensajes: intercambio explícito de datos entre dos procesos mediante el paso de un mensaje mediante alguna forma que brinda el SO o el propio lenguaje

Modelo de concurrencia

- Estructura: nro de procesos fijo o variable.
- Nivel: paralelismo soportado
- Granularidad: muchos o pocos procesos
- Inicialización: paso de parámetros, o comunicación explícita después de su ejecución
- Finalización: al completar la ejecución, error, aborto, nunca, suicidio, excepción sin manejar, cuando no se necesita más necesarios

Variables compartidas

Considere dos procesos que actualizan una variable compartida, `X`, mediante la sentencia: `x = x + 1`

1. Carga el valor de `x` en algún registro
2. Incrementa el valor en el registro en `1`
3. Almacena el valor del registro de nuevo en `x`

Como ninguna de las tres operaciones es indivisible, dos procesos que actualicen la variable simultáneamente generarían un entrelazamiento que podría producir un resultado incorrecto.

Variables compartidas

- Las situaciones donde los resultados dependen del orden en que se ejecutan los procesos se llaman **Condiciones de Competencia**.
- Las partes de un proceso que tienen acceso a las variables compartidas han de ejecutarse indivisiblemente unas respecto a las otras.
- Estas partes se denominan **Secciones Críticas**
- La protección requerida se conoce como **Exclusión Mutua**

Secciones o regiones críticas

- Para obtener una solución se deben cumplir:
  1. Ningún par de procesos pueden estar simultáneamente dentro de sus regiones críticas
  2. No debe hacerse ninguna suposición sobre la velocidad o el número de CPUs
  3. Ningún proceso fuera de su región crítica puede bloquear a otros procesos
  4. Ningún proceso deberá tener que esperar infinitamente para entrar en su región crítica

Exclusión Mutua

Exclusión Mutua con Espera Ocupada

- Posibles soluciones:
  - Inhabilitación de interrupciones
  - Variables de cerradura
  - Alternancia estricta
  - Solución de Peterson

Inhabilitación de Interrupciones

- Se inhabilitan las interrupciones antes de entrar a la sección crítica. Se rehabilitan al salir.
- Ventaja: muy simple
- Desventajas:
  - Los procesos de usuario no pueden deshabilitar las interrupciones
  - Si hay más de un procesador no funciona, pues la inhabilitación afecta a un solo CPU

Inhabilitación de Interrupciones

```
while (cierto)
{
    /* inhabilitar interrupciones */;
    /* sección critica */;
    /* habilitar interrupciones */;
    /* resto */;
}
```

Variables de Cerradura

- Variable compartida inicialmente en `0`
- Antes de entrar a su sección crítica un proceso chequea la bandera:
  - Si está `0`, el proceso la setea en `1` y entra a su sección crítica.
  - Si está en `1`, espera hasta que se ponga en `0`
- Desventajas:
  - Espera ocupada.
  - No funciona por condiciones de competencia.

Alternancia Estricta

```
while(TRUE) {
    while (turno != 0); /* nada */
    region_critica();
    turno = 1;
    region_no_critica();
}
```

```
while(TRUE) {
    while (turno != 1); /* nada */
    region_critica();
    turno = 0;
    region_no_critica();
}
```

- Desventajas:
  - Ineficiente si un proceso es más lento que el otro
  - Espera ocupada
  - Cuando un proceso no esta en la sección crítica igualmente tiene bloqueado el acceso a la misma y por lo tanto no permite que otro proceso que requiera ingresar a la misma logre hacerlo

Solución de Peterson

- permite a dos o más procesos o hilos de ejecución compartir un recurso sin conflictos, utilizando sólo memoria compartida para la comunicación
- simplificación del algoritmo de Dekker para dos procesos
- posteriormente fue generalizado para que funcione para N procesos

Solución de Peterson

```
int turno;
int interesado[2];

void entrar_en_region(int proceso)
{
    int otro;
    otro = 1 - proceso;
    interesado[proceso] = TRUE;
    turno = proceso;
    while (turno == proceso && interesado[otro] == TRUE);
}

void abandonar_region(int proceso)
{
    interesado[proceso] = FALSE;
}
```

Comunicación entre procesos (IPC)

Primitivas de IPC con bloqueo

- La solución de Peterson es correcta, pero tiene el defecto de requerir espera ocupada:
  - Cuando un proceso quiere entrar en su región crítica, comprueba si se le permite la entrada
  - Si no es así, el proceso se mete en un bucle vacío esperando hasta que sí se le permita entrar.
- Este método no sólo gasta tiempo de CPU, sino que también puede tener efectos inesperados.

- Consideremos una computadora con dos procesos, uno H con alta prioridad y otro L con baja prioridad
- Las reglas de planificación son tales que H pasa a ejecución inmediatamente siempre que se encuentre en estado listo.
- En un cierto momento, estando L en su región crítica, H pasa al estado listo (por ejemplo, debido a que se completa una operación de E/S que lo mantenía bloqueado).

Inversión de Prioridades

- De inmediato H comienza la espera activa, pero ya que L nunca se planifica mientras H esté ejecutándose, L nunca tendrá la oportunidad de abandonar su región crítica, con lo que H quedará para siempre dando vueltas al bucle de espera activa.
- Este es el problema de la Inversión de Prioridades

Primitivas de IPC con bloqueo

- Son primitivas que bloquean al proceso que las invoca
- Permite a la CPU continuar sin desperdiciar tiempo como en la espera ocupada.
  - Dormir y despertar
  - Semáforos
  - Monitores
  - Métodos sincronizados

Dormir y Despertar

- **Sleep** es una llamada al sistema que provoca que el proceso que la invoca se bloquee, esto es, se suspenda hasta que otro proceso lo despierte
- **Wakeup** tiene un parámetro, que es el proceso a ser despertado

El problema del Productor-Consumidor

- Son aquellos problemas en los que existe un conjunto de procesos que producen información que otros procesos consumen, siendo diferentes las velocidades de producción y consumo de la información.
- Este desajuste en las velocidades, hace necesario que se establezca una sincronización entre los procesos de manera que la información no se pierda ni se duplique, consumiéndose en el orden en que es producida

El problema del Productor-Consumidor

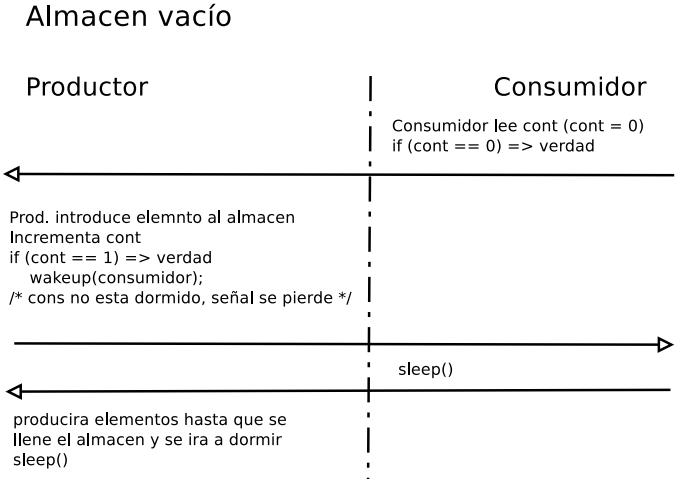
```
#define 100
int contador = 0;

void productor(void)
{
    int elemento;

    while (TRUE) {
        elemento = producir_elemento();
        if (contador == N) sleep();
        meter_elemento(elemento);
        contador = contador + 1;
        if (contador == 1) wakeup(consumidor);
    }
}
```

```
void consumidor (void)
{
    int elemento;

    while (TRUE) {
        if (contador == 0) sleep();
        elemento = sacar_elemento();
        contador = contador - 1;
        if (contador == N - 1) wakeup(productor);
        consumir_elemento(elemento);
    }
}
```

El problema del Productor-Consumidor

- Existen condiciones de competencia
- La esencia del problema es la pérdida de una señal enviada para despertar a un proceso que no estaba (todavía) dormido.

El problema del Productor-Consumidor cont



El problema del Productor-Consumidor

- Posible solución: añadir un bit de espera por la señal que despierta al proceso (**wakeup waiting bit**).
- Este bit se activa cuando se envía una señal para despertar a un proceso que todavía está despierto. Posteriormente, cuando el proceso intente dormirse, si encuentra ese bit activo, simplemente lo desactiva permaneciendo despierto
- Existen ejemplos con tres o más procesos en los cuales un único bit de este tipo es insuficiente.

Semáforos

- Los introdujo Dijkstra en 1968
- Permiten resolver la mayoría de los problemas de sincronización entre procesos y forman parte del diseño de muchos sistemas operativos y de lenguajes de programación concurrentes
- Dos o más procesos pueden cooperar por medio de simples señales
- La señalización se realiza a través de variables especiales llamadas semáforos
- La idea es utilizar una variable entera para contar el número de señales enviadas para despertar un proceso y guardarlas para su uso futuro

- Introdujo un nuevo tipo de variable, denominado **semáforo**
- El valor de un semáforo puede ser 0, indicando que no se ha guardado ninguna señal, o algún valor positivo de acuerdo con el número de señales pendientes para despertar al proceso
- Operaciones sobre los semáforos: **down** y **up** (las cuales generalizan las operaciones *sleep* y *wakeup*, respectivamente).

- Está garantizado que una vez que comienza una operación sobre un semáforo, ningún otro proceso puede acceder al semáforo hasta que la operación se completa o hasta que el proceso se bloquea
- La comprobación del valor del semáforo, su modificación y la posible acción de dormirse, se realizan como una única **acción atómica** indivisible (no puede ser interrumpida). Esta atomicidad es absolutamente esencial para resolver los problemas de sincronización y evitar que se produzcan condiciones de competencia

Semáforos

- La operación **up** incrementa el valor del semáforo al cual se le aplica
- Si uno o más procesos estuviesen dormidos sobre ese semáforo, el sistema elige a uno de ellos (por ejemplo de forma aleatoria) y se le permite continuar
- El proceso que usa **down** decrementa el semáforo

Semáforos

```
public final class Semaforo {

    private int s;

    public Semaforo (int inicial) {
        s = inicial;
    }

    public synchronized void down () {
        if (s == 0) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        s--;
    }

    public synchronized void up () {
        s++;
        notify();
    }
}
```

- La operación **up** nunca bloquea al proceso que la ejecuta, de la misma forma que en el modelo anterior nunca se bloquea un proceso ejecutando una operación wakeup
- La nomenclatura utilizada varía según el autor:

`P()`	`V()`
`down(S)`	`up(S)`
`sleep(S)`	`wakeup(S)`
`wait(S)`	`notify()`
`Espera(S)`	`Signal(S)`

Solución al problema del Productor-Consumidor con semáforos

```
tuberia.cantidadOcupados.down();
tuberia.mutex.down();
c = tuberia.consumir();
tuberia.mutex.up();
tuberia.cantidadVacios.up();
```

```
   tuberia.cantidadVacios.down();
tuberia.mutex.down();
tuberia.producir(c);
tuberia.mutex.up();
   tuberia.cantidadOcupados.up();
```

Solución al problema del Productor-Consumidor con semáforos

- Hemos utilizado los semáforos de dos formas muy distintas:
  - El semáforo *mutex* se utiliza para conseguir la exclusión mutua. Está diseñado para garantizar que solamente un proceso esté en cada momento leyendo o escribiendo en el búfer y sus variables asociadas. Cada proceso hace un **down** justo antes de entrar en su región crítica, y un **up** justo después de abandonarla
  - El otro uso de los semáforos es la **sincronización**

Características de los Semáforos

- La manera normal es implementar **down** y **up** como llamadas al sistema, encargándose el sistema operativo de inhibir brevemente todas las interrupciones mientras comprueba el valor del semáforo, lo actualiza y bloquea el proceso, si es necesario
- Como todas estas acciones requieren pocas instrucciones, no se provoca ningún daño al sistema inhibiendo las interrupciones durante ese corto lapso de tiempo

Críticas a los semáforos

- Es difícil programar con semáforos
- Si el semáforo se ubicó en un lugar erróneo falla (deadlock).

No se garantiza la exclusión mutua

Posible solución: monitores

Monitores

- Para hacer más fácil escribir programas correctos, Hoare (1974) y Brinch Hansen (1975) propusieron una primitiva de sincronización de alto nivel denominada **monitor**
- Sus propuestas difieren ligeramente

Monitores

- Un monitor es una colección de procedimientos, variables y estructuras de datos que están todos agrupados juntos en un tipo especial de módulo o paquete
- Los procesos pueden llamar a los procedimientos de un monitor siempre que quieran, pero no se les permite acceder directamente a las estructuras de datos internas del monitor desde procedimientos declarados fuera del monitor

Monitores

- En cualquier instante solamente un proceso puede estar activo dentro del monitor
- Los monitores son construcciones del lenguaje, por lo que el compilador sabe que son especiales, de manera que puede tratar las llamadas a los procedimientos del monitor de forma diferente que a otras llamadas a procedimientos normales

```
Adquiere el Mutex
// sección crítica
Libera el Mutex
```

- Si otro proceso está actualmente activo dentro del monitor, el proceso que hizo la llamada debe suspenderse hasta que el otro proceso abandone el monitor
- Si ningún otro proceso está utilizando el monitor, el proceso que hizo la llamada puede entrar inmediatamente

Críticas a monitores

- Solución elegante a problemas de exclusión mutua
- Sólo lo implementan muy pocos lenguajes: Pascal Concurrente y Java (métodos sincronizados)
- No proporciona intercambio de información entre diferentes máquinas

Métodos sincronizados

- Es el concepto de monitor implementado en el paradigma de Orientación a Objetos
- Se utilizan en lenguajes como Java, que tiene la concurrencia totalmente integrada
- Los métodos se califican con el modificador **synchronized**
- Puede existir también **synchronized** a nivel de bloque

Métodos sincronizados

- **notify()** despierta un único hilo entre los que están esperando en el monitor del objeto
- **wait()** provoca que el hilo invocante espere hasta que otro hilo invoque el método **notify()** para ese objeto
- Ejemplo Productor-Consumidor con Monitores en Java

PThreads

La biblioteca pthreads

- pthreads (POSIX threads) es una bilbioteca para trabajar con hilos
- Puede ser utilizada con hilos del usuario o del kernel

¿Que es POSIX?

- Portable Operating System Interface
- Es un estandar para unificar los programas y las llamadas al sistema que proveen los diferentes Sistemas Operativos

Crear nuevos hilos

```
#include <pthreads.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine) (void *), void *arg);
```

- Compilar y enlazar con `-pthread`
- La función `pthread_create()` inicia un nuevo hilo en la llamada.
- El nuevo hilo comienza la ejecución invocando `start_routine()`
- `arg` se pasa como el único argumento de `start_routine()`.

Esperar que finalice un hilo

```c
#include <pthreads.h>

int pthread_join(pthread_t thread, void **retval);
```

- Compilar y enlazar con `-pthread`
- La función `pthread_join()` espera el hilo especificado por `thread` para Terminar
- Si ese hilo ya ha terminado, entonces `pthread_join()` regresa inmediatamente

thread.c

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

void* say_hello(void* data) {
    char *str;
    str = (char*)data;
    while(1) {
        printf("%s\n",str);
        sleep(1);
    }
}
void main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, say_hello, "hello from 1");
    pthread_create(&t2, NULL, say_hello, "hello from 2");
    pthread_join(t1, NULL);
}
```

Inicializar un semáforo

```c
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- Compilar y enlazar con `-pthread`
- `sem_init()` inicializa el semáforo en la dirección apuntada a por `sem`
- El argumento de `value` especifica el valor inicial para el semáforo
- El argumento `pshared` indica si este semáforo debe ser compartido entre los hilos de un proceso, o entre procesos
- Si `pshared` tiene el valor 0, entonces el semáforo se comparte entre los subprocesos de un proceso, y debe estar ubicado en alguna dirección que sea visible para todos los hilos (por ejemplo, una variable global o una variable asignada dinámicamente en el montón).

Bloquear un semáforo

```c
#include <semaphore.h>

int sem_wait(sem_t *sem);
```

- Compilar y enlazar con `-pthread`
- `sem_wait()` decrementa (bloquea) el semáforo apuntado por `sem`
- Si el valor del semáforo es mayor que cero, entonces el decremento procede, y la función regresa, inmediatamente.
- Si el semáforo actualmente tiene el valor cero, luego la llamada se bloquea hasta que sea posible realizar la disminución (es decir, el valor del semáforo sube por encima de cero), o un manejador de señales interrumpe la llamada

Desbloquear un semáforo

```c
#include <semaphore.h>

int sem_post(sem_t *sem);
```

- Compilar y enlazar con `-pthread`
- `sem_post()` incrementa (desbloquea) el semáforo al que apunta `sem`
- Si el valor del semáforo se vuelve mayor que cero, entonces otro proceso o subproceso bloqueado en una llamada `sem_wait(3)` se despertará y procederá a bloquear el semáforo

```c
#include <pthread.h>
#include <string.h>
#include <semaphore.h>

sem_t binaySemphore;

void* thread_1_function(void *ptr);
void* thread_2_function(void *ptr);

int main() {
    int iRet;
    pthread_t thread_1;
    pthread_t thread_2;

    unsigned char ucBuff[10];
    sem_init(&binaySemphore, 0, 1);
    strcpy(ucBuff, "thread_1");
    iRet = pthread_create(&thread_1, NULL, thread_1_function, (void *)ucBuff);

    if (iRet == 0) {
        printf("pthread 1 created...\n");
    }
```

```
    sleep(1);
    strcpy(ucBuff, "thread_2");
    iRet = pthread_create(&thread_2, NULL, thread_2_function, (void *)ucBuff);

    if(iRet == 0) {
        printf("pthread 2 created...\n");
    }

    pthread_join(thread_1, NULL);
    pthread_join(thread_2, NULL);
    sem_destroy(&binaySemphore); /* destroy semaphore */
    exit(0);
}

/* prototype for thread routine */
void* thread_1_function(void *ptr) {
    unsigned char* ucBuffPtr, ucThreadBuff[10];
    ucBuffPtr = (unsigned char *) ptr;
    strcpy(ucThreadBuff, ucBuffPtr);
    printf("thread_1_function entered\n");
```

```
    while(1) {
        sem_wait(&binaySemphore);
        /* down semaphore */
        printf("Semaphore is with %s\n",ucThreadBuff);
        sleep(1);
        sem_post(&binaySemphore);
        /* up semaphore */
        sleep(1);
    }
    pthread_exit(0); /* exit thread */
}

void* thread_2_function(void *ptr) {
    unsigned char* ucBuffPtr, ucThreadBuff[10];
    ucBuffPtr = (unsigned char *) ptr;
    strcpy(ucThreadBuff, ucBuffPtr);
    printf("thread_2_function entered\n");

    while(1) {
        sem_wait(&binaySemphore);
```

```
        /* down semaphore */
        printf("Semaphore is with %s\n", ucThreadBuff);
        sleep(1);
        sem_post(&binaySemphore);
        /* up semaphore */
        sleep(1);
    }
    pthread_exit(0); /* exit thread */
}
```

Inicializar un mutex

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);
```

- Compilar y enlazar con `-pthread`
- La función `pthread_mutex_init()` inicializa el mutex referenciado por `mutex` con atributos especificados por `attr`
- Si `attr` es `NULL`, se utilizan los atributos mutex predeterminados; el efecto es el mismo que pasar la dirección de un objeto de atributos mutex predeterminado
- Tras la inicialización exitosa, el estado del mutex se inicializa y desbloquea

Destruir un mutex

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Compilar y enlazar con -pthread
- La función `pthread_mutex_destroy()` destruye el objeto mutex referenciado por `mutex`; el objeto `mutex` se vuelve, en efecto, no inicializado
- Una implementación puede hacer que `pthread_mutex_destroy()` establezca el objeto referenciado por `mutex` en un valor no válido
- Un objeto mutex destruido se puede reinicializar usando `pthread_mutex_init()`; los resultados de hacer referencia al objeto después de su destrucción no están definidos
- Es seguro destruir un mutex inicializado que está desbloqueado. Intentar destruir un mutex bloqueado da como resultado un comportamiento indefinido.

Bloquear un mutex

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- Compilar y enlazar con -pthread
- El objeto mutex al que hace referencia `mutex` se bloquea llamando `pthread_mutex_lock()`
- Si el mutex ya está bloqueado, la llamada bloquea hasta que el mutex esté disponible

Desbloquear un mutex

```c
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Compilar y enlazar con `-pthread`
- La función `pthread_mutex_unlock()` libera el objeto mutex referenciado por `mutex`
- La forma en que se libera un mutex es depende del atributo de tipo del `mutex`
- Si hay subprocesos bloqueados en el objeto mutex al que hace referencia `mutex` cuando se llama a `pthread_mutex_unlock()`, lo que hace que el mutex esté disponible

mutex.c

```c
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t lock;
int shared_data;

void *thread_function(void *arg) {
    int i;
    for (i = 0; i < 10000; i++) {
        pthread_mutex_lock(&lock);
        shared_data++;
        pthread_mutex_unlock(&lock);
        usleep(1000);
    }
    return NULL;
}
```

mutex.c

```c
int main(void) {
    pthread_t thread_ID;
    void *exit_status;
    int i;

    pthread_mutex_init(&lock,NULL);
    pthread_create(&thread_ID, NULL, thread_function, NULL);
    sleep(1);

    for(i = 0; i < 10; i++) {
        pthread_mutex_lock(&lock);
        printf("\rShared integer's value = %d before\n", shared_data);
        sleep(1);
        printf("\rShared integer's value = %d after\n", shared_data);
        pthread_mutex_unlock(&lock);
        sleep(1);
    }
    printf("\n");
    pthread_join(thread_ID, &exit_status);
    pthread_mutex_destroy(&lock);
    return 0;
}
```