Procesos ligeros

Arquitectura de Computadores II
Universidad Carlos III de Madrid

Introducción

- Programa
  - Archivo ejecutable residente en un dispositivo de almacenamiento permanente
  - Se ejecuta por medio de la llamada **`exec()`**
- Proceso
  - Es un programa en ejecución
  - Los procesos se crean con la llamada **`fork()`**
- Servicios del sistema operativo
  - Invocados por medio de funciones
  - POSIX no diferencia entre llamadas al sistema y procedimientos de biblioteca

Definición

- Definición de proceso
  - Un proceso es un programa en ejecución, que se ejecuta **secuencialmente** (no más de una instrucción a la vez)
  - El proceso es una abstracción creada por el SO, que se compone de:
    - **Identificación del proceso.**
    - **Identificación del proceso padre.**
    - **Información sobre el usuario y grupo.**
    - **Estado del procesador.**
    - **Información de control de proceso.**
      - **Información del planificador.**
      - **Segmentos de memoria asignados.**
      - **Recursos asignados.**

Creación de procesos: `fork()`

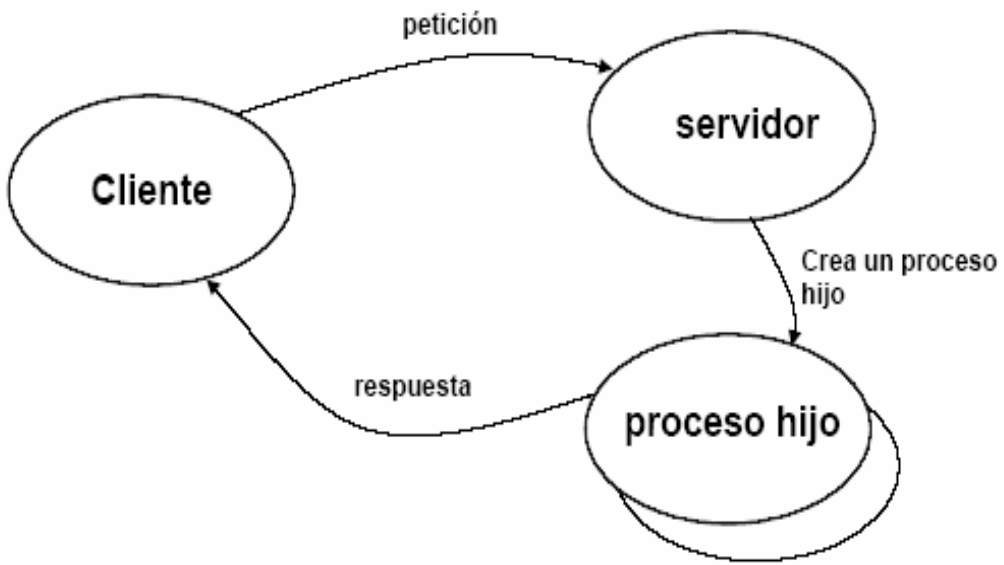
- La llamada **`fork()`** crea una copia (hijo) del proceso que la invoca
- El hijo hereda del padre:
  - Estado
  - Semáforos
  - Objetos de memoria
  - Política de planificación, etc.
- El hijo no hereda:
  - El PID
  - Alarmas y temporizadores
  - Operaciones de E/S asíncronas

Ejemplo de Proceso Pesado

```
#include <sys/types.h>
int main(void) {
pid_t id;
    id = fork();
    if (id == -1) {
        perror (“Error en el fork”);
        exit (1);
    }
    if (id == 0) {
        while (1) printf (“Hola: soy el hijo\n”);
    } else {
        while (1) printf (“Hola: soy el padre\n”);
    }
} /* Fin de main */
```

Ejemplo de procesos pesados (*fork*)

- Problemas:
  - Copia de todos los recursos del proceso.
  - Coste de comunicacion entre proceso

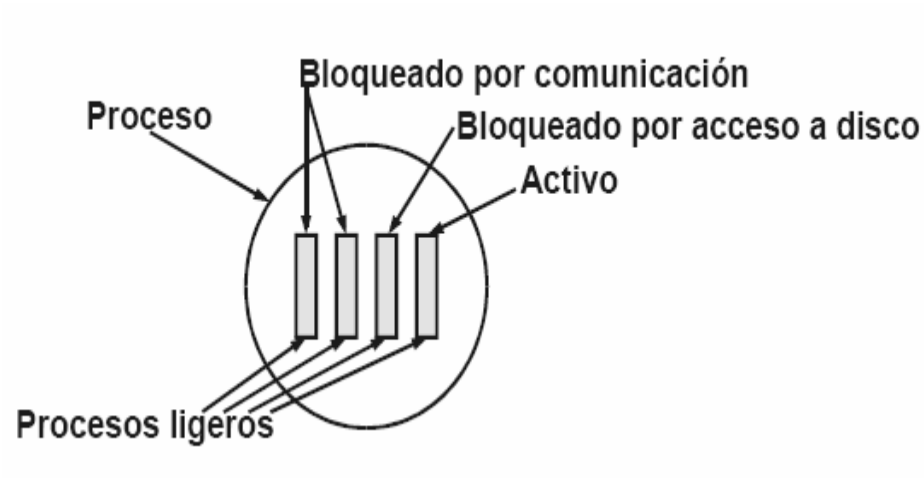


Qué es un proceso ligero

- Un *thread* es un flujo de control perteneciente a un proceso (a veces se habla de tareas con *threads*)
- Se les suele denominar también procesos ligeros, hebras, hilos, etc.
- La sobrecarga debida a su creación y comunicación es menor que en los procesos pesados
- Cada hilo pertenece a un proceso pesado
- Todos los hilos comparten su espacio de direccionamiento
- Cada hilo dispone de su propia política de planificación, pila y contador de programa

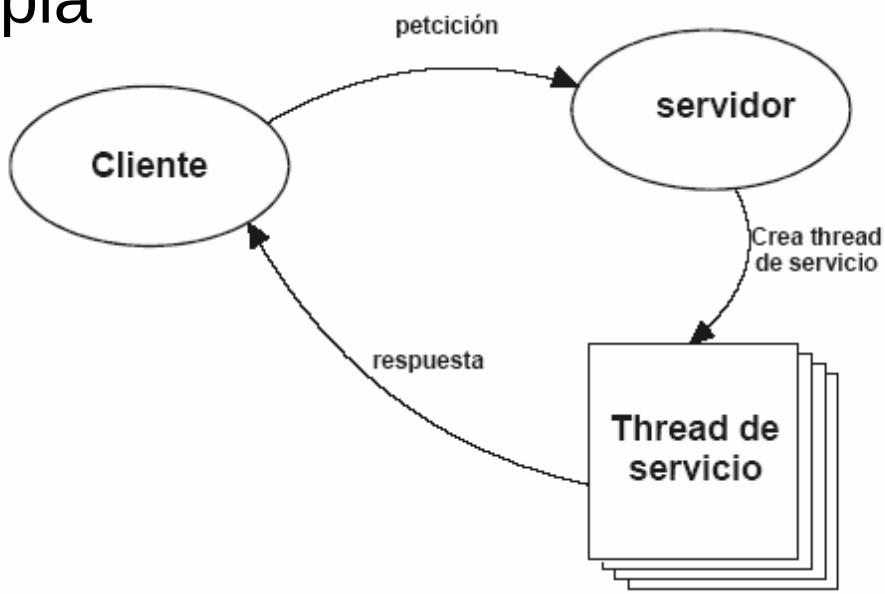
Procesos Ligeros

- Por proceso ligero
  - Contador de programa, Registros
  - Pila
  - Estado (ejecutando, listo o bloqueado)
- Por proceso
  - Espacio de memoria
  - Variables globales
  - Ficheros abiertos
  - Procesos hijos
  - Temporizadores
  - Señales y semáforos
  - Contabilidad



Ejemplo de procesos ligeros

- Soluciona:
  - Solo hay una copia de las variables compartidas.
  - Sin coste de comunicaciones entre procesos.



Generalidades

- Beneficios:
  - Explotación del paralelismo
  - Explotación de concurrencia (I/O)
  - Estilo de programación
- Inconvenientes:
  - Overhead por creación de threads
  - Sincronización: más bloqueos al haber más threads
  - Colisiones en el acceso a memoria
  - Más difícil la depuración: debuggers, trazadores, puntos de ruptura, repreducción de la ejecución, ...
- Uso de programación con threads:
  - Computación intensiva
  - Varios procesadores
  - Solapamiento computación y I/O
  - Aplicaciones servidor distribuidas

Operaciones con procesos ligeros

- Creación y destrucción
- Sincronización
- Gestión de prioridades
- Gestión de señales
- Gestión de memoria
- Se pueden utilizar todas las funciones incluidas en POSIX.1 y POSIX.1b
- La interfaz de hilos POSIX es pthreads, aunque existen otras bibliotecas de hilos

Operaciones

- Crear un nuevo hilo:

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

  - En **`thread`** devuelve el identificador de hilo
  - **`attr`** es un puntero a los atributos de la tarea
  - El tercer argumento en un puntero a la función que ejecutará el hilo
  - **`arg`** es un puntero a los argumentos del hilo

- Finalizar un hilo:

```
void pthread_exit(void *retval);
```

Ejemplo

```
#include <pthread.h>
void *Hilo (void *arg) {
   printf ("%s\n", (char *)arg);
   pthread_exit (0);
} /* Fin de Hilo */

main()
{
pthread_t th1, th2;
    pthread_create (&th1, NULL, Hilo, "Hilo 1");
    pthread_create (&th2, NULL, Hilo, "Hilo 2");
    sleep(5);
    puts ("Adios: Hilo principal");
} /* Fin de main */
```

Compilación

- Para compilar se usa la biblioteca *thread*:
  - *gcc –o ejemplo ejemplo.c -lpthread*

Atributos

- Los atributos definibles son:
  - Tamaño de la pila
  - Dirección de la pila
  - Control de devolución de recursos
- Los atributos se crean con:

```
int pthread_attr_init(pthread_attr_t *attr);
```

- Los atributos se destruyen con:

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

Dependencia de procesos

- Los hilos pueden operar en dos modos diferentes para controlar la devolución de recursos:
  - *Detached*: opera de modo autónomo, cuando termina devuelve sus recursos (identificador, pila, etc.)
  - *Joinable*: en su terminación mantiene sus recursos hasta que otro hilo invoca a **`pthread_join()`**

Dependencia de procesos

- Los recursos de una tarea *joinable* se liberan cuando esperamos con **`pthread_join()`**

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

- Si la tarea opera en modo *detached*, el propio hilo al terminar libera sus recursos.
- Para convertir a un hilo en *detached*, si no se hizo en su inicio:

```
int pthread_detach(pthread_t thread);
```

Ejemplo

```
#include <pthread.h>
void *Hilo (void *arg) {
   printf (“%s\n”, (char *)arg);
   sleep(3);
   pthread_exit (NULL);
} /* Fin de Hilo */
main()
{
pthread_t th1, th2;
void *st1;
    pthread_create (&th1, NULL, Hilo, "Hilo 1");
    pthread_join (th1, (void **) &st1);
    printf (“Retorno del hilo: %d\n”, st1);
} /* Fin de main */
```

Identificación

- **`pthread_t pthread_self(void);`**
  - Devuelve el id del proceso ligero.
- **`int pthread_equal(pthread_t thread1,`**
  `pthread_t thread2);`
  - Devuelve 0 si ambos id son iguales y 1 en caso contrario.

Ámbito de los procesos

- Existen las mismas políticas que para los procesos
- Los atributos de planificación se pueden especificar al crear el hilo en el objeto de atributos
- Se puede seleccionar entre dos ámbitos de planificación:
  - Ámbito de proceso: un planificador de segundo nivel planifica los hilos de cada proceso PTHREAD_SCOPE_PROCESS
  - Ámbito de sistema: los hilos se planifican como los procesos pesados PTHREAD_SCOPE_SYSTEM

Sincronización

- Se puede sincronizar asegurando el acceso en exclusión mutua a variables. Se usan mutex.

- O usando variables condición, que comunican información sobre el estado de datos compartidos

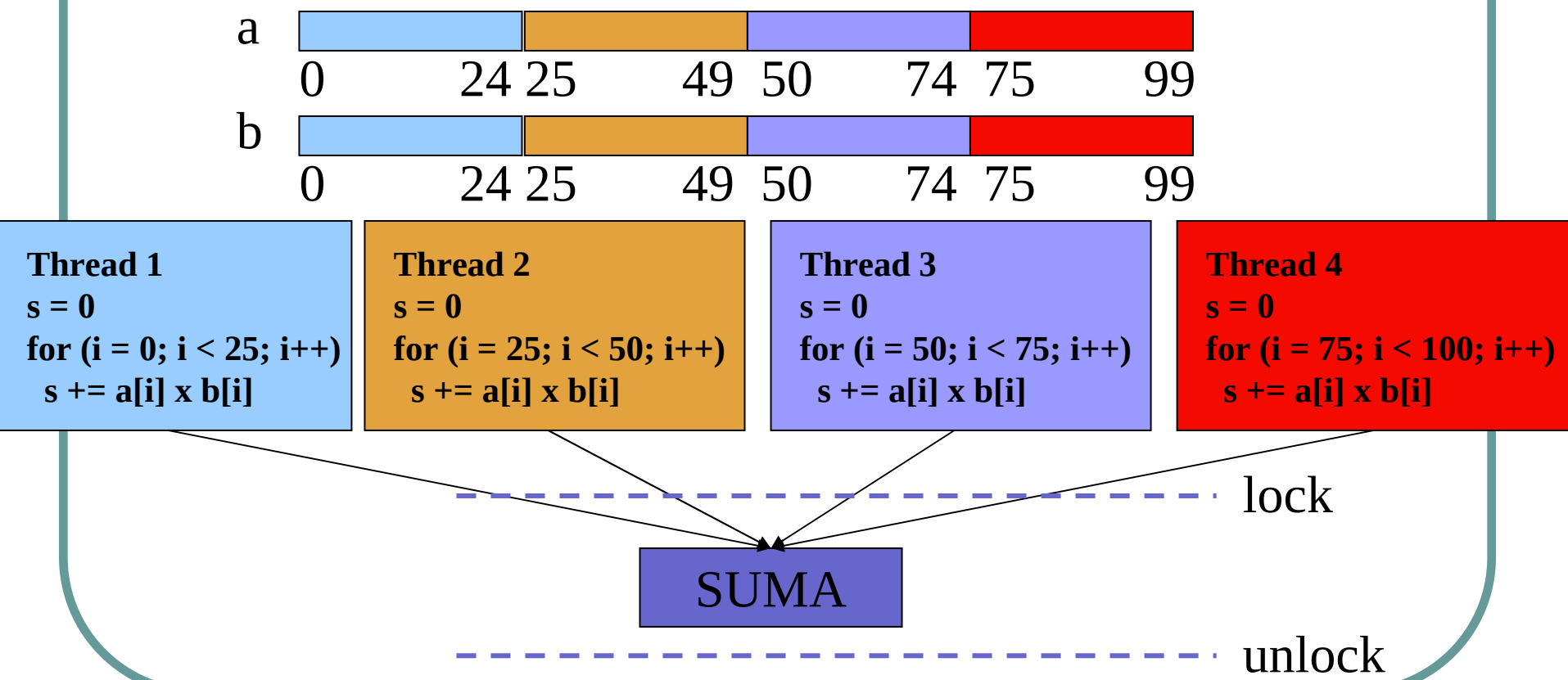
Mutex

- Un mutex es un mecanismo de sincronización indicado para procesos ligeros.
- Es un semáforo binario con dos operaciones atómicas:
  - *lock(m)*: Intenta bloquear el mutex; si el mutex ya está bloqueado el proceso se suspende.
  - *unlock(m)*: Desbloquea el mutex; si existen procesos bloqueados en el mutex se desbloquea a uno.

Manejo de Mutex

- **int pthread_mutex_init(pthread_mutex_t *mutex,**
  const pthread_mutexattr_t *attr);
- **int pthread_mutex_lock(pthread_mutex_t *mutex);**
  - Bloquea un mutex. No se puede bloquear un mutex cuando el thread ya lo tiene bloqueado. Esto devuelve un código de error, o se bloquea el thread.
- **int pthread_mutex_trylock(pthread_mutex_t *mutex);**
  - **Bloquea un mutex si no está bloqueado.**
  - **Si el mutex está bloqueado, el thread que llama *no* se bloquea y la función regresa el código de error: EBUSY**.
- **int pthread_mutex_unlock(pthread_mutex_t *mutex);**
  - Para desbloquear mutex. No se puede desbloquear un mutex desbloqueado, ni un mutex bloqueado por otro thread, pues los mutex pertenecen a los threads que los bloquean.
- **int pthread_mutex_destroy(pthread_mutex_t *mutex);**

Uso de Mutex
Ejemplo



Ejemplo

```
#include <pthread.h>
#include <stdio.h>
#include <malloc.h>
/*
    La siguiente estructura contiene la información necesaria para que “dotprod”
     obtenga datos y coloque el resultado
*/
typedef struct
 {
   double      *a;
   double      *b;
   double     sum;
   int          veclen;
 } DOTDATA;
```

Ejemplo

```
/*
    Variables globales accesibles y mutex
*/

#define NUMTHRDS 4
#define VECLEN 25

   DOTDATA dotstr;
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;
```

```
typedef struct
 {
   double      *a;
   double      *b;
   double     sum;
   int          veclen;
 } DOTDATA;
```

Ejemplo

```
void *dotprod(void *arg) {
int i, start, end, offset, len ;
double mysum, *x, *y;
offset    = (int)arg;
len       = dotstr.veclen;
start     = offset*len;
end       = start + len;
x = dotstr.a;   y = dotstr.b;
mysum = 0;
for (i=start; i < end ; i++)  {  mysum += (x[i] * y[i]); }
pthread_mutex_lock (&mutexsum);
dotstr.sum += mysum;
pthread_mutex_unlock (&mutexsum);
pthread_exit((void*) 0);
}
```

```
typedef struct
 {
   double      *a;
   double      *b;
   double     sum;
   int           veclen;
 } DOTDATA;
```

Ejemplo

```
int main (int argc, char *argv[]) {
   int i;
   double *a, *b;
   int status;
   pthread_attr_t attr;

   a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
   b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
   for (i=0; i < VECLEN*NUMTHRDS; i++)
    {
     a[i]=1.0;
     b[i]=a[i];
    }
```

Ejemplo

```
dotstr.veclen = VECLEN;
dotstr.a      = a;
dotstr.b      = b;
dotstr.sum    = 0;

pthread_mutex_init(&mutexsum, NULL);

for(i=0;i < NUMTHRDS;i++) {
      pthread_create( &callThd[i], NULL, dotprod, (void *)i);
    }

  for(i=0;i < NUMTHRDS;i++) {
      pthread_join( callThd[i], (void **)&status);
     }
```

Ejemplo

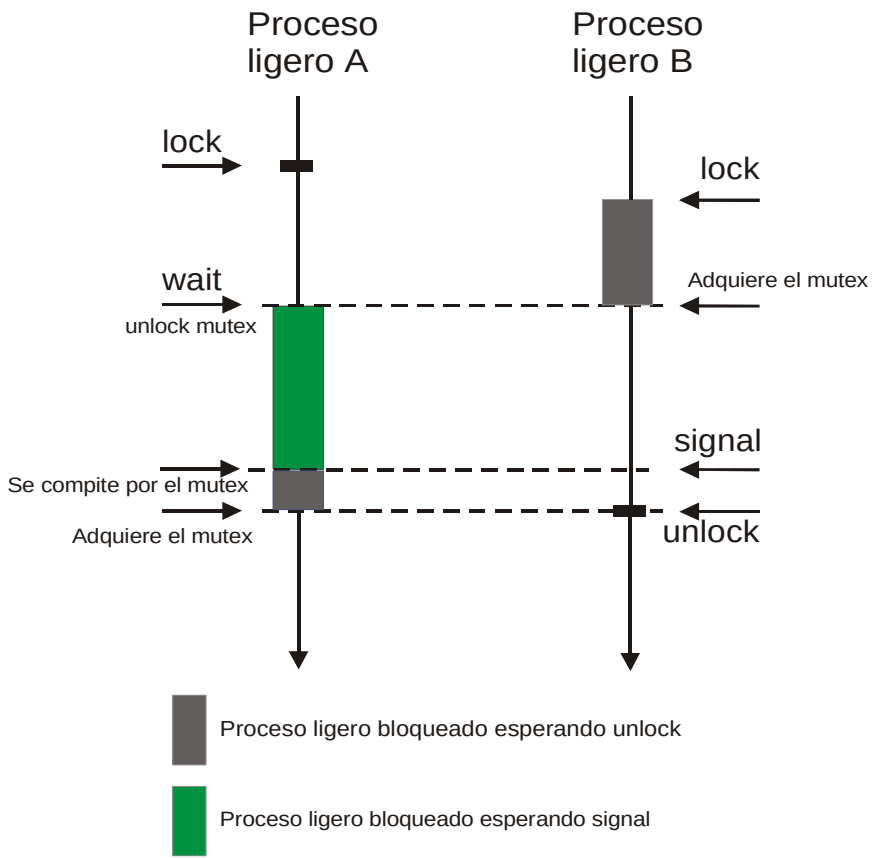
```
  /* Luego del join, imprima resultado */

    printf ("Sum =  %f \n", dotstr.sum);
    free (a);
    free (b);

    pthread_mutex_destroy(&mutexsum);
    pthread_exit(NULL);
}
```

Variables condición

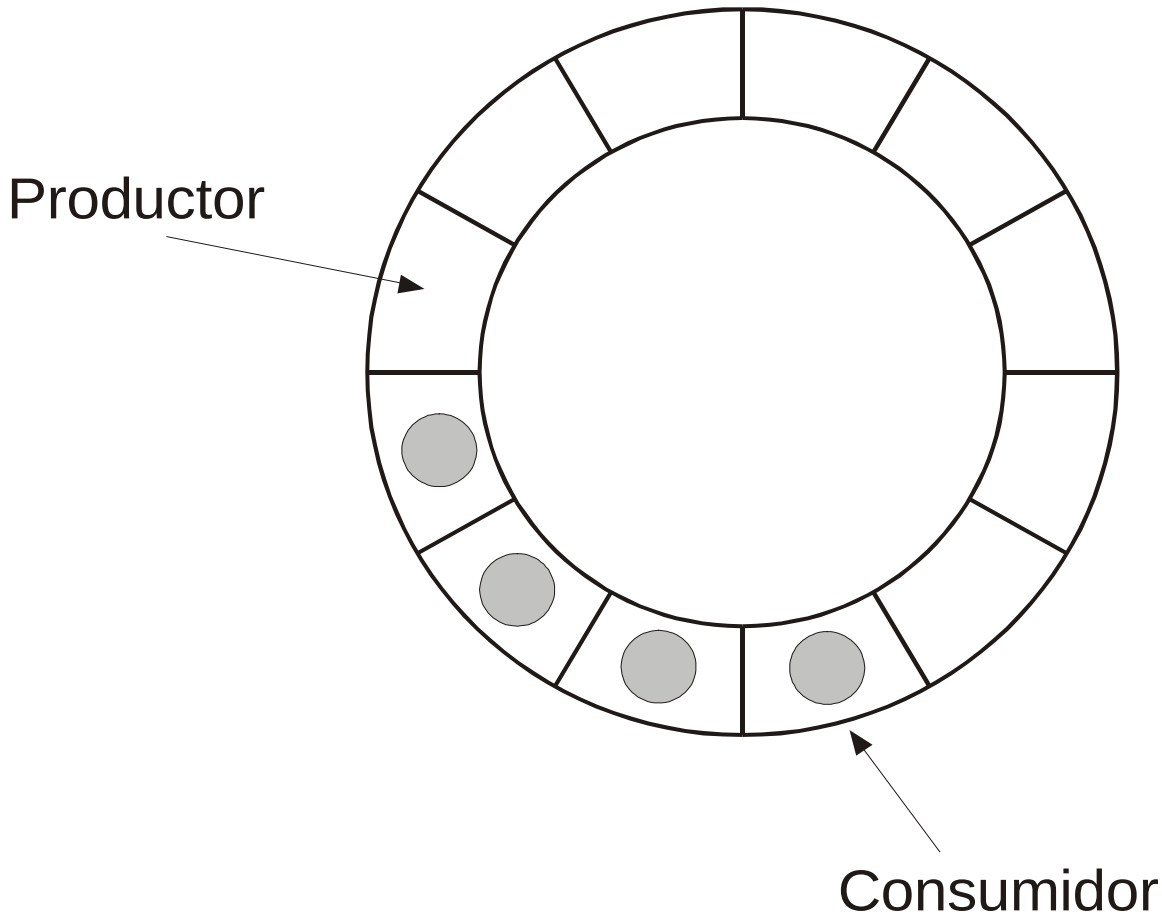
- Es un objeto de sincronización que permite bloquear a un hilo hasta que otro decide reactivarlo
- Operaciones:
  - Esperar una condición: un hilo se suspende hasta que otro señaliza la condición. En este punto se comprueba la condición y el proceso se repite si la condición es falsa
  - Señalizar una condición: se avisa a uno o más hilos suspendidos
  - *broadcast*: se reactivan todos los hilos suspendidos en la condición

Uso de variables condición y mutex

- Variables de sincronización asociadas a un mutex
- Conveniente ejecutarlas entre *lock* y *unlock*.
- Dos operaciones atómicas:
  - wait: Bloquea al proceso ligero que la ejecuta y le expulsa del mutex
  - *signal*: Desbloquea a uno o varios procesos suspendidos en la variable condicional. El proceso que se despierta compite de nuevo por el mutex



Ejemplo productor-consumidor



Codificación del ejemplo (I)

```
#define MAX_BUFFER       1024   /* tamaño del buffer */
#define DATOS_A_PRODUCIR 100000 /* datos a producir */
pthread_mutex_t mutex;     /* mutex para controlar el
                              acceso al buffer compartido
   */
pthread_cond_t no_lleno;   /* llenado del buffer */
pthread_cond_t no_vacio;   /* vaciado del buffer */
int n_elementos;           /* elementos en el buffer */
int buffer[MAX_BUFFER];    /* buffer común */

main(int argc, char *argv[])
{
   pthread_t th1, th2;

   pthread_mutex_init(&mutex, NULL);
```

Codificación del ejemplo (II)

```
    pthread_cond_init(&no_lleno, NULL);
    pthread_cond_init(&no_vacio, NULL);

    pthread_create(&th1, NULL, Productor, NULL);
    pthread_create(&th2, NULL, Consumidor, NULL);

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&no_lleno);
    pthread_cond_destroy(&no_vacio);

    exit(0);
}
```

Codificación del ejemplo (III)

```
void Productor(void) {  /* código del productor */
    int dato, i ,pos = 0;
    for(i=0; i < DATOS_A_PRODUCIR; i++ ) {
        dato = i;       /* producir dato */
        pthread_mutex_lock(&mutex);  /* acceder al buffer   */
        while (n_elementos==MAX_BUFFER)/* si buffer lleno   */
           pthread_cond_wait(&no_lleno, &mutex); /* bloqueo */
        buffer[pos] = dato;
        pos = (pos + 1) % MAX_BUFFER;
        n_elementos ++;
        pthread_cond_signal(&no_vacio); /* buffer no vacío  */
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(0);
}
```

Codificación del ejemplo (IV)

```
void Consumidor(void) { /* código del consumidor */
    int dato, i ,pos = 0;
    for(i=0; i < DATOS_A_PRODUCIR; i++ ) {
        pthread_mutex_lock(&mutex); /* acceder al buffer    */
        while (n_elementos == 0)    /* si buffer vacío      */
           pthread_cond_wait(&no_vacio, &mutex); /* bloqueo */
        dato = buffer[pos];
        pos = (pos + 1) % MAX_BUFFER;
        n_elementos --;
        pthread_cond_signal(&no_lleno); /* buffer no lleno  */
        pthread_mutex_unlock(&mutex);
        printf("Consume %d \n", dato);  /* consume dato     */
    }
    pthread_exit(0);
}
```

Otro ejemplo (I)

```
int dato = 5;                    /* recurso */
int n_lectores = 0;              /* numero de lectores */
pthread_mutex_t mutex;           /* control del acceso a dato */
pthread_mutex_t mutex_lectores; /* control de n_lectores */
main(int argc, char *argv[])
{
   pthread_t th1, th2, th3, th4;
   pthread_mutex_init(&mutex, NULL);
   pthread_mutex_init(&mutex_lectores, NULL);
    pthread_create(&th1, NULL, Lector, NULL);
    pthread_create(&th2, NULL, Escritor, NULL);
    pthread_create(&th3, NULL, Lector, NULL);
    pthread_create(&th4, NULL, Escritor, NULL);

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    pthread_join(th3, NULL);
    pthread_join(th4, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_mutex_destroy(&mutex_lectores);

    exit(0);
}
```

Otro ejemplo (II)

```
void Lector(void) {  /* código del lector */
   pthread_mutex_lock(&mutex_lectores);
   n_lectores++;
    if (n_lectores == 1)
       pthread_mutex_lock(&mutex);
    pthread_mutex_unlock(&mutex_lectores);

    printf("%d\n", dato);  /* leer dato */

    pthread_mutex_lock(&mutex_lectores);
    n_lectores--;
    if (n_lectores == 0)
       pthread_mutex_unlock(&mutex);
    pthread_mutex_unlock(&mutex_lectores);
   pthread_exit(0);
}
```

Otro ejemplo (III)

```
void Escritor(void)    /* código del escritor */
{
   pthread_mutex_lock(&mutex);

   dato = dato + 2;    /* modificar el recurso */

   pthread_mutex_unlock(&mutex);

   pthread_exit(0);
}
```

Señales: Introducción

- Una señal es un mecanismo para avisar a los proceso de la llegada de un evento
  - División por cero
  - Desbordamiento
  - Expiración de alarmas
  - Llegada de un mensaje
  - etc.
- Cada señal es identificada por un número
- Cuando llega una señal, el proceso es interrumpido y se invoca a un manejador de señal

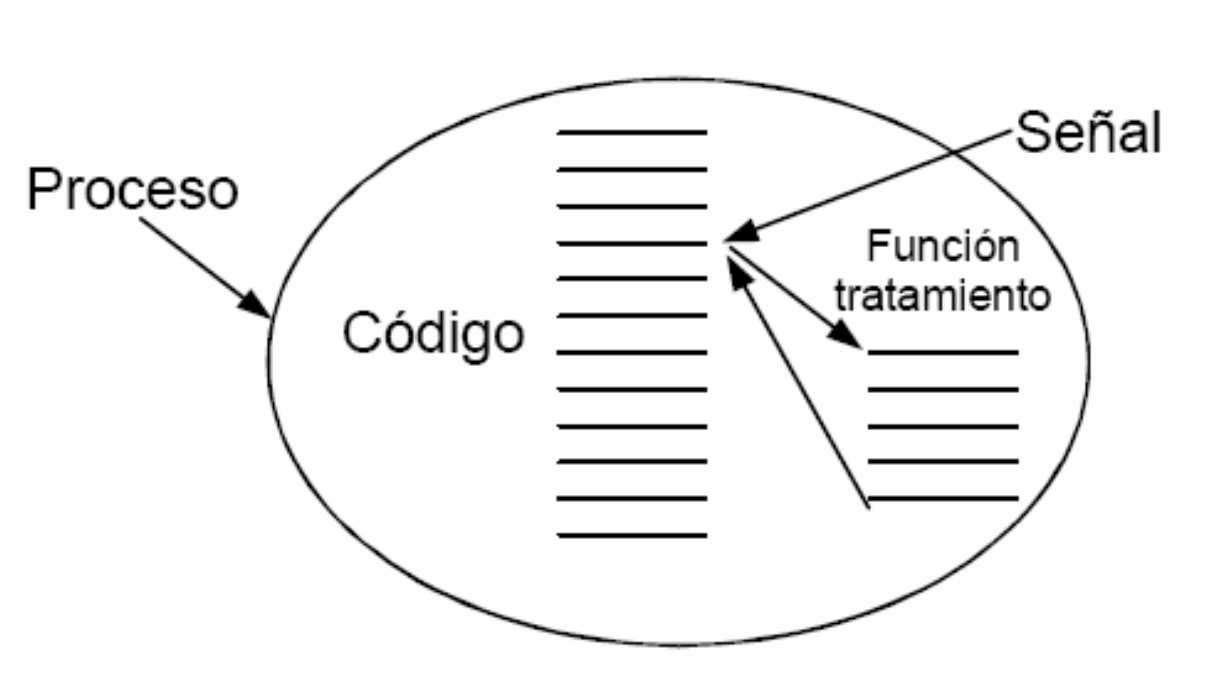
Tipos de señales

- Una señal viene dada por un número entero.
- Cualquier señal tiene un nombre simbólico que comienza con SIG.
- Todas las señales se encuentran definidas en signal.h
- Algunos ejemplos:
  - SIGALRM expiración del temporizador
  - SIGFPE error de operación aritmética
  - SIGILL ejecución de instrucción ilegal
  - SIGINT enviada al pulsar Ctrl-C
  - SIGKILL terminación
  - SIGSEGV referencia a memoria inválida
  - SIGUSR1 señal definida por el usuario
  - SIGUSR2 señal definida por el usuario

Acciones asociadas a funciones

- Un proceso puede:
  - ***Bloquear*** una señal y manejarla más tarde.
  - ***Manejar*** la señal, ejecutando una función cuando se genera la señal.
  - ***Ignorar*** a la señal cuando se genera (la señal se pierde).
- Una señal no bloqueada y no ignorada se ***entrega*** al proceso tan pronto como se ***genera***.
- Una señal bloqueada no se entrega, queda pendiente de entrega.
- La ***acción por defecto*** cuando llega una señal consiste en abortar la ejecución del mismo (el proceso muere).

Manejadores



Manejo de señales

```
#include <signal.h>
int sigaction( int signo, struct sigatcion *act,
                        struct sigaction *oact);
struct sigaction {
            void (*sa_handler)(int sig);/* manejador */
            sigset_t sa_mask; /* señales bloqueadas durante la ejecución
                                    del manejador */
            int sa_flags; /* posibles opciones */
};
```

- El manejador (sa_handler) puede ser:
  - SIG_DFL acción por defecto
  - SIG_IGN ignorar la señal, no hacer nada
  - Un puntero a una función que será invocada cuando se genere la señal.

Manejadores

- Fragmento de código que ignora la señal SIGINT.

```
#include <signal.h>
struct sigaction act;
act.sa_handler = SIG_IGN;
sigemptyset(&act.sa_mask);
act.sa_flags = 0;
sigaction(SIGINT, &act, NULL)
```

Ejemplo

```
#include <singal.h>
#include <stdio.h>
int total = 0;
void contar(void) {
    total++;
    printf("Ctlr-C = %d\n", total);
}

void main void() {
    struct sigaction act;
    act.sa_handler = contar;
    sigemptyset(&act.sa_mask); act.sa_flags = 0;
    sigaction(SIGINT, &act, NULL);
    for(;;) ;
}
```

Señales de reloj

#include <unistd.h>

unsigned int alarm(unsigned int seconds)

- Programa la señal SIGALRM para que ocurra al cabo de un cierto número de segundos (seconds).
- Un proceso solo puede tener una petición de alarma pendiente. Cada nueva alarma anula la anterior.
- alarm(0) cancela todas las alarmas pendientes
- La función devuelve el tiempo restante para el vencimiento de la alarma.

Ejemplo

```
#include <signal.h>
#include <stdio.h>
void tratar_alarma(void) {
     printf("Activada \n");
}
main() {
        struct sigaction act;
        /* establece el manejador para SIGALRM */
        act.sa_handler = tratar_alarma; /* función a ejecutar */
        act.sa_flags = 0; /* ninguna acción especifica */
        sigemptyset(&act.sa_mask);
        sigaction(SIGALRM, &act, NULL);
        /* recibe SIGALRM cada 3 segundos */
        for(;;) {
             alarm(3);
             pause();
      }
}
```

Referencias

- http://dis.cs.umass.edu/~wagner/threads_html/tutorial.html
- http://arcos.inf.uc3m.es/~ssdd/tema3.pdf
- http://arcos.inf.uc3m.es/~stf/transparencias/tema2.pdf
- http://atc1.aut.uah.es/~arqui/Trps/POSIX_de_tiempo_real.ppt
- http://it.aut.uah.es/amoreno/SCTR/Comunicacion%20y%20sincronizacion.pdf
- http://laurel.datsi.fi.upm.es/~ssoo/STR/CS_POSIX_SSOO2.ppt