Trabajo con Hilos en C - Posix

Antonio Espín Herranz

HILOS: Conceptos

- **Proceso**: podemos considerar un proceso como un programa en ejecución.
  - Un mismo programa puede derivar en varios procesos.
    - Tener varios ejemplares en ejecución de un programa, por ejemplo, Word o cualquier otro.
  - Y también un mismo programa puede necesitar utilizar diferentes procesos:
    - Por ejemplo, Word puede requerir utilizar un editor de ecuaciones, o de gráficos.

Procesos

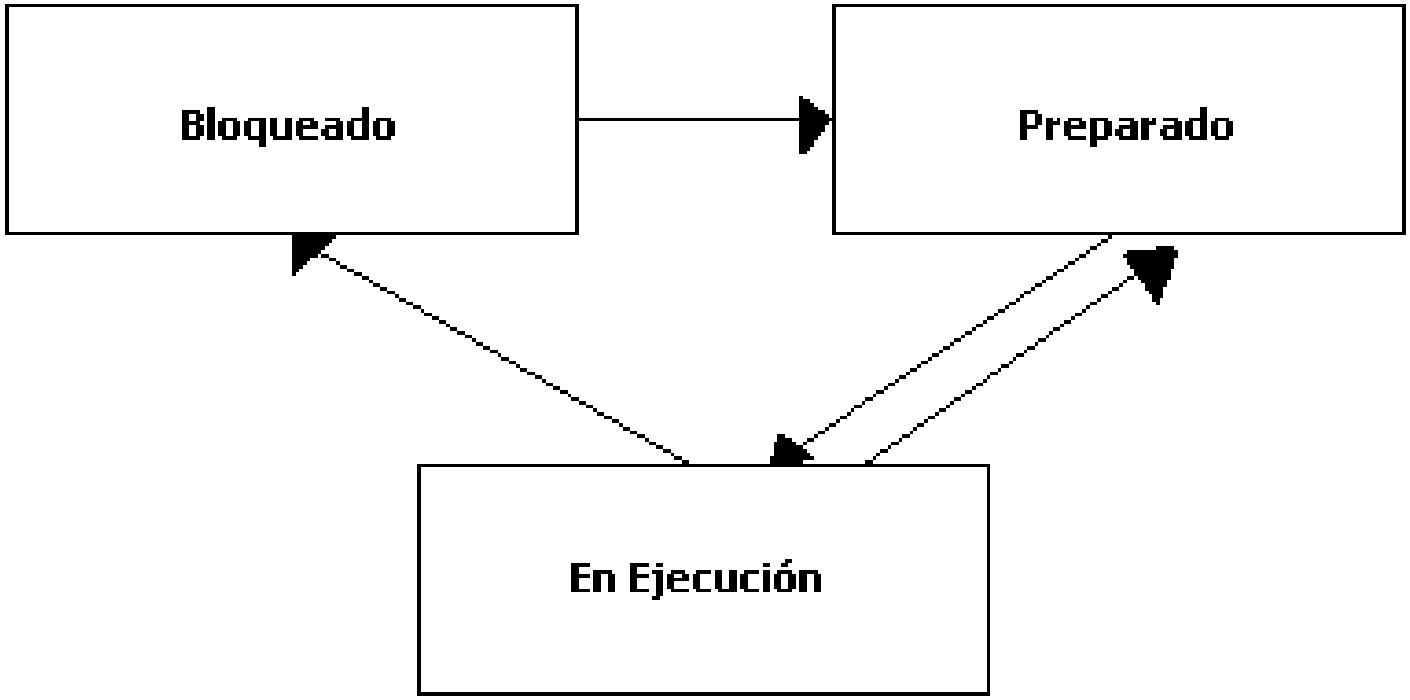
- Cada proceso consta de bloques de código y de datos cargados desde un fichero ejecutable.
- Además, un proceso posee:
  - Su propio espacio de direcciones.
  - Sus variables.
  - Ficheros abiertos.
  - Referencias a procesos hijos.
  - Registros, pila, etc.

- El equivalente a una CPU virtual.
- Los procesos se controlan por el Sist. Oper. Y se van alternando en la utilización de la CPU.

Estados de un proceso

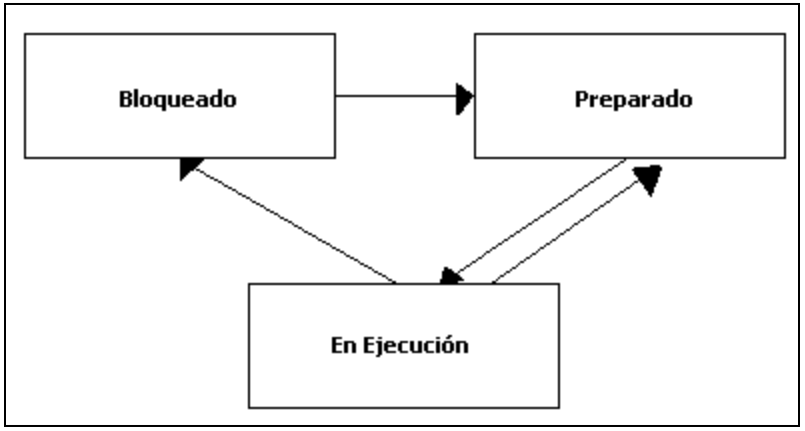
- Posibles estados de un proceso:
  - **Ejecución**: está utilizando CPU.
  - **Preparado**: está detenido temporalmente para que se ejecute otro proceso.
  - **Bloqueado**: El proceso está esperando a que ocurra algo para continuar. Puede estar esperando leer datos de la entrada estándar.

Ciclo de vida del proceso



Hilos

- Un hilo (*thread*), proceso ligero.
- Es la unidad de ejecución de un proceso.
- Dispone de:
  - Secuencia de instrucciones.
  - De un conjunto de registros.
  - Y una pila.
- Cuando se crea un proceso, el Sist. Ope. Crea un primer hilo (hilo primario), que a su vez puede crear hilos adicionales.
- Un proceso puede ejecutar de forma concurrente varias secuencias de instrucciones.
- Cada secuencia de instrucciones recibe el nombre de hilo (thread), todos los hilos comparten el espacio de memoria virtual del proceso y los recursos asignados por el sistema operativo.



Estados de un hilo

- **Nuevo**: El hilo se ha creado, pero no se ha activado. De aquí pasará a preparado.
- **Preparado**: Está activo y a la espera de que se le asigne la CPU.
- **Ejecución**: Está activo y se le ha asignado la CPU.
- **Bloqueado**: El hilo espera a que termine otro hilo. Puede estar:
  - **Dormido**: Se bloquea durante x tiempo, de aquí pasará al estado de preparado.
  - **Esperando**: El hilo está esperando a que ocurra alguna cosa: espera una operación de E/S o adquirir una propiedad de un método sincronizado.
- **Muerto**: El hilo ha terminado y no ha sido recogido por el padre. De aquí ya no puede ir a ningún estado.

Biblioteca POSIX

- Posix: *Portable Operating System Interface*.
- Esta biblioteca se encuentra definida en el fichero <pthread.h>
- Funciones:
  - pthread_create: Crear hilos.
  - pthread_once: Ejecutar el código una vez.
  - pthread_join: Esperar a otro hilo.
  - pthread_equal: Comparar hilos.
  - pthead_detach: Alternativa a pthread_join.
  - pthead_self:Devuelve el identificador del hilo que invoca a esta función.

Compilación con gcc

- gcc fichero.c –o fichero **–pthread**

- Para simular retardos dentro de un hilo:
- Utilizar la función **sleep(segundos**).
- Se encuentra en el fichero: <**unistd.h**>

pthread_create

- Recibe 4 parámetros:

pthread_t *tid:

El identificador del hilo.

const pthread_attr_t *attr:

Se suele dejar a 0 para que utilice los parámetros por defecto de tamaño de pila.

void *(*start)(void *):

Puntero a una función. La que va a ejecutar el hilo. A esta función SOLO se le puede pasar un dato y devuelve un dato, en caso de necesitar mandar mas información, utilizar estructuras.

void *arg:

Es la dirección del argumento que se le pasará a la función start

Devuelve 0 si todo OK, si no un código de error.

pthread_once

- Recibe dos parámetros:

  pthread_once_t *once_control:

  – Nos asegura que el código que indiquemos en la rutina que vamos a pasar sólo se ejecutará una vez. El primer parámetro apunta a una variable static o extern iniciada a la constante: PTHREAD_ONCE_INIT.

  void (*init_routine)(void):

  – La dirección de la rutina a ejecutar.

  Devuelve 0 si todo OK, si no un código de error.

pthread_join

- Recibe dos parámetros:

  pthread_t thread: Un hilo.

  – Esta función suspende la ejecución del hilo que la invoca, hasta que el hilo indicado por thread no termine su ejecución.

  void **value_ptr

  – El segundo parámetro cuando NO es cero, hace referencia al valor devuelto por el hilo identificado por thread (valor devuelto por return). Podemos poner 0.

- Esta función devuelve 0, si todo OK, sino != 0.
- La función main se encarga de crear los hilos y esperar a que terminen para ello utiliza la función **pthread_join**.

pthread_equal

- Recibe los dos identificadores de los hilos.
- De tipo pthread.
- Devuelve 0 si son distintos y si no devuelve != 0.

pthread_detach

- Esta función es una alternativa de join.
- Recibe el hilo: parámetro de tipo pthread_t.

pthread_self

- Devuelve el identificador del hilo que la invoca.

- No tiene parámetros.

Creación / Destrucción de Hilos

- Incluir el fichero de cabecera:

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

- Escribir las instrucciones del hilo dentro de una función, con el prototipo:

```
void *fnHilo(void *p); // Al ser punteros genéricos podemos pasar y
                       //  devolver cualquier cosa.
```

- Definir el identificador del hilo(s):

```
pthread_t id_hilo;
```

- Crear el hilo e iniciar su ejecución:

```
pthread_create(&id_hilo, 0, fnHilo, 0);
```

- Desde el hilo primario esperamos a que termine el hilo secundario.

```
pthread_join(id_hilo, 0);
```

Ejemplo

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

int random(int n) {
    static int primera_vez = 1;
    if (primera_vez) {
        srand(time(0));
        primera_vez = 0;
    }
    return (rand() % n) + 1; // valor entre 1 y n
}
```

Ejemplo (2)

```
void *tomarMuestraTipoA(int *n) {
	*n = random(10);

	printf("Vamos a tomar %d muestras …\n" , *n );
	for (int i = 0; i < *n; i++) {
		printf ("Tomando muestra de tipo A\n“);
	}
	Sleep(2);
	return n;
}

void *tomarMuestraTipoB(int *n) {
	*n = random(15);

	 printf("Vamos a tomar %d muestras …\n" , *n );
	for (int i = 0; i < *n; i++) {
		printf(“Tomando muestra de tipo B\n“);
	}
	Sleep(5);
	return n;
}
```

FUNCIONES QUE SE VAN A EJECUTAR CON HILOS

TIENE QUE SER DEL PROTOTIPO

void *funcion(void *)

Punteros genéricos → conversiones implicitas.

Ejemplo (3)

```
void resultados(int n, int m) {
        printf("Muestras de tipo A: %d \n“, n);
        printf("Muestras de tipo B: %d \n”, m);
        printf("Total:  %d\n“,  n + m );
}

typedef void *(*ptrfnthr)(void *);
int main() {
        int nMuestrasTipoA = 0;
        int nMuestrasTipoB = 0;
        int codigo, *estado;

        // Identificadores de los hilos
        pthread_t hilo1, hilo2;
        // Crear los hilos 1 y 2

        printf( "Creamos el hilo 1\n”);
        pthread_create(&hilo1, 0, (ptrfnthr) tomarMuestraTipoA, (void *) &nMuestrasTipoA);

        printf("Creamos el hilo 2\n”);
        pthread_create(&hilo2, 0, (ptrfnthr) tomarMuestraTipoB, (void *) &nMuestrasTipoB);

        // Esperar a que los hilos terminen
        codigo = pthread_join(hilo1, (void **) &estado );
        codigo = pthread_join(hilo2, (void **) &estado);

        // Mostrar resultados
        resultados(nMuestrasTipoA, nMuestrasTipoB);

        }
```

Paso de parámetros

- Si necesitamos pasar mas de un parámetro a nuestro hilo tendremos que definir una estructura.

- Se definirá de la siguiente manera:

```
typedef struct {
    tipo var;
    tipo2 var2;
} pthread_param_t;
```

Finalizar Hilos

- Tenemos varias formas de terminar un hilo:
  - **Retornando** de la función de ejecución.
    - return …
  - Llamando él mismo a **pthread_exit**.
    - void pthread_exit(void *);
  - Invocando a la función **pthread_cancel** pasando por argumento su identificador.
    - pthread_cancel(pthread_t);
    - *Esta es delicada, podemos cancelar un hilo que está interactuando con un recurso y no lo ha liberado.*

Ejemplo

```
void *tomarMuestra(pthread_param_t *p) {
    p->cuenta = random(15);
    for (int i = 0; i < p->cuenta; i++)
        printf("%s\n", p->str);
    return &p->id;
}
```

```
void tomarMuestra(pthread_param_t *p) {
    p->cuenta = random(15);
    for (int i = 0; i < p->cuenta; i++)
        printf("%s\n", p->str);
    pthread_exit(&p→id);
}
```

Sincronización

- Habrá veces que nos interese trabajar con hilos independientes, es decir, no interfieren entre sí en la ejecución.

- Y otra veces tendremos un esquema de colaboración, pueden acceder a una zona de datos / recursos compartidos → **sincronización**.

- Las zonas comunes donde acceden los hilos reciben el nombre de **secciones críticas**.

Sincronización (2)

- Dentro de la librería de Posix tenemos una serie de elementos de sincronización como son:
  - Secciones mutuamente excluyentes (los **mutex** o **semáforos binarios**).
    - Los mutex se utilizan para el acceso mutuamente exclusivo a recursos compartidos. Similares a los semáforos, pero el hilo que bloquea el mutex (el propietario) debe ser el mismo que lo libere.

Secciones críticas

- Una sección crítica para nosotros va a ser una zona de memoria compartida por dos o mas hilos.

- Podemos tener un array de muestras para almacenar datos y varios hilos que acceden simultáneamente a dicho array.

- La estructura podría ser algo así:

```
struct {
    int muestras[MAX_MUESTRAS];
    int ind;
} st_bufer;
```

Secciones críticas (2)

- La función a ejecutar por los hilos sería algo así:

```
void *adquirirDatos(void *pid){
    int x = 0, id = *((int *)pid);
    do {
        if (st_bufer.ind >= MAX_MUESTRAS)
                return pid;
        x = random();
        printf( “hilo-%d tomo la muestra %d\n”, id,st_bufer.ind );
        st_bufer.ind++;
     } while (st_bufer.ind < MAX_MUESTRAS);
     return 0;
}
```

Secciones críticas (3)

- El código tal cúal está puede generar resultados inesperados, si lo ejecutamos con mas de un hilo se pueden duplicar valores o saltarse algún valor.
- Este problema se conoce con el nombre de condiciones de carrera.
- La sección de código en la que se actualizan datos comunes a mas de un hilo recibe el nombre de sección crítica.
- La forma de evitar problemas planteados es que cuando un hilo esté ejecutando código de su sección crítica, ningún otro hilo pueda hacerlo.
  - Este mecanismo se conoce como **Exclusión Mutua**.

Exclusión mutua

- La biblioteca Posix proporciona un objeto llamado **mutex** (exclusión mutua), recibe el nombre de **semáforo binario** o **cerrojo**.

- Con un mutex se pueden realizar dos tipos de operaciones:
  - Echar el cerrojo.
    - Sección crítica.
  - Quitar el cerrojo.

- Cuando un hilo adquiere el mutex tiene la propiedad de que cuando es adquirido por un hilo, ningún otro hilo puede adquirirlo hasta que sea liberado.

Mutex

- Los mutex se definen **a nivel global**:
  - pthread_mutex_t objeto_mutex;
  - Las funciones que ofrece la librería para trabajar con mutex devuelve un 0 si va todo bien y algo distinto de 0 si hay error.

Funciones para trabajar con mutex

- int pthread_mutex_init(pthead_mutex_t *mutex, const pthread_mutexattr_t *attr);
  - Es la primera que se invoca para invocar al mutex.
  - Primer parametro el mutex, y el segundo los parámetros del mutex, con 0 se utilizan los valores por defecto.

- int pthread_mutex_lock(pthead_mutex_t *mutex);
  - Cuando se ha iniciado el mutex, se puede adquirir con un hilo para echar el cerrojo a su sección crítica.

Funciones para trabajar con mutex (2)

- int pthread_mutex_unlock(pthread_mutex_t *mutex);
  - Un hilo puede quitar el cerrojo que puso a su sección crítica.

- int pthread_mutex_destroy(pthread_mutex_t *mutex);
  - Destruye el mutex especificado, liberando todos los recursos adquiridos por el mismo.

Codificación

- Dentro de la estructura, añadimos el mutex:

```
struct {
   int muestras[MAX_MUESTRAS];
   int ind;
   pthread_mutex_t mutex_adquirir;
} st_bufer;
```

Codificación (2)

- La función que ejecutan los hilos:

```
void *adquirirDatos(void *pid){
    int x = 0, id = *((int *)pid);
    do {
        cod = pthread_mutex_lock(&st_bufer.mutex_adquirir);
        if (st_bufer.ind >= MAX_MUESTRAS)
                return pid;
        x = random();
        printf("hilo-%d tomo la muestra %d \n",id,st_bufer.ind);
       st_bufer.ind++;
       cod = pthread_mutex_unlock(&st_bufer.mutex_adquirir);
      } while (st_bufer.ind < MAX_MUESTRAS);
     return 0;
}
```

- Desde main() creamos el mutex:
  - **cod = pthread_mutex_init(&st_bufer.mutex_adquirir, 0);**

Semáforos

- Un semáforo puede controlar varios recursos accedidos por distintos hilos simultáneamente.
- Incluir el fichero: ***semaphore.h***
- Definir un semáforo:
  - **sem_t** semaforo;
- Una vez definido el semáforo puede ser manejado a través de una serie de funciones:
  - sem_init

Funciones de los semáforos

- int sem_init(sem_t *sem, int pshared, unsigned int value);
  - Inicia el semáforo al valor value. Si pshared es 0 no se puede compartir con otros procesos.

- int sem_wait(sem_t *sem);
  - Espera a que el contador del semáforo sea positivo. Esto es, si es 0, el hilo que hace la llamada espera (pasa al estado de bloqueado) hasta que el contador sea incrementado por una operación sem_post, o hasta que el hilo sea interumpido, y si es positivo lo decrementa en uno y retorna.

Funciones de los semáforos (2)

- int sem_post(sem_t *sem);
  - Incrementa en 1 el contador del semáforo si no hay hilos bloqueados, si hubiera hilos esperando por el semáforo, el primero de ellos será despertado y pasa de bloqueado a preparado.

- int sem_destroy(sem_t *sem);
  - Elimina el semáforo referenciado por sem.

Problema del productor / consumidor

- Es un problema muy típico.
- La idea es que tenemos un hilo que produce (lo que sea) y otro que consume (lo que genera el otro).
- El productor no puede generar nuevos datos si el almacén está lleno.
- Y el consumidor no puede capturar datos si no hay.
- Tiene que establecerse una sincronización entre ambos hilos.

Esquema

- Necesitamos dos hilos: uno hará de consumidor y otro de productor.
- Dos semáforos, uno controla que esté lleno el buffer y el otro que esté vacío.
- A parte de un mutex para controlar el acceso exclusivo.

Esquema

```
const int ELEMENTOS_BUFER = 16;
const int TOTAL_DATOS = 512;

struct {
  int bufer[ELEMENTOS_BUFER]; // matriz de datos
  int ind_p, ind_c; // índices del productor y del consumidor
  pthread_mutex_t exmut;    // exclusión mutua
  sem_t sem_vacios; // semáforo contador de elementos vacíos
  sem_t sem_llenos; // semáforo contador de elementos llenos
} st_bufer;
```

- El semáforo vacíos se inicializa al número de elementos del buffer.
- El otro a cero, al inicio no hemos puesto ninguno y están todos libres.

Productor

- Espera al semáforo vacíos.
- Echar el cerrojo.
- Genera el carácter y lo pone en el buffer.
- Trabaja con el índice del productor.
- Quita el bloqueo del cerrojo.
- Incrementa el semáforo de llenos.

Consumidor

- Espera al semáforo de llenos.
- Echa el cerrojo.
- Recupera un dato del buffer.
- Trabaja con el índice del consumidor.
- Desbloquea el cerrojo.
- Incrementa el semáforo de vacíos.

PRACTICA
HILOS 2