



Programación en Hilos (Threads)

*Ing. Eddie Angel
Sobrado Malpartida
sobrado.ea@pucp.edu.pe*

Librería **pthread.h**

- **Pthreads** esta definida en ANSI/IEEE POSIX 1003.1
 - Las subrutinas el cual comprende los Pthreads se dividen:
 - a. **Administracion Thread** : trabaja directamente sobre la creacion, detaching, joining de hilos etc.
 - b. **Mutex**: Trata con un tipo de **sincronización**, llamado *mutex*, el cual es una abreviación de exclusión mutua. Estas funciones proveen creación, destrucción, locking y unlocking de mutexes.
- También son complementados por funciones de **atributo** de mutex que modifican los atributos asociado con los mutex.

Pthreads (Threads POSIX)

- c. **Variables de Condición:** trata con un tipo fino de **sincronización** basado sobre las **condiciones específicas** del programador.

Esta clase incluye funciones para *crear, destruir, suspender y señalar* en base a valores de variables especificadas.

Creación de hilos: *pthread_create()*

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(* start_routine)(void *),void *arg);
```

- Crea inmediatamente el hilo en estado **preparado**, por lo que el hilo creado y su hilo creador **compitan** por la CPU según la política de planificación del sistema
- Puede ser invocada por cualquier hilo del proceso (no sólo por el “hilo inicial”) para crear otro hilo
- **Parámetros:**
 - ✓ **attr** es el atributo que contiene las características del hilo creado (véanse atributos de un hilo en las siguientes transparencias)
 - ✓ **start_routine** es la función que ejecutará el hilo
 - ✓ **arg** es un puntero a los parámetros iniciales del hilo
 - ✓ En **thread** se devuelve el identificador del hilo creado si la llamada tiene éxito
- **Valor de retorno:**
 - ✓ 0 si éxito y un valor negativo si hay error

Terminación de hilos: *pthread_exit()*

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

```
void pthread_exit(void *status);
```

- **pthread_exit** finaliza explícitamente la ejecución del hilo que la invoca. La finalización de un hilo también se hace cuando finaliza la ejecución de las instrucciones de su función
- La finalización del último hilo de un proceso finaliza la ejecución del proceso
- Si el hilo es sincronizable (joinable) el identificador del hilo y su valor de retorno puede examinarse por otro hilo mediante la invocación a **pthread_join** a través del parámetro **status**

Identificación de Hilo: *pthread_self()*

```
#include <pthread.h>  
  
pthread_t pthread_self(void);
```

- La función **pthread_self** devuelve el identificador de hilo (tid, thread identifier) del hilo que la invoca

Programa 1

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    printf("\n%d: Hola !\n", threadid);
    pthread_exit(NULL);
}
```

```
int main()
{
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0;t < NUM_THREADS;t++){
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL,
                           PrintHello, (void *)t);
        if (rc){ printf("ERROR ");
                exit(-1); }
    }
    pthread_exit(NULL);}
```

Programa 1

```
Creating thread 0  
Creating thread 1  
0: Hola!  
1: Hola!  
Creating thread 2  
Creating thread 3  
2: Hola!  
3: Hola!  
Creating thread 4  
4: Hola!
```

*Observe
como se
muestra la
conurrencia*



Programa 2

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

```
void * periodic (void *arg)
```

```
{
```

```
int period;
```

```
period = *((int *)arg);
```

```
while (1) {
```

```
printf("En tarea con periodo %d\n", period);
```

```
sleep (period);
```

```
}
```

```
}
```

```
main() {
```

```
pthread_t th1, th2;
```

```
int period1, period2;
```

```
period1 = 2;
```

```
period2 = 3;
```

```
if (pthread_create(&th1, NULL,  
periodic, &period1)==-1)
```

```
{ perror(""); }
```

```
if (pthread_create(&th2, NULL,  
periodic, &period2)==-1)
```

```
{ perror(""); }
```

```
sleep(30);
```

```
printf("Salida del hilo  
principal\n");
```

```
exit(0);
```

```
}
```

Atributos de hilos

- `int pthread_attr_init(pthread_attr_t *attr);`
Inicializa un atributo de thread a su valor por defecto
- `int pthread_attr_destroy(pthread_attr_t *attr);`
Destruye un atributo de thread
- `int pthread_attr_getstacksize(const pthread_attr_t *attr, size_t *stacksize);`
Devuelve el tamaño de pila de un atributo de thread
- `int pthread_attr_setstacksize(pthread_attr_t *attr, size_t *stacksize);`
Establece el tamaño de pila de un atributo de thread

Atributos de los Hilos

- `int pthread_attr_getstackaddr(pthread_attr_t *attr, void *stackaddr);`
Devuelve la dirección de la pila de un atributo de thread
- `int pthread_attr_setstackaddr(pthread_attr_t *attr, void stackaddr);`
Establece la dirección de la pila de un atributo de thread
- `int pthread_attr_getdetachstate(const pthread_attr_t *attr, void *detachstate);`
Devuelve el estado *detach* de un atributo de thread.
Un thread detached no es *joinable*

Atributos de los Hilos

- `int pthread_attr_setdetachstate(pthread_attr_t *attr, void *detachstate);`
Establece el estado *detach* de un atributo de thread
- `int pthread_attr_setscope(pthread_attr_t *attr, int contention_scope);`
Establece el ámbito de contienda del atributo de un thread
- `int pthread_attr_getscope(pthread_attr_t *attr, int *contention_scope);`
Devuelve el ámbito de contienda del atributo de un thread

Atributos de los Hilos

- `int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);`

Establece la política de planificación del atributo de un thread

- `int pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy);`

Devuelve la política de planificación del atributo de un thread

Atributos de los Hilos

```
#include <pthread.h>

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

- **pthread_attr_init/destroy**: Manipulación atributos de un hilo
- **pthread_attr_init** inicializa el objeto de atributos de un hilo **attr** y establece los valores por defecto
- Posteriormente, este objeto, con los atributos por defecto de un hilo, se puede utilizar para crear múltiples hilos
- **pthread_attr_destroy**, destruye el objeto de atributos de un hilo, **attr**, y éste no puede volver a utilizarse hasta que no se vuelva a inicializar

Atributos de los Hilos

Atributos:

- **detachstate:** controla si otro hilo podrá esperar por la terminación de este hilo (mediante la invocación a *pthread_join()*):
 - ✓ `PTHREAD_CREATE_JOINABLE` (valor por defecto)
 - ✓ `PTHREAD_CREATE_DETACHED`
- **schedpolicy:** controla cómo se planificará el hilo
 - ✓ `SCHED_OTHER` (valor por defecto, planificación normal + no tiempo real)
 - ✓ `SCHED_RR` (Round Robin + tiempo real + privilegios root)
 - ✓ `SCHED_FIFO` (First In First Out + tiempo real + privilegios root)
- **scope:** controla a qué nivel es reconocido el hilo
 - ✓ `PTHREAD_SCOPE_SYSTEM` (valor por defecto, el hilo es reconocido por el núcleo)
 - ✓ `PTHREAD_SCOPE_PROCESS` (no soportado en la implementación LinuxThreads de hilos POSIX)

Atributos de los Hilos

```
int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate);  
int pthread_attr_getdetachstate (const pthread_attr_t *attr, int *detachstate);  
int pthread_attr_setschedpolicy (pthread_attr_t *attr, int policy);  
int pthread_attr_getdetachstate (const pthread_attr_t *attr, int *policy);  
int pthread_attr_setscope (pthread_attr_t *attr, int contentionscope);  
int pthread_attr_getscope (const pthread_attr_t *attr, int *contentionscope);
```


Espera por terminación: *pthread_join()*

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

```
void pthread_join(pthread_t tid, void **status);
```

- Esta función **suspende la ejecución del hilo** que la invoca **hasta** que el hilo identificado por el valor **tid** **finaliza**, bien por la invocación a la función **pthread_exit** o por estar cancelado
- Si **status** no es **NULL**, el valor devuelto por el hilo (el argumento de la función **pthread_exit**, cuando el hilo hijo finaliza) se almacena en la dirección indicada por **status**
- El valor devuelto es o bien el argumento de la función **pthread_exit** o el valor **PTHREAD_CANCELED** si el hilo **tid** está cancelado
- El hilo por el que se espera su terminación debe estar en estado sincronizable (**joinable state**)
 - ✓ Cuando un hilo en este estado termina, no se liberan sus propios recursos (descriptor del hilo y pila) hasta que otro hilo espere por él
 - ✓ La espera por la terminación de un hilo para el cual ya hay otro hilo esperando, genera un error

Programa 3

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 3
void *BusyWork(void *null){
    int i; double result=0.0;
    for (i=0; i < 1000000; i++) {
        result = result + (double)random(); }
    printf("result = %d\n",result);
    pthread_exit((void *) 0);
}
void main(){
    pthread_t thread[NUM_THREADS];
    pthread_attr_t attr;
    int rc, t, status;
```

Programa 3

```
pthread_attr_init(&attr); /* Initialize and set thread detached attribute */
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_UNDETACHED);
for(t=0;t < NUM_THREADS;t++) {
    printf("Creating thread %d\n", t);
    pthread_create(&thread[t], &attr, BusyWork, NULL);}

/* libero atributo y espero a los otros hilos */
pthread_attr_destroy(&attr);
for(t=0;t < NUM_THREADS;t++)
{ pthread_join(thread[t], (void **)&status);
printf("Completed join with thread %d status= %d\n",t, status); }
pthread_exit(NULL);}
```

Programa 3

```
Creating thread 0  
Creating thread 1  
Creating thread 2  
result = 125025002  
Completed join with thread 0  
result = 1125026112  
result = 1125026256  
Completed join with thread 1  
Completed join with thread 2
```

Observe como
se muestra el
efecto de JOIN




Cancelación de Hilos: *pthread_cancel()*

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

```
void pthread_cancel(pthread_t tid);
```

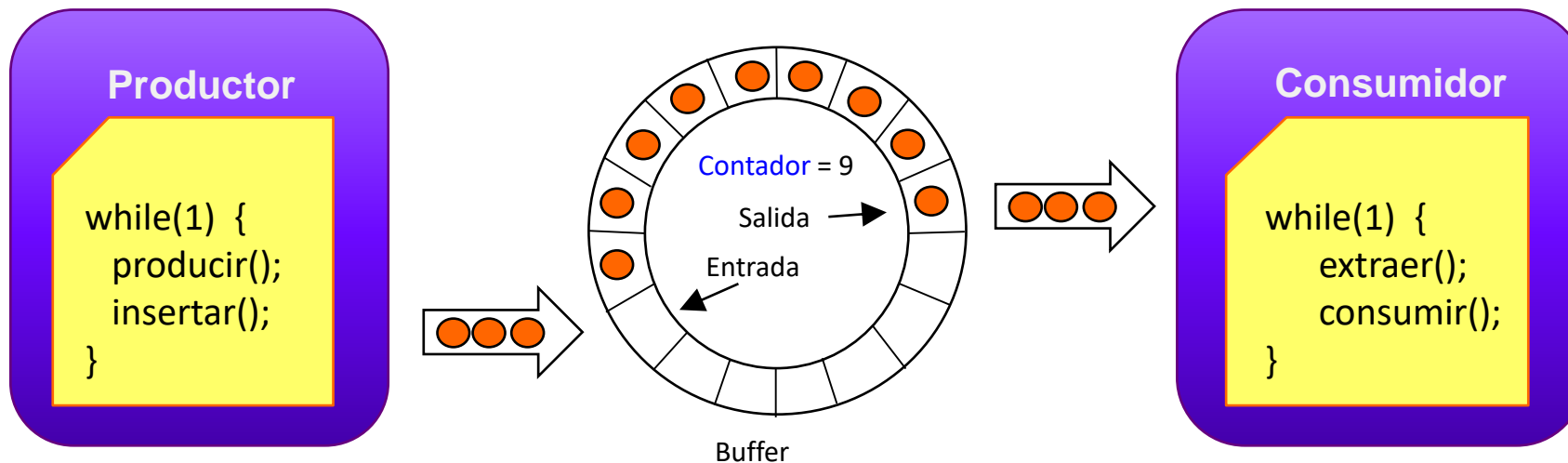
- La cancelación es el mecanismo por el cual un hilo puede solicitar la terminación de la ejecución de otro
- Dependiendo de la configuración del hilo al que se solicita su cancelación, puede aceptar peticiones de cancelación (**PTHREAD_CANCEL_ENABLE**, estado por defecto) o rechazarlas (**PTHREAD_CANCEL_DISABLE**)
- En caso de aceptar peticiones de cancelación, un hilo puede completar la cancelación de dos formas diferentes:
 - ✓ De forma asíncrona (**PTHREAD_CANCEL_ASYNCHRONOUS**), o
 - ✓ De forma diferida (**PTHREAD_CANCEL_DEFERRED**, valor por defecto) hasta que se alcance un punto de cancelación: Un punto de cancelación es un punto en el flujo de control de un hilo en el que se comprueba si hay solicitudes de cancelación pendientes
- Cuando un hilo acepta una petición de cancelación, el hilo actúa como si se hubiese realizado la siguiente invocación **pthread_exit(PTHREAD_CANCELED)**



Comunicación entre Tareas

Sincronización de Hilos: Problemática

- Ejemplo: el problema del ‘productor/consumidor’ (o del buffer acotado)
 - ✓ Existen dos tipos de entidades: **productores** y **consumidores** (de ciertos “items” o elementos de datos)
 - ✓ Existe un buffer acotado (cola circular) que acomoda la diferencia de velocidad entre productores y consumidores:
 - ✓ Si el buffer se llena, los productores deben suspenderse
 - ✓ Si el buffer se vacía, los consumidores deben suspenderse



Mutex: exclusión mutua

- Se utilizan para bloquear el acceso a recursos de hardware o software que deben ser compartidos por distintas tareas
- El **mutex** actúa como una ticket (**token**) que debe ser adquirido por la tarea que desea acceder al recurso compartido
- Una vez que una tarea adquiere el mutex asociado a un recurso, ningún otra tarea puede adquirirlo hasta que sea liberado por la tarea que lo adquirió primero
- Su uso correcto es responsabilidad del programador

NOTA

- En todos los casos presentados es de destacar que su uso correcto es responsabilidad del programador, habiendo algunos problemas de sincronización a evitar:
- **Deadlocks (bloqueo mútuo)**: cuando dos o más tareas concurrentes se encuentran c/u esperando a la otra para proseguir (lo que nunca ocurrirá)
- **Starvation (inanición)**: cuando a una tarea se le niega el acceso a un recurso compartido
- **Inversión de prioridades**: cuando dos tareas de distinta prioridad comparten un recurso y la de menor prioridad bloquea el recurso antes que la de prioridad mayor, bloqueándose esta última al momento que precise el uso del recurso compartido

Sincronizar hilos

- Hay dos mecanismos que permiten sincronizar hilos:
 - ✓ Un **mutex** es una variable que proporciona exclusión mutua mediante dos operaciones, **lock** y **unlock**
 - ✓ una **variable de condición** proporciona sincronización condicional mediante dos operaciones, **wait** y **signal**
- Ambos tipos de variables pueden tener **atributos**

MUTEX

- `int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);`
Inicializa un mutex con ciertos atributos.
- `int pthread_mutex_destroy(pthread_mutex_t *mutex);`
Destruye el mutex. Comportamiento indefinido si el mutex está cerrado.
- `int pthread_mutexattr_init (attr)`
Crea un objeto atributo mutex e inicializa con valores por defecto.
- `int pthread_mutexattr_destroy (attr)`
Elimina el objeto de atributo de mutex attr

MUTEX

- `int pthread_mutex_lock(pthread_mutex_t *mutex);`

Cierra el mutex..

El propietario del mutex es el thread que lo cerró.

- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`

Abre el mutex cuando lo invoca su thread propietario Comportamiento indefinido si no es el thread propietario o el mutex está abierto

Activa un thread suspendido en el mutex.

Programa 4

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

struct {
    int data[1];
    pthread_mutex_t mutex;
} buffer;

int main(void) {
    pthread_t rd;
    pthread_attr_t attr;
    int valores[50], indice,in, c,x;
    void *status;
    void *consumidor(void *);
    void escritura(int);
```

```
pthread_mutex_init(&buffer.mutex,NULL)
pthread_attr_init(&attr);

pthread_attr_setdetachstate(&attr,PTHREAD
                            _CREATE_JOINABLE );

/* create hilo consumidor */
thread_create(&rd,&attr,consumidor,NULL)

/* lleno valores a escribir y leer*/
for(indice=0;indice<15;indice++)
    { valores[indice]=indice; }
    valores[indice]=50;
```

Programa 4

```
/* HILO PRODUCTOR (MAIN) */
indice=0;
do {  c=valores[indice++];
      escritura(c);
} while (c !=50);
/* espera hasta que finalice el consumidor */
pthread_join(rd,&status)
printf("fin del programa\n");
exit(0);
}

/* HILO CONSUMIDOR) */
void *consumidor(void *arg) {
  int c,yy;
  do {  c = lectura();

        } while (c != 50);
}
```

Programa 4

```
void escritura(int c) {
```

```
    int xx;
```

```
    pthread_mutex_lock(&buffer.mutex); /* lock mutex */
```

```
    buffer.data[1] = c; /* inserta un dato */
```

```
    pthread_mutex_unlock(&buffer.mutex); /* unlock mutex */
```

```
}
```

```
int lectura() {
```

```
    int elem, c;
```

```
    pthread_mutex_lock(&buffer.mutex); /* lock mutex */
```

```
    elem = buffer.data[1]; /* lee un dato */
```

```
    c=elem;
```

```
    printf("%3d",c);
```

```
    pthread_mutex_unlock(&buffer.mutex); /* unlock mutex */
```

```
    return elem;
```

```
}
```

Variable de condición

- Variables de Condición(VC) proveen otra manera de sincronizar hilos. Mientras que los **Mutexs** implementan sincronización controlando al hilo el acceso de dato, las **variables de condición** permiten sincronizar hilos en base al **valor actual** del dato
- Sin VC, el programador podría tener hilos que continuamente estén **polling** (posiblemente en una sección crítica), para verificar si una condición se mantiene.

Variable de condición

- **pthread_cond_init(condition, attr)**

Inicializa una vc y fija sus atributos

- **pthread_cond_destroy(condition)**

libera una VC que ya no es necesario

- **pthread_condattr_init(attr)**

- **pthread_condattr_destroy(attr)**

estas rutinas son usado para crear y destruir atributos de la variable de condición.

Variable de condición

- **pthread_cond_wait (condition, mutex)**

bloquea al hilo que llama hasta que la condición especificada es señalada. Esta rutina deberá ser llamada mientras el mutex esta locked, y este librara automáticamente el mutex mientras este espera. Debe unlock el mutex después que la señal ha sido recibida

- **pthread_cond_signal(condition)**

se usa para señalar o despertar otro hilo el cual esta esperando sobre la VC. Este debe ser llamado después que el mutex es is locked, y debe unlock mutex para que thread_cond_wait()complete

Programa 5

```
#include <pthread.h>
#include <stdio.h>
struct {
    int data[1];                /*buffer comnpartido */
    pthread_mutex_t mutex;      /* mutex */
    pthread_cond_t cond;        /* variable de condition */
} buffer;

int nnElem=0;                  //indica si el buffer esta lleno/vacio
int main(void)
{
    pthread_t rd;               /* id del consumidor */
    pthread_attr_t attr;        /* atributos */
    int valores[50], indice, c,x;
    void *status;               /* exit thread status */
    void *consumidor(void *);   /* prototype */
    void write_to_buffer(int);  /* prototype */
}
```

Programa 5

```
indice=0;
/* inicializa mutex */
pthread_mutex_init(&buffer.mutex, NULL)
/* inicializa variable de condicion */
pthread_cond_init(&buffer.cond, NULL)
/* inicializa atributos de hilo */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,
                           PTHREAD_CREATE_JOINABLE );
/* crea hilo consumidor */
pthread_create(&rd,&attr,consumidor,NULL)
for(indice=0;indice<15;indice++)
{valores[indice]=indice;} valores[indice]=50;
```

```
/* HILO PRODUCTOR (MAIN) */
indice=0;
do { c = valores[indice++];
    write_to_buffer(c);
} while (c !=50);
pthread_join(rd,&status)
printf("Fin de Programa\n");
}
/* HILO CONSUMIDOR*/
void *consumidor(void *arg) { int c;
int read_from_buffer();
do { c = read_from_buffer();
} while (c != 50);
}
```

Programa 5

```
void write_to_buffer(int c) {  
    pthread_mutex_lock(&buffer.mutex); /* lock mutex */  
    if (nnElem == 1) //si esta lleno no escribe  
    /* no puede escribir, buffer full ==> espera */  
    pthread_cond_wait(&buffer.cond,&buffer.mutex);  
    buffer.data[tamano] = c; /* inserta data */  
    nnElem=nnElem+1; /* increase para indicar lleno */  
    /* despierta el hilo consumidor */  
    pthread_cond_signal(&buffer.cond);  
    pthread_mutex_unlock(&buffer.mutex); /* unlock mutex*/  
}
```

Programa 5

```
int read_from_buffer() {  
    int elem,c;  
    pthread_mutex_lock(&buffer.mutex); /* lock mutex */  
    if (nnElem == 0)  
        /* no puede leer, buffer vacio ==> espera */  
        pthread_cond_wait(&buffer.cond,&buffer.mutex);  
    elem = buffer.data[tamano]; /* consume un dato */  
    nnElem=nnElem-1; /* decrementa para indicar vacio */  
    /* despierta al hilo productor */  
    pthread_cond_signal(&buffer.cond);  
    pthread_mutex_unlock(&buffer.mutex); /* unlock mutex */  
    return elem;  
}
```