uc3m | Universidad Carlos III de Madrid

Programación con Hilos

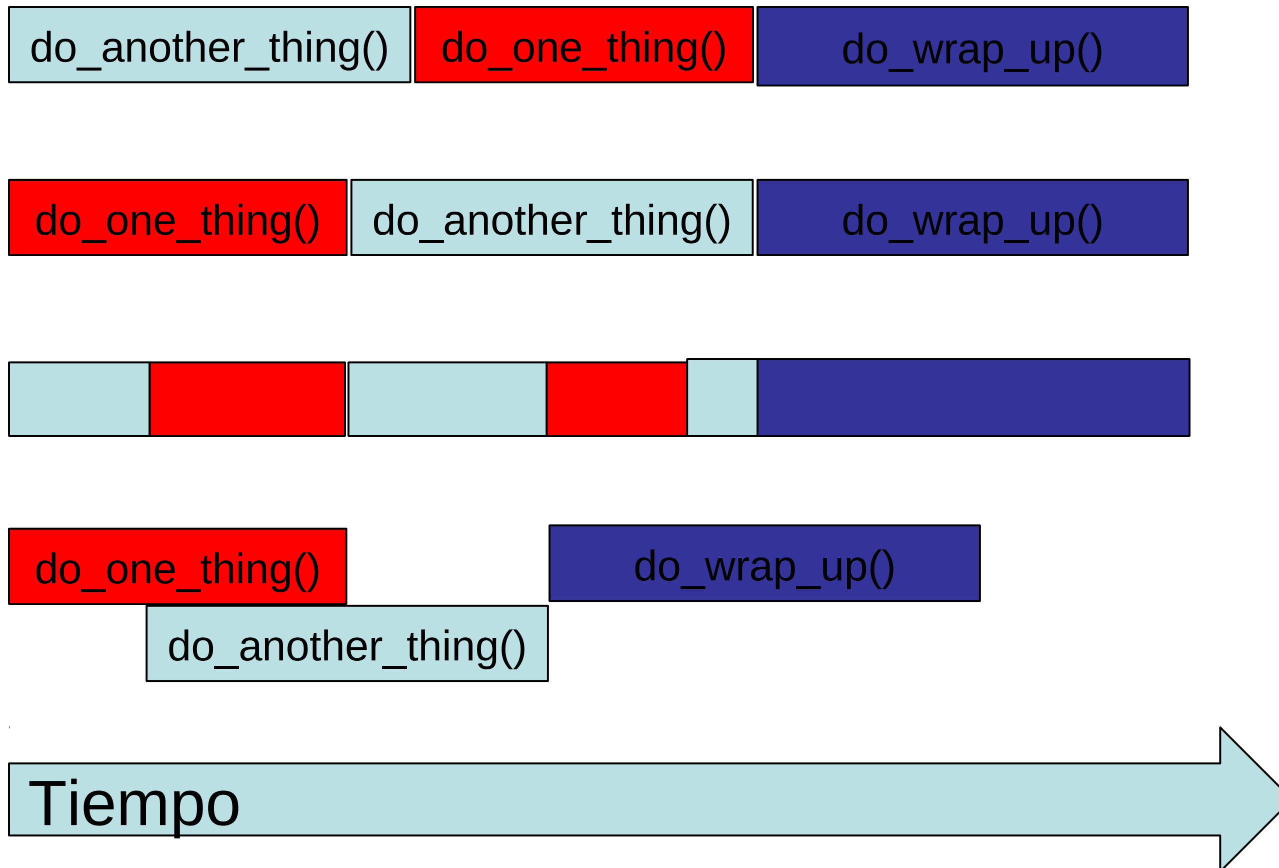
Contenido

- Conceptos generales de programación paralela
- Programación con hilos
- Pthreads
- Sincronización
- Exclusión mutua (Mutex)
- Variables condición (Var Condition)

Preliminares: Computación Paralela

- **Paralelismo**: Múltiples cálculos que se realizan simultáneamente.
  - A nivel de instrucción (pipelining)
  - Paralelismo de datos (SIMD)
  - Paralelismo de tareas (vergonzosamente paralelo)
- **Concurrencia**: Múltiples cálculos que ***pueden*** ser realizados simultáneamente.
- Concurrencia vs. Paralelismo

Concurrencia y Paralelismo



Procesos vs. Hilos

- **Proceso (Process)**: Instancia de un programa que se está ejecutando en su propio espacio de direcciones. En sistemas POSIX, cada proceso mantiene su propio heap, pila, registro, descriptores de ficheros, etc.

  Comunicación:
  - Memoria compartida
  - Red
  - Pipes, Colas
- **Hilo** (**Thread**): Una versión ligera del proceso que comparte dirección con otros hilos. En sistemas POSIX, cada hilo mantiene sus propios: registros, pila, señales.

  Comunicación:
  - Espacio de direcciones compartido

Concurrencia usando hilos

- **Ejecución serializada:**
  - Hasta ahora… nuestros programas constan de un solo hilo.
  - Un programa termina su ejecución cuando su hilo termina su ejecución.

- **Multi-hilos:**
  - Un programa es organizado como múltiples y concurrentes hilos de ejecución
  - El programa principal puede crear (spawns) muchos hilos
  - El hilo **puede** comunicarse con otro(s) hilo(s).
  - Ventajas potenciales:
    - Mejora de rendimiento
    - Mejores tiempos de respuesta
    - Mejora en la utilización de recursos
    - Menor cantidad de trabajo extra comparada con el uso de múltiples procesos.

Programación Multihilos

- Incluso en C, la programación multihilos puede ser realizada de diversas formas:
  - Pthreads: biblioteca POSIX C,
  - OpenMP,
  - Intel threading building blocks,
  - Cilk (de CSAIL!),
  - Grand central dispatch,
  - CUDA (GPU) y
  - OpenCL (GPU/CPU)

No todo el código es paralelizable

```
float params[10];

for (int i=0; i<10; i++)
    do_something ( params[i] );
```

```
float params[10];
float prev = 0;

for (int i=0; i<10; i++)
    prev = complicated ( params[i], prev);
```

Paralelizable

No paralelizable

No todo el código multi-hilos es seguro

```
int balance =500;
void deposit ( int sum ) {
        int currbalance=balance ; /∗ read balance ∗/
        ...
        currbalance+=sum ;
        balance=currbalance ; /∗ write balance ∗/
}
void withdraw ( int sum ) {
        int currbalance=balance ; /∗ read balance ∗/
        if ( currbalance >0)
            currbalance−=sum ;
        balance=currbalance ; /∗ write balance ∗/
}
  ... deposit (100); /∗ thread 1∗/
  ... withdraw (50); /* thread 2∗/
 . .. withdraw(100); /∗ thread 3∗/
        ...
```

Escenario: T1(read),T2(read,write),T1(write) ,balance=600
Escenario: T2(read),T1(read,write),T2(write) ,balance=450

API del **pthread**

API:

- Gestión de Hilos: crear, unir, atributos

  pthread__

- Exclusión mutua: crear, destruir exclusión mutua

  pthread_mutex_

- Variables condición: crear, destruir, esperar y continuar

  pthread_cond_

- Sincronización: lectura/escritura candados y barreras

  pthread_rwlock_, pthread_barrier_

API:

```
#include <pthread.h>
gcc -Wall -O0 -o <output> file.c -pthread (no -l prefix)
```

Creación de hilos

```
int pthread_create( pthread_t *thread,
                    const pthread _attr _t  *attr,
                      void *(* start _routine )( void *),
                      void *arg );
```

- Crea un nuevo hilo que tiene los atributos especificados por **`attr`**.
- Utiliza las opciones de defecto de atributos cuando **`attr`** es **`NULL`**.
- Si no se producen errores en la creación, se almacena el hilo en `thread`
- Se invoca la función **`start_routine(arg)`** en un hilo de ejecución diferente.
- Regresa cero si triunfa la creación, diferente de cero si se produce un error.

```
void pthread_exit(void *value_ptr);
```

- Se invoca implícitamente cuando termina la función thread.
- Análogo con **`exit`**`()`

Ejemplo

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

void ∗PrintHello ( void ∗threadid )
{
     long tid ;
     tid = ( long ) threadid ;
     printf( " Hello World ! It's me, thread #%ld ! \ n " , tid ) ;
     pthread _exit(NULL ) ;
}
int main ( int argc , char ∗argv [ ] )
{
   pthread_t threads [NUM_THREADS] ;
   int r c ;
   long t ;
   for ( t =0; t <NUM_THREADS; t ++) {
       printf ( " In  main : c reat ing thread %l d \ n " , t ) ;
       rc = pthread_create (& threads [ t ] , NULL, PrintHello( void ∗) t  ) ;
       if ( rc ) {
            printf("ERROR;  return code from pthread_create ( ) i s %d \ n ", rc ) ;
            exit( −1);
        }
   }
   pthread_exit(NULL ) ;
}
```

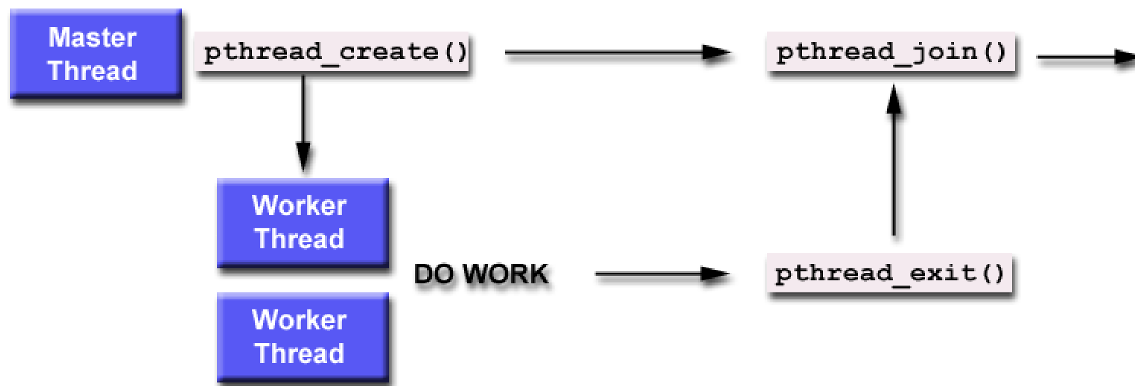
Traza de salida

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
Hello World! It's me, thread #1!
In main: creating thread 2
In main: creating thread 3
Hello World! It's me, thread #2!
Hello World! It's me, thread #3!
In main: creating thread 4
Hello World! It's me, thread #4!
```

```
In main: creating thread 0
Hello World! It's me, thread #0!
In main: creating thread 1
Hello World! It's me, thread #1!
In main: creating thread 2
Hello World! It's me, thread #2!
In main: creating thread 3
Hello World! It's me, thread #3!
In main: creating thread 4
Hello World! It's me, thread #4!
```

Sincronización: unión (joining)

- "Joining" es una manera de lograr la sincronización entre hilos. Ejemplo:



int pthread_join(pthread_t thread, **void** **value_ptr);

- `pthread_join()` Bloquea el thread que hace la llamada hasta que el thread especificado como atributo termina su ejecución.
- Si `value_ptr` es distinto de null, contiene el estatus de finalización del thread invocado.

Otras formas de sincronización: exclusión mutua (mutex), variables de condición

Exclusión mutua

- Mutex (exclusión mutua) actúa como un “candado" para proteger el acceso a un dato.
- Solo un hilo a la vez “posee” el candado. Los hilos deben turnarse para acceder al dato.

```
int pthread_mutex_destroy ( pthread_mutex_t *mutex ) ;
int pthread_mutex_init ( pthread_mutex_t *mutex ,
                         const pthread_mutexattr_t attr );
thread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

  - `pthread_mutex_init()` inicializa un “mutex”. Si los atributos valen NULL, entonces se usan los valores de defecto de la librería.
  - La macro `PTHREAD_MUTEX_INITIALIZER` puede usarse para inicializar “mutexes” estáticos.
  - `pthread_mutex_destroy()` destruye el “mutex”.
  - Ambas funcionan regresan 0 si no hay error, y un valor distinto de cero si se produce algún error.

Exclusión mutua

```
int pthread_mutex_lock    ( pthread_mutex_t *mutex ) ;
int pthread_mutex_trylock ( pthread_mutex_t *mutex ) ;
int pthread_mutex_unlock  ( pthread_mutex_t *mutex ) ;
```

- `pthread_mutex_lock()` bloquea (cierra) la variable mutex. Si la variable mutex está bloqueada (cerrada), la función que hizo la llamada se bloquea hasta que la variable mutex esté disponible (no-bloqueada).
- `pthread_mutex_trylock()` es la versión no-bloqueante. Si la variable mutex está bloqueada (cerrada), la llamada regresa inmediatamente.
- `pthread_mutex_unlock()` desbloquea (abre) la variable mutex.

Ejemplo revisado

```
int balance =500;
void deposit ( int sum ) {
        int currbalance=balance ; /∗ read balance ∗/
        ...
        currbalance+=sum ;
        balance=currbalance ; /∗ write balance ∗/
}
void withdraw ( int sum ) {
        int currbalance=balance ; /∗ read balance ∗/
        if ( currbalance >0)
              currbalance−=sum ;
        balance=currbalance ; /∗ write balance ∗/
}
   ... deposit (100); /∗ thread 1∗/
   ... withdraw (50); /* thread 2∗/
   ... withdraw(100); /∗ thread 3∗/
        ...
```

Escenario: T1(read),T2(read,write),T1(write) ,balance=600
Escenario: T2(read),T1(read,write),T2(write) ,balance=450

Usando Exclusión Mutua

```
int balance =500;
pthread_mutex_t mutexbalance=PTHREAD_MUTEX_INITIALIZER;
void deposit ( int sum ) {
        pthread_mutex_lock(&mutexbalance );
        {
           int currbalance=balance ; /∗ read balance ∗/
           ...
           currbalance+=sum ;
           balance=currbalance ; /∗ write balance ∗/
           }
         pthread_mutex_unlock(&mutexbalance );
}
void withdraw ( int sum ) {
         pthread_mutex_lock(&mutexbalance );
        {
           int currbalance=balance ; /∗ read balance ∗/
           if ( currbalance >0)
                currbalance−=sum ;
                balance=currbalance ; /∗ write balance ∗/
           }
        pthread_mutex_unlock(&mutexbalance );
}
   ... deposit (100); /∗ thread 1∗/
   ... withdraw (50); /* thread 2∗/
   ... withdraw(100); /∗ thread 3∗/
   ...
```

Escenario: T1(read),T2(read,write),T1(write) ,balance=550

Escenario: T2(read),T1(read,write),T2(write) ,balance=550

Ejemplo: Versión secuencial del producto de dos vectores

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

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

#define VECLEN 100000
   DOTDATA dotstr;
```

```
void dotprod() {
    int start, end, i;
    double mysum, *x, *y;
    start=0;
    end = dotstr.veclen;
    x = dotstr.a;
    y = dotstr.b;
    mysum = 0;
    for (i=start; i<end ; i++) {
       mysum += (x[i] * y[i]);
    }
    dotstr.sum = mysum;
}
```

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

   len = VECLEN;
   a = (double*) malloc (len*sizeof(double));
   b = (double*) malloc (len*sizeof(double));

   for (i=0; i<len; i++) {
      a[i]=1; b[i]=a[i]; }
      dotstr.veclen = len;
      dotstr.a = a;
      dotstr.b = b;
      dotstr.sum=0;
      dotprod ();
      printf ("Sum = %f \n", dotstr.sum);
      free (a);
      free (b);
  }
```

Ejemplo: Versión concurrente del producto de dos vectores

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

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

#define NUMTHRDS 4
#define VECLEN 100000
   DOTDATA dotstr;
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;
```

```
void *dotprod(void *arg)
{

   int i, start, end, len ;
   long offset;
   double mysum, *x, *y;
   offset = (long)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;
   printf("Thread %ld did %d to %d:  mysum=%f global sum=%f\n",
   offset,start,end,mysum,dotstr.sum);
   pthread_mutex_unlock (&mutexsum);

   pthread_exit((void*) 0);
}
```

Ejemplo: Versión concurrente del producto de dos vectores

```
int main (int argc, char *argv[])
{
long i;
double *a, *b;
void *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;
  b[i]=a[i];
  }

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

pthread_mutex_init(&mutexsum, NULL);

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,
       PTHREAD_CREATE_JOINABLE);

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

pthread_attr_destroy(&attr);

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

printf ("Sum =  %f \n", dotstr.sum);
free (a);
free (b);
pthread_mutex_destroy(&mutexsum);
pthread_exit(NULL);
}
```

Variables de Condición

En ocasiones, bloquear o desbloquear un mutex depende de una condición ocurre en ejecución. Sin variables condicionales los programas tendrían que permanecer en espera-ocupada (hacer "polling" sobre datos) continuamente.

Consumidor:

(a) Bloquea (cierra) el mutex que protege a la variable global ítem.

(b) Espera por (item>0) que es una signal que envía el productor (así el mutex se desbloqueará automáticamente).

(c) Se despierta cuando el productor envíe signal (el mutex se bloquea de nuevo automáticamente), desbloquear el mutex y consumir ítem

Productor:

(1) Produce algo

(2) Bloquea (cierra) variable global mutex que protege a ítem, actualiza el ítem.

(3) Despierta (envía una "signal") hilos que están esperando

(4) Desbloquea (abre) la variable mutex

Variables condición

```
int pthread_cond_destroy ( pthread_cond_t *cond );
int pthread _cond_init(pthread_cond_t *cond,
                                  const pthread_condattr_t *attr );
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- `pthread_cond_init()` inicializa la variable condición. Si attr es NULL, se usan los atributos definidos por defecto en la librería.
- `pthread_cond_destroy()` destruirá la variable condición. Cuidado, al destruir una variable condición sobre la que otros hilos están actualmente bloqueados ocurrirá un comportamiento inesperado.
- macro `PTHREAD_COND_INITIALIZER` puede usarse para inicializar variables condición. No se realizan verificaciones sobre errores.
- Ambas funciones regresan 0 si no ocurren problemas y un valor diferente de cero si se produce un error.

Variables condición

int pthread_cond_wait(pthread_cond_t ∗cond,pthread_mutex_t ∗mutex);

- Bloquea sobre una variable condición.
- Debe ser invocada sobre un mutex que esté bloqueado (cerrado), de otro modo ocurrirá un comportamiento inesperado.
- Libera automáticamente el mutex
- Si no se produce error, el control de ejecución regresa a la función que invocó esta operación y el mutex se bloquea automáticamente de nuevo.

int pthread_cond_broadcast(pthread_cond_t ∗cond);

int pthread_cond_signal(pthread_cond_t ∗cond);

- desbloquea threads esperando sobre una variable condición.
- `pthread_cond_broadcast()` desbloquea todos los hilos que están esperando
- `pthread_cond_signal()` desbloquea **uno de los** hilos que están esperando
- Ambas funciones regresan 0 si no ocurren problemas y un valor diferente de cero si se produce un error.

Ejemplo

```
#include <pthread .h>
    pthread_cond_t   cond_recv  =PTHREAD_COND_INITIALIZER;
    pthread_cond_t   cond_send =PTHREAD_COND_INITIALIZER;
    pthread_mutex_t cond_mutex=PTHREAD_MUTEX_INITIALIZER;
    pthread_mutex_t count_mutex=PTHREAD_MUTEX_INITIALIZER;
int full =0;
int count =0;
```

```
void *produce(void*)
{
 while(1)
 {
  pthread_mutex_lock(&cond_mutex);
  while( full )
  {
    pthread_cond_wait(&cond_recv,
                      &cond_mutex);
  }
    pthread_mutex_unlock(&cond_mutex);
    pthread_mutex_lock(&count_mutex);
    count++;
    full = 1;
    printf("produced(%d):%d\n",
           pthread_self(), count();
    pthread_cond_broadcast(&cond_send);
   pthread_mutex_unlock(&count_mutex);
   if (count >=10) break;
 }
}
```

```
void *consume(void*)
{
 while(1)
 {
  pthread_mutex_lock(&cond_mutex);
  while( ! full )
  {
    pthread_cond_wait(&cond_send,
                      &cond_mutex);
  }
     pthread_mutex_unlock(&cond_mutex);
     pthread_mutex_lock(&count_mutex);
    count--;
    full = 0;
     printf("consumed(%d):%d\n",
            pthread_self(), count);
     pthread_cond_broadcast(&cond_recv);
     pthread_mutex_unlock(&count_mutex);
     if (count >=10) break;
 }
}
```

Ejemplo

```
int main () {
        pthread_t cons_thread , prod_thread ;
        pthread_create(&prod_thread ,NULL,produce ,NULL);
        pthread_create(&cons_thread ,NULL,consume,NULL);
        pthread _join(cons_thread ,NULL);
        pthread _join(prod_thread ,NULL);
        return 0;
        }
```

Salida:

```
        produced (3077516144):1
        consumed(3069123440):1
        produced (3077516144):2
        consumed(3069123440):2
        produced (3077516144):3
        consumed(3069123440):3
        produced (3077516144):4
        consumed(3069123440):4
        produced (3077516144):5
        consumed(3069123440):5
        produced (3077516144):6
        consumed(3069123440):6
        produced (3077516144):7
        consumed(3069123440):7
```

uc3m | Universidad Carlos III de Madrid

Gracias por su atención