Master en Computación

Programación Concurrente

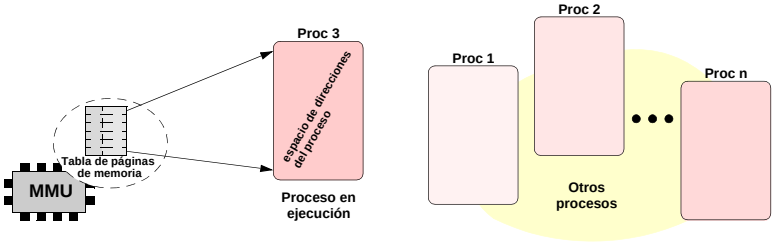
Bloque II: Programación concurrente en POSIX

Tema 3. Gestión de Threads

3.1 Procesos y threads

El modelo de proceso proporciona protección, pero es inadecuado para sistemas que requieren alta eficiencia:

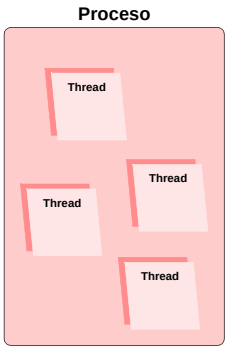
- **Tiempos de cambio de contexto altos**
- **Tiempos de creación y destrucción altos**
- **Necesidad de hardware especial (MMU)**



Threads (cont.)

POSIX.1 define interfaces para soportar múltiples threads o flujos de control en cada proceso POSIX

- **Los threads de un proceso comparten un único espacio de direcciones**
- **El estado asociado es pequeño**
- **Los tiempos de cambio de contexto son menores**
- **Los tiempos de creación y destrucción son menores**
- **Son similares a las tareas de un kernel de tiempo real de alta eficiencia**



Sistema mínimo (MaRTE OS): existe un único proceso

3.2 Threads POSIX: conceptos básicos

Thread:

- **un flujo de control simple perteneciente a un proceso**
- **tiene un identificador de thread (`tid`)**
- **el `tid` solo es válido para threads del mismo proceso**
- **tiene su propia política de planificación, y los recursos del sistema necesarios, tales como su propio stack, etc**
- **todos los threads de un proceso comparten un único espacio de direccionamiento**

Proceso en una implementación multi-thread:

- **un espacio de direccionamiento con uno o varios threads**
- **inicialmente contiene un solo thread: el thread principal**
- **el thread principal es un thread más, salvo que cuando él termina, todos los demás también**

Threads POSIX: conceptos básicos (cont.)

Servicios bloqueantes:

- **sólo se suspende el thread que invoca el servicio**
  En MaRTE OS sobre Linux se bloquea todo el proceso

Los threads tienen dos estados posibles para controlar la devolución de recursos al sistema:

- **"*detached*" o independiente: `PTHREAD_CREATE_DETACHED`**
  - **cuando el thread termina, devuelve al sistema los recursos utilizados (`tid`, stack, etc)**
  - **no se puede esperar su terminación**
- **"*joinable*" o sincronizado: `PTHREAD_CREATE_JOINABLE`**
  - **cuando el thread termina, mantiene sus recursos**
  - **se liberan cuando otro thread llama a `pthread_join()`**
  - **por defecto un thread es "*joinable*"**

3.3 Creación de threads

Para crear un thread es preciso definir sus atributos en un objeto especial (`pthread_attr_t`)

El objeto de atributos

- **debe crearse antes de usarlo: `pthread_attr_init()`**
- **puede borrarse: `pthread_attr_destroy()`**
- **se pueden modificar o consultar atributos concretos del objeto (pero no los del thread, que se fijan al crearlo)**

Los atributos definidos son:

- **tamaño de stack mínimo (opcional)**
- **dirección del stack (opcional)**
- **control de devolución de recursos ("*detach state*")**
- **atributos de planificación (tema 5)**

Atributos de creación: interfaz

```
#include <pthread.h>

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

int pthread_attr_setstacksize (pthread_attr_t *attr,
                               size_t stacksize);
int pthread_attr_getstacksize (const pthread_attr_t *attr,
                               size_t *stacksize);

int pthread_attr_setstackaddr (pthread_attr_t *attr,
                               void *stackaddr);
int pthread_attr_getstackaddr (const pthread_attr_t *attr,
                                  void **stackaddr);

int pthread_attr_setdetachstate (pthread_attr_t *attr,
                                    int detachstate);
int pthread_attr_getdetachstate (const pthread_attr_t *attr,
                                    int *detachstate);
```

```
PTHREAD_CREATE_DETACHED
PTHREAD_CREATE_JOINABLE
```


Llamada para creación de threads

Interfaz:

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

Esta función:

- **crea un nuevo thread con atributos especificados por `attr`**
- **devuelve el `tid` del nuevo thread en `thread`**
- **el thread se crea ejecutando `start_routine(arg)`**
- **si `start_routine` termina, es equivalente a llamar a `pthread_exit` (observar que esto difiere del thread main)**

Ejemplo de creación de threads

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <misc/error_checks.h>

// Thread que pone pseudo-periódicamente un mensaje en pantalla
// el periodo se le pasa como parámetro
void *periodic (void *arg) {
      int period;

      period = *((int *)arg);
      while (1) {
         printf("En el thread con periodo %d\n",period);
         sleep (period);
      }
}
```


Ejemplo de creación de threads (cont.)

```
// Programa principal que crea dos threads pseudo-periódicos
int main ()
{
      pthread_t th1,th2;
      pthread_attr_t attr;
      int period1=2
      int period2=3;
      // Crea el objeto de atributos para crear threads independientes
      CHK( pthread_attr_init(&attr) );
      CHK( pthread_attr_setdetachstate(&attr,
                                       PTHREAD_CREATE_DETACHED) );

      // Crea los threads
      CHK( pthread_create(&th1, &attr, periodic, &period1) );
      CHK( pthread_create(&th2, &attr, periodic, &period2) );

      // Les deja ejecutar un rato y luego termina
      sleep(30);
      printf("thread main terminando \n");
      exit (0);
}
```


3.4 Terminación de threads

Función para terminación de threads:

```
#include <pthread.h>
void pthread_exit (void *value_ptr);
```

Esta función termina el thread y hace que el valor apuntado por `value_ptr` esté disponible para una operación "*join*"

- **se ejecutan las rutinas de cancelación pendientes**
- **al terminar un thread es un error acceder a sus variables locales**

Terminación de threads (cont.)

Se puede esperar (*join*) a la terminación de un thread cuyo estado es sincronizado (*joinable*), liberándose sus recursos:

```
#include <pthread.h>
int pthread_join (pthread_t thread,
                  void **value_ptr);
```

También se puede cambiar el estado del thread a "*detached*", con lo que el thread, al terminar, libera sus recursos:

```
#include <pthread.h>
int pthread_detach (pthread_t thread);
```

Por defecto el estado de un thread es "*joinable*"

- **Existen también funciones para cancelar threads, habilitar o inhibir la cancelación, etc. (ver el manual).**

Ejemplo: uso de `pthread_join`

```
#include <pthread.h>
#include <stdio.h>
#include <misc/error_checks.h>

#define MAX   500000
#define MITAD (MAX/2)

typedef struct {
  int *ar;
  long n;
} subarray_t;

// Thread que incrementa n componentes de un array
void * incrementer (void *arg) {
  long i;
  subarray_t * subarray = ((subarray_t *)arg);

  for (i=0; i < subarray->n; i++) {
    subarray->ar[i]++;
  }
  return NULL;
}
```



```
// programa principal que reparte entre dos threads el trabajo
// de incrementar los componentes de un array
int main() {
  int ar [MAX] = { 0 }; // inicializa los elementos a 0
  pthread_t th1,th2;
  subarray_t sb1,sb2;
  long suma=0, i;

  // crea los theads
  sb1.ar = &ar[0]; sb1.n = MITAD; // primera mitad del array
  CHK( pthread_create(&th1, NULL, incrementer, &sb1) );

  sb2.ar = &ar[MITAD]; sb2.n = MITAD; // segunda mitad del array
  CHK( pthread_create(&th2, NULL, incrementer, &sb2) );

  // sincronizacion de espera a la finalizacion
  CHK( pthread_join(th1, NULL) );
  CHK( pthread_join(th2, NULL) );
  printf ("Ambos threads han finalizado \n");

  // muestra resultados
  for (i=0; i<MAX; i++)
    suma=suma+ar[i];
  printf ("Suma=%d\n",suma);
  return 0;
}
```

3.5 Identificación de threads

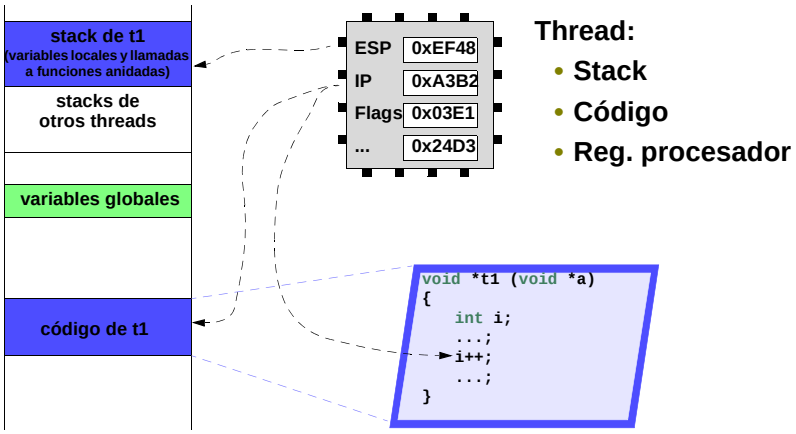
Identificación del propio thread:

```
pthread_t pthread_self(void);
```

Comparación de `tids`:

```
int pthread_equal (pthread_t t1, pthread_t t2);
```

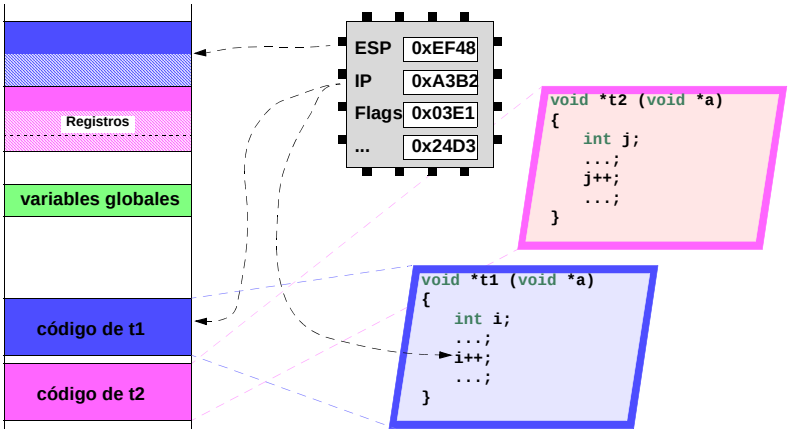

3.6 Implementación de la concurrencia a nivel de threads



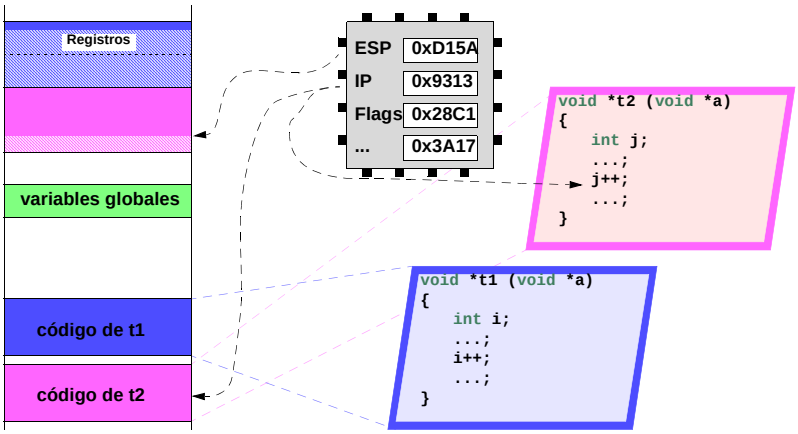
Thread:

- **Stack**
- **Código**
- **Reg. procesador**

Cambio de contexto



Cambio de contexto (cont.)



Implementación de threads a nivel de librería