# pthreads: uso básico

Rubén García Rodríguez
Alexis Quesada Arencibia
Eduardo Rodríguez Barrera
Francisco J. Santana Pérez
José Miguel Santos Espino

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

# PROCESOS PESADOS E HILOS

- Las llamadas al sistema fork() y exec() nos permiten crear **procesos pesados**.
- Cada proceso pesado tiene su propio espacio de memoria y puede estar ejecutando una aplicación diferente al resto de los procesos del sistema.
- Dentro de un proceso pesado pueden existir múltiples **procesos ligeros o hilos**, que comparten el mismo espacio de memoria del proceso pesado.

# LA BIBLIOTECA PTHREADS

- Es una API estándar (POSIX) para manejo de procesos ligeros.
- Escrita para lenguaje C.
- Ofrece servicios para:
  - Manejo de hilos (creación, sincronización básica…)
  - Sincronización con cerrojos y variables condición

# LA BIBLIOTECA PTHREADS

- Cómo se usa:
  - En el fuente: **#include <pthread.h>**
  - Al compilar: **gcc –lpthread …**
- Se usan tipos de datos para:
  - Hilos: **pthread_t**
  - Cerrojos: **pthread_mutex_t**
  - Variables condición: **pthread_cond_t**
- Todas las llamadas son de la forma pthread_*acción*(), o pthread_*objeto_acción* (…);
  - Ejemplo: pthread_create (…), pthread_mutex_lock(…)

# OPERACIONES BÁSICAS CON HILOS

- **pthread_create** ( &hilo, atribs, rutina, args );
- **pthread_exit** ( &resultado );
- **pthread_cancel** (hilo);
- **pthread_join** ( hilo, &resultado );
- **pthread_yield**();

# EJEMPLO BÁSICO

```
// rutina que ejecuta el hilo
// siempre debe ser void* f (void*)
void* escribe (void* arg)
{
  int max = (int)arg, j;
  for (j=1;j<=max;j++) {
    puts("soy un hilo\n");
  }
  return NULL;
}

…

pthread_t hilo;
pthread_create (&hilo, NULL, escribe, (void*)20);
// el hilo ejecutará escribe(20), de forma concurrente.
// Usamos conversión de tipos entre int y void*
// porque pthreads nos obliga a trabajar con el tipo void*
```

# ATRIBUTOS

- Algunas operaciones de pthreads llevan un argumento que permite pasar atributos al objeto que se está creando.
- Normalmente pondremos NULL en ese argumento (no vamos a hacer un uso avanzado de la biblioteca).

# TERMINACIÓN DE HILOS

- Un hilo termina cuando:
  - La función que ejecuta retorna
  - Llama a **pthread_exit()**.
  - Otro hilo lo cancela con **pthread_cancel()**.
  - El proceso pesado que lo contiene finaliza - ej. con **exit()** o con **exec()**
- Al finalizar, un hilo devuelve un valor de retorno, siempre de tipo **void***

# pthread_join()

- Muchas veces hace falta esperar a que cierto hilo finalice.
- Ojo, si la rutina main() termina, **todos** los hilos activos se cancelan automáticamente (no se espera a que los hilos acaben).
  - *Hay que esperar explícitamente por los otros hilos*
- **pthread_join(&h,&r)** espera hasta que el hilo h haya finalizado. En "r" deposita el valor de salida del hilo h.
  - Si no se quiere recoger nada, en *r* se puede pasar la dirección de una variable *tonta* de tipo void*.

# Pasar argumentos

- pthread_create() exige pasar como argumento un puntero **void***
- Para pasar al hilo varios parámetros, podemos utilizar una estructura (*struct*) y forzar cambios de tipo con void*, «engañando» al compilador.

# PASAR ARGUMENTOS: EJEMPLO (C99)

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

typedef struct {
  int nveces;
  char* texto;
} arg_t;

void* escribe (void* arg) {
  arg_t* args = (arg_t*)arg;
  for (int j=1; j<=args->nveces; j++) {
    printf("imprimo el texto: %s\n", args->texto);
  }
  return NULL;
}

int main() {
  pthread_t hilo;
  arg_t params = { .nveces=10, .texto="hola" };
  pthread_create (&hilo1, NULL, escribe, &params);
  void* dummy;
  pthread_join(hilo1,&dummy);
}
```

# CERROJOS Y VARIABLES CONDICIÓN

- Proporcionan sincronización con el modelo de monitores.
- **pthread_mutex_t:** cerrojos para regular el acceso a recursos en exclusión mutua
- **pthread_cond_t:** variables condición

# CERROJOS (MUTEX)

- *mutex* = "mutual exclusion". Sirve para adquirir en exclusiva el derecho de acceso a un recurso.
- Dos posibles estados: libre y adquirido. Inicialmente está libre.
- Operaciones:
  - **`pthread_mutex_lock(&mutex)`**
    → Adquiere el *mutex*. Si el *mutex* ya estaba adquirido, el hilo se bloquea hasta que lo consigue adquirir.
  - **`pthread_mutex_unlock(&mutex)`**
    → Libera el *mutex*. Si hay hilos en espera por el *mutex*, uno de ellos lo adquiere y se desbloquea.
  - **`pthread_mutex_trylock(&mutex)`**

  → Adquiere el *mutex* si está libre (en ese caso devuelve 0). Si estaba adquirido retorna error (EBUSY).
- Las operaciones garantizan ejecución atómica.

# CERROJOS: creación y destrucción

- Un cerrojo se debe inicializar antes de ser usado (de esta forma se establecen sus atributos). Hay dos formas de inicializar un cerrojo:
  - Estática: en el momento de declararlo

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

  - Dinámica: empleando **pthread_mutex_init()**

```
pthread_mutex_t mymutex;
…
pthread_mutex_init(&mymutex,NULL);
```

  - Destrucción

```
pthread_mutex_destroy(&mymutex);
```

# CERROJOS: CASO DE USO TÍPICO

```
pthread_mutex_t cerrojo;
pthread_mutex_init (&cerrojo, NULL);
// forma alternativa
pthread_mutex_t otro = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock (&cerrojo);
// acceso a un recurso en exclusión mutua
pthread_mutex_unlock (&cerrojo);

if ( pthread_mutex_trylock(&cerrojo) == 0 ) {
  // accede al recurso
  pthread_mutex_unlock (&cerrojo);
}
```

# VARIABLES CONDICIÓN

- Una variable condición sirve para gestionar una cola de espera por un recurso o una condición lógica.
- La v.c. está siempre asociada a un cerrojo.
- Semántica Mesa
- Operaciones:
  - **`pthread_cond_wait(&condition,&mutex)`**
    → bloquea al hilo y lo mete en la cola de la v.c. Mientras el hilo está bloqueado, se libera el *mutex*. Al despertar el hilo adquiere automáticamente el *mutex*.
  - **`pthread_cond_signal(&condition)`**
    → desbloquea a un hilo de la cola.
  - **`pthread_cond_broadcast(&condition)`**
    → desbloquea a todos los hilos de la cola.

# VARIABLES CONDICIÓN: inicialización y destrucción

- Una variable condición se debe inicializar antes de ser usada (de esta forma se establecen sus atributos). Hay dos formas de inicializar una variable condición:
  - Estática: en el momento de declararla

```
pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;
```

  - Dinámica: empleando **pthread_cond_init()**

```
pthread_cond_t myconvar;
…
pthread_cond_init(&myconvar,NULL);
```

  - Destrucción

```
pthread_cond_destroy(&myconvar);
```

# VARIABLES CONDICIÓN: CASO DE USO TÍPICO

```
pthread_mutex_lock(&mutex);
// espera condicional
while (! condición) {
  pthread_cond_wait(&cond, &mutex);
}
// … más acciones
pthread_mutex_unlock(&mutex);
```

```
pthread_mutex_lock(&mutex);

// …
// Se produce la condición

pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mutex);
```