SISTEMAS OPERATIVOS AVANZADOS

Universidad Tecnológica Nacional
Facultad Regional La Plata
Departamento Ingeniería en Sistemas de Información

2018

POSIX Threads

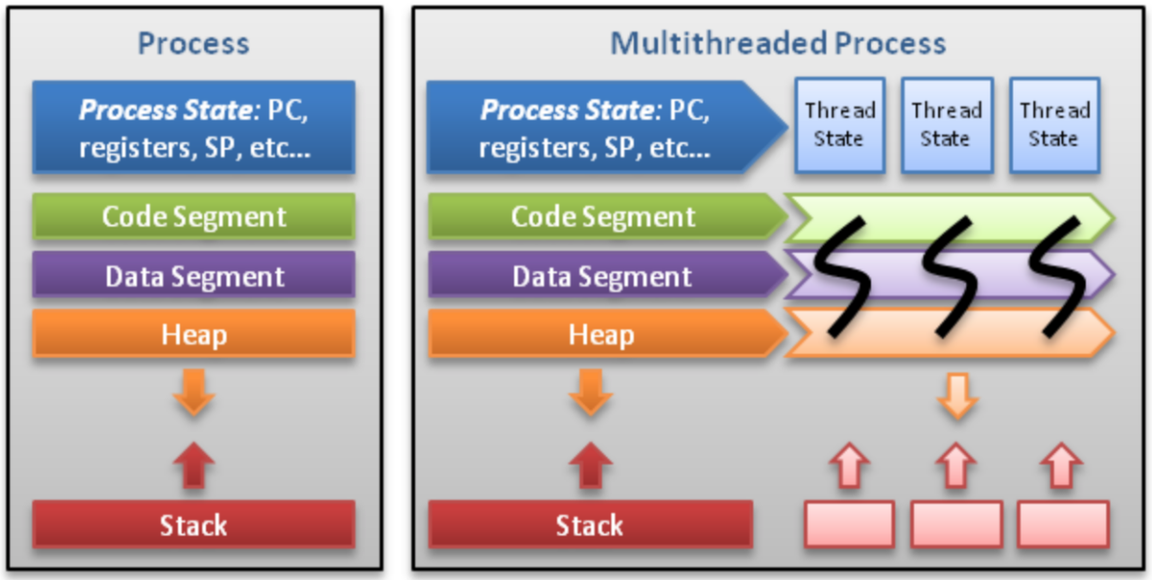
pthreads

¿Qué es un *thread*?

- Para definir un thread formalmente, primero debemos entender sus límites de operación.
- Un programa se convierte en un **proceso** cuando se carga en memoria y comienza la ejecución. Un proceso puede ser ejecutado por uno o muchos procesadores.
- Cada proceso guarda su *"descripción"* en memoria, contiene información vital tal como el contador de programa (que realiza un seguimiento de qué instrucción se está ejecutando actualmente), registros, variables, identificadores de archivo, señales, etc.
- Un **thread** es una secuencia de tales instrucciones dentro de un programa que se puede ejecutar independientemente de otro código.

¿Qué es un *thread*?

- En la figura se puede ver que los threads están dentro del **mismo espacio de direcciones de proceso**, por lo tanto, gran parte de la información se puede compartir entre hilos.



¿Qué es un *thread*?

- Cada thread contiene sólo la información necesaria: *stack* (pila para variables locales, argumentos de funciones y valores a devolver), copia de los registros, process counter y datos necesarios del proceso que les permita ser programados independientemente.
- Se requiere soporte explícito del sistema operativo para ejecutar programas multiproceso. Aunque la mayoría de los sistemas operativos modernos los admiten (Linux, variantes de BSD, Mac OS X, Windows, Solaris, etc).

1a_task.c

```
void task() {
  long i;
  double result = 0.0;

  for (i = 0; i < 10000000; i++) {
    result = result + sin(i) * cos(i) * tan(i);
  }
  printf("Task completed with result %e\n", result);
}

int main() {
  task();
}
```

lb_task.c

```c
#define NUM_ITER 5

void task(long id) {
  long i;
  double result = 0.0;

  for (i = 0; i < 10000000; i++) {
    result = result + sin(i) * cos(i) * tan(i);
  }
  printf("Task %ld completed with result %e\n", id, result);
}

int main() {
  long t;

  for (t = 0; t < NUM_ITER; t++) {
    task(t);
  }
}
```

¿Qué sucede con el tiempo de ejecución?

¿Cómo se podría mejorar?

POSIX Threads

- **POSIX threads**, o *pthread* es una librería estándar de C/C++ utilizada para generar un nuevo flujo de subprocesos (*threads*) concurrentes.
- **Multithreading** es una técnica que permite a un programa realizar múltiples tareas concurrentemente.
- Suele ser más efectivo en sistemas multiprocesador o multi-núcleo donde el flujo del proceso puede programarse para ejecutarse en otro procesador, ganando así velocidad a través del procesamiento en paralelo.
- Los threads requieren menos sobrecarga que operaciones *fork()* o nuevos procesos ya que el sistema operativo no inicializa un nuevo espacio de memoria ni entorno para el proceso.

POSIX Threads

- Si bien el uso de threads es más efectivo en un sistema multiprocesador, también se encuentran mejoras en sistemas con un solo procesador ya que aprovechan la latencia de E/S que pueden detener la ejecución del proceso, haciendo que **se puede ejecutar un subproceso mientras otro está esperando E/S o alguna otra acción**.

- Los threads se limitan a un único sistema informático, mientras que otras tecnologías de programacion en paralelo, como MPI, se utilizan en entornos distribuidos.

- Un thread no mantiene una lista de threads creados, ni conoce el thread que lo creó.

¿Cómo utilizar *pthreads*?

- Importar:

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

- Compilar:

```
sudo apt-get install gcc libc6-dev

gcc script.c -o script -lpthread
```

Primeras sentencias *pthread*

- Creación de thread:

```
pthread_create(
  pthread_t *thread, // objeto pthread_t
  pthread_attr_t *attr, // atributos del thread
  void *start_routine, // función que el thread ejecutará
  void *arg // argumentos de la función anterior
);
```

- Terminación de thread:

```
pthread_exit(
  void *retval // valor de retorno
);
```

lc_task.c

```
#define NUM_THREADS 5

void *task(void *t) {
  # stuff
  pthread_exit(0);
}

int main() {
  long t;
  pthread_t thread[NUM_THREADS];
  long rc;

  for (t = 0; t < NUM_THREADS; t++) {
    rc = pthread_create(&thread[t], NULL, task, (void *) t);
  }

  printf("Main completed\n");
  pthread_exit(0);
}
```

¿La ejecución está sincronizada?

Más sentencias *pthread*

- Unión de threads:

```
pthread_join(
  pthread_t *thread, // objeto pthread_t
  void **retval // puntero para retorno de pthread_exit()
);
```

Suspende al thread actual hasta la terminación del thread pasado como parámetro.

ld_task.c

```
#define NUM_THREADS 5

void *task(void *t) {
  # stuff
}

int main() {
  # stuff

  for (t = 0; t < NUM_THREADS; t++) {
    rc = pthread_create(&thread[t], NULL, task, (void *) t);
  }

  for (t = 0; t < NUM_THREADS; t++) {
    rc = pthread_join(thread[t], NULL);
  }

  printf("Main completed\n");
  pthread_exit(0);
}
```

Otro ejemplo: contador

Ahora haremos un programa que sume todos los números intermedios desde 1 hasta cierto número.

```
count(1) // → 1
count(2) // → 1 + 2 = 3
count(10) // → 1 + 2 + 3 + .... + 10 = 55
```

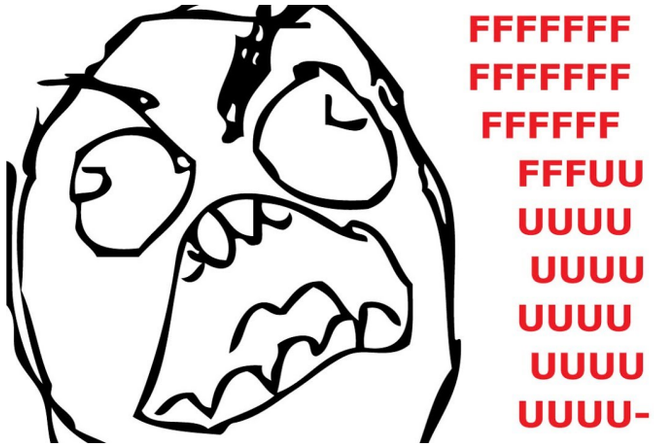
2a_counter.c

```
#define NUM_COUNTS 1000

long counter;

void count() {
  long i;

  for (i = 1; i <= NUM_COUNTS; i++) {
    counter += i;
  }
}

int main() {
  count();

  printf("Result = %ld\n", counter);
}
```

2b_counter.c

```
#define NUM_THREADS 1000
long counter = 0;

void *count(void *t) {
  long value = (long) t;
  counter += value; pthread_exit(0);
}

int main() {
  long t, rc; pthread_t thread[NUM_THREADS];

  for (t = 1; t <= NUM_THREADS; t++) {
    rc = pthread_create(&thread[t], NULL, count, (void *) t);
  }

  for (t = 0; t < NUM_THREADS; ++t) {
    pthread_join(thread[t], NULL);
  }

  printf("Result = %ld\n", counter); pthread_exit(0);
}
```

Otra vez problemas de sincronización...



Sección crítica

- Cuando dos threads intentan editar el mismo área de memoria *(sección crítica)* a la vez, en este caso la variable *counter*, pueden darse inconsistencias.
- Es recomendable sincronizar los threads para que escriban de a uno a la vez y evitar dichas *actualizaciones sobre valores fantasma*.
- Un ***mutex***, o mecanismo de **exclusión mutua** puede darnos la solución a dicho problema.

Exclusión mutua en *pthreads*

- Creación de mutex:

```
pthread_init(
  pthread_mutex_t *mutex, // objeto pthread_mutex_t
  const pthread_mutexattr_t *attr // atributos del mutex
);
```

- *Lock* de mutex:

```
pthread_mutex_lock(pthread_mutex_t *mutex);
```

- *Unlock* de mutex:

```
pthread_mutex_unlock(pthread_mutex_t *mutex);
```

2c_counter.c

```c
#define NUM_THREADS 1000
long counter = 0;
pthread_mutex_t counter_lock;

void *count(void *t) {
  long value = (long) t;

  pthread_mutex_lock(&counter_lock);
  counter += value;
  pthread_mutex_unlock(&counter_lock);

  pthread_exit(0);
}

int main() {
  pthread_mutex_init(&counter_lock, NULL);

  // ...
  printf("Result = %ld\n", counter); pthread_exit(0);
}
```

Otro ejemplo: contador condicional

Ahora haremos un contador que utilice dos funciones distintas para contar:

- *functionCount1()* contará los números entre 1-3 y 8-10.
- *functionCount2()* contará los números entre 4-7.

Es decir: **1 2 3 4 5 6 7 8 9 10**

Variables de condición

- Las variables de condición son otra herramienta utilizada para sincronizar threads a partir de un valor específico. Normalmente se utilizan como un sistema de notificación entre subprocesos.
- Cuando se espera en las variables de condición, la espera debe estar dentro de un bucle, no en una instrucción *if* para mantener la escucha activa. El thread estará esperando perpetuamente una señal.
- El mecanismo de la variable de condición permite que los hilos suspendan su ejecución y liberen al procesador hasta que alguna condición sea verdadera.
- Siempre debe asociarse con un *mutex* para evitar interbloqueos.

Variables de condición en *pthreads*

- Creación de variable de condición:

```
pthread_cond_init(
  pthread_cond_t *cond, // objeto pthread_cond_t
  pthread_condattr_t *attr // atributos de la variable de condición
);
```

- Espera de la condición:

```
pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mut);
```

- Envío de señal *(terminar espera)*:

```
pthread_cond_signal(pthread_cond_t *cond);
pthread_cond_broadcast(pthread_cond_t *cond);
```

3a_conditional-counter.c

```
pthread_cond_t condition_var;

void *functionCount1() {
  while (counter < COUNTER_DONE) {
    pthread_cond_wait(&counter_lock);
    counter++;
  }
}

void *functionCount2() {
  while (counter < COUNTER_DONE) {
    if (counter < COUNTER_HALT1 || counter > COUNTER_HALT2) {
      pthread_cond_signal(&condition_var);
    } else {
      counter++;
    }
  }
}

pthread_cond_init(&condition_var, NULL);
```

Otro ejemplo: contador doble

El próximo ejemplo será para mantener dos contadores: *counter_1* y *counter_2*.

A su vez habrá dos funciones que deberémos ejecutar: *functionInc()* la cual deberá incrementar 10 a cada contador; y *functionDesc()* la cual deberá decrementar 10 a cada contador.

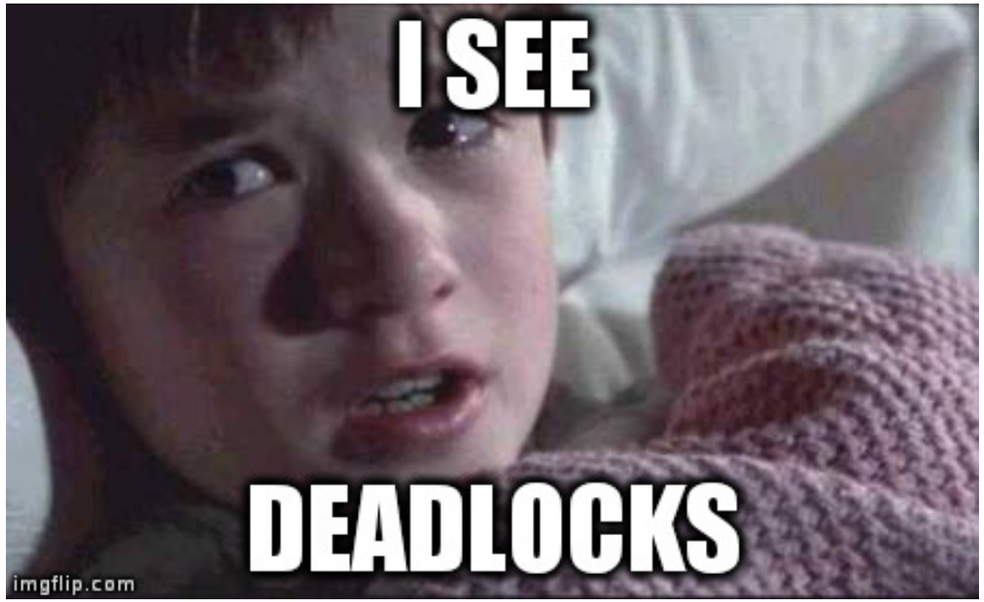
- Se deberá utilizar *pthreads* para ejecutar cada función en un thread diferente.
- Valores finales esperados: *counter_1 = 0* y *counter_2 = 0*

¿Cómo resolvería dicho ejemplo utilizando?


LET THE GAME BEGIN
memegen.com

4a_double-counter.c

```
pthread_mutex_t counter_1_lock;
pthread_mutex_t counter_2_lock;

void *functionInc() { ... }

void *functionDesc() { ... }

int main() {
  pthread_t thread1, thread2;

   pthread_mutex_init(&counter_1_lock, NULL);
   pthread_mutex_init(&counter_2_lock, NULL);

   pthread_create(&thread1, NULL, functionInc, NULL);
   pthread_create(&thread2, NULL, functionDec, NULL);

  // ...
}
```


I SEE
DEADLOCKS
imgflip.com

Más sentencias *pthread*

- *Lock* de mutex condicional:

```
pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Dicha función se utiliza para intentar bloquear un *mutex* y, en caso de ya estar bloqueado devolverá el código de error.

Útil para prevenir condiciones de interbloqueo.

4b_double-counter.c

```
pthread_mutex_t counter_1_lock;
pthread_mutex_t counter_2_lock;

void *functionInc() { ... }

void *functionDesc() {
  pthread_mutex_lock(&counter_2_lock);

  while (pthread_mutex_trylock(&counter_1_lock)) {
    pthread_mutex_unlock(&counter_2_lock);
    // en este instante counter_2_lock podrá ser adquirido por la otra fun
    pthread_mutex_lock(&counter_2_lock);
  }
}

int main() {
  // ...
}
```

Recomendaciones finales y errores frecuentes

- Condiciones de carrera: si bien el código puede aparecer en la pantalla en el orden en que desea que se ejecute el código, los *threads* son programados por el sistema operativo y se ejecutan al azar.
- No se puede suponer que los hilos se ejecutan en el orden en que se crearon. Cuando los hilos se están ejecutando (carreras para completar) pueden dar resultados inesperados (condición de carrera).
- Los mutexes y los *joins* se deben utilizar para lograr un orden de ejecución y un resultado predecibles.

Recomendaciones finales y errores frecuentes

- Thread safe code: las rutinas ejecutadas en cada *thread* deben llamar a funciones que son "seguras para ejecutar concurrentemente". Esto significa que no hay variables estáticas o globales que otros threads puedan tachar o leer asumiendo una operación de un solo *thread*.
- Si se utilizan variables estáticas o globales, se deben aplicar mutexes o las funciones se deben volver a escribir para evitar el uso de estas variables.
- Las funciones inseguras se pueden usar solo con un *thread* a la vez, y se debe garantizar la exclusividad del mismo.
- Cuidado con los interbloqueos!!!

Más de *pthreads*

- *https://en.wikipedia.org/wiki/POSIX_Threads*
- *http://man7.org/linux/man-pages/man7/pthreads.7.html*
- *http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html*
- *https://randu.org/tutorials/threads/*
- *https://www.cs.nmsu.edu/~jcook/Tools/pthreads/pthreads.html*
- *https://computing.llnl.gov/tutorials/pthreads/*

Módulo *threading*

threads en *Python*

Python: módulo *threading*

- El módulo *threading* aparece en la versión 1.5.2 de Python como una mejora al módulo *thread* usado para manejo de hilos a bajo nivel.
- Este nuevo módulo hace que el manejo se *threads* sea más transparente y fácil de llevar a cabo, a partir de la utilización de métodos de alto nivel.

Python: módulo *threading*

Contiene todos los métodos del módulo *thread* y además incluye otros métodos:

- **threading.activeCount()** devuelve la cantidad de *threads* activos
- **threading.currentThread()** devuelve el número del *thread* actual
- **threading.enumerate()** devuelve una lista de *threads* activos

La clase *Thread*

Además de los métodos anteriores, el módulo *threading* contiene la clase **Thread** la cual implementa el manejo de cada *thread* independiente. Tiene los siguientes métodos:

- **run()** es la función que le dice qué hacer al *thread*
- **start()** comienza el *thread* llamando al método *run()*
- **join([time])** espera a otros *threads* para terminar
- **isAlive()** verifica si el *thread* aún se está ejecutando
- **getName()** devuelve el nombre de un *thread*
- **setName()** define el nombre de un *thread*

Creación de *threads*

- 1) Definir una subclase de la clase *Thread*
- 2) Reescribir el método *__init__()* para agregar argumentos adicionales
- 3) Reescribir el método *run()* con la función que quiera que ejecute el *thread*

Una vez que esté creada la subclase, se puede crear una instancia de la misma y comenzar el *thread* ejecutando el método *start()* del mismo.

1_threading.py

```
# ver fichero
```

Sincronización de *threads*

- Como *pthreads*, *threading* también ofrece mecanismos para sincronizar *threads*. Para crear un nuevo *"lock"* (cerrojo) simplemente hay que invocar al método *Lock()*, la cual devuelve un objeto lock.
- El método ***acquire(blocking)*** del objeto sirve para forzar que los *threads* se ejecuten sincronizados. Su parámetro *blocking* sirve para indicar si el lock es bloqueante o no.
- El método ***release()*** es utilizado para liberar el lock.

2_threading.py

```
# ver fichero
```

Prioridad

El módulo *Queue* permite crear una cola de ejecución a partir de los siguientes métodos:

- **get()** toma de la cola y devuelve un item
- **put()** agrega a la cola un item
- **qsize()** devuelve la cantidad de items que contiene la cola
- **empty()** devuelve *True* si la cola está vacía, sino *False*.
- **full()** devuelve *True* si la cola está completa, sino *False*

3_threading.py

```
# ver fichero
```

Otros componentes de *threading*

El módulo *threading* también tiene soporte para:

- **Semaphore** para manejar contadores internos que irán decrementando cada vez que sean adquiridos e incrementados al ser liberados.
- **Event** para comunicación entre *threads* utilizando señales.
- **Barrier** primitiva para sincronizar *threads* dentro de un *pool* de ejecución.

4_threading.py

```
# ver fichero
```

Fin ✨