PROGRAMACIÓN MULTITHREADING

Sergio Nesmachnow (sergion@fing.edu.uy)
Gerardo Ares (gares@fing.edu.uy)

Escuela de Computación de Alto Rendimiento
(ECAR 2012)

TEMA 3: PROGRAMACIÓN AVANZADA

PROGRAMACIÓN MULTITHREADING

Escuela de Computación de Alto Rendimiento (ECAR 2012)

CONTENIDO

- Thread ID.
- Estrategias de sincronización.
- Reentrancia.
- Balance de carga.
- Pool de threads.
- Cancelación de threads.
- Señales.

PROGRAMACIÓN PARALELA

THREAD ID

THREAD ID

- A veces es necesario obtener que thread está ejecutando cierta parte de un código.
- El thread id es obtenido a través de la función `pthread_self` en la función que lo invoca.
- El thread id puede ser comparado con otro thread id con la función `pthread_equal`.
- Los prototipos de las funcione son los siguientes:

```
pthread_t pthread_self(void)
int pthread_equal(pthread_t tid1, pthread_t tid2)
```

- Esto permite por ejemplo controlar de antemano la posibilidad de hacer un `pthread_join` sobre uno mismo.

PROGRAMACIÓN PARALELA

ESTRATEGIAS DE SINCRONIZACIÓN

LOCKS GLOBALES

- La primer estrategia de sincronización es la de asignar un único lock global.
- Todos los threads “serializan” su ejecución a través de un único lock global.
- La estructura de cada thread es la siguiente:

```
void reader(void *){
   pthread_mutex_lock(&lock);
   …
   pthread_mutex_unlock(&lock);
}
```

- Ventajas:
  - Mecanismo simple de sincronización.
- Desventajas:
  - No se aprovecha los equipos multiprocesadores.

LOCKS ESTRUCTURADOS EN CÓDIGO

- **Locks estructurados en código.**
  - **Un conjunto de datos es accedido a través de un conjunto de funciones que pertenecen a un módulo.**
  - **Las funciones se sincronizan el acceso a los datos compartidos a través de un único mutex definido a nivel global en el módulo.**

```
struct element {
   ...
};
struct listFIFO {
   ...
};
pthread_mutex_t listLOCK= PTHREAD_MUTEX_INITIALIZER;
// Funciones exportadas
int put(struct listFIFO *lf, data d) {
   pthread_mutex_lock(&listLOCK);
   addTail(lf,consElement(d));
   pthread_mutex_unlock(&listLOCK);
}
```

LOCKS ESTRUCTURADOS EN DATOS

- **Locks estructurados en datos.**
  - **Los datos se pueden agrupar en diferentes conjuntos de datos independientes.**
  - **Se asocia el elemento de sincronización al objeto y no al módulo.**

```
struct listFIFO {
    pthread_mutex_t lock;
    pthread_cond_t cond;
    ...
};
// Funciones externas
int put(struct listFIFO *lf, data d) {
    ...
    pthread_mutex_lock(&lf->lock);
    addTail(lf,consElement(d));
    pthread_mutex_unlock(&lf->lock);
    pthread_mutex_signal(&lf->cond);
    ...
}
```

PROGRAMACIÓN PARALELA

REENTRANCIA

REENTRANCIA

- En la programación con threads es importante utilizar funciones reentrantes.
- Una función es reentrante cuando su código puede ser ejecutado en paralelo por al menos dos thread de un mismo proceso y su resultado es el mismo que si la función se hubiera ejecutado en forma serial.
- Las variables de tipo static o globales deben ser tratadas con especial cuidado en las funciones.

```
long esPrimo(long numero) {
 long i;
 static long m;
 ...
```

```
static long m;
long esPrimo(long numero) {
  long i;
  ...
```

PROGRAMACIÓN PARALELA

BALANCE DE CARGA

BALANCE DE CARGA

- A veces algunas estrategias de división de dominio generan desbalance en las tareas.
- Por ejemplo si para el cálculo de la cantidad de números primos menores a un número dado, se divide el dominio en dos tareas donde una toma la primera mitad y la segunda toma la segunda mitad.
- Generalmente la primer mitad terminará antes debido a que tiene menos cómputo a realizar.
- Esa división generará que ciertos recursos queden ociosos y que los tiempos de ejecución dependan de la tarea que tiene más carga.
- Es necesario entonces re-diseñar la solución de forma de balancear la carga para que las tareas terminen lo más próximo posible de forma de minimizar los tiempos ociosos de los procesadores.

BALANCE DE CARGA

- En el problema planteado, una mejor alternativa es que cada tarea no tome una mitad, sino que se reporte para calcular si un número es primo o no.
- Eso lo hacen de forma sucesiva cada una hasta que logren evaluar todos los números menores o iguales que el número pedido.
- La nueva distribución permitirá un mejor balance y, por lo tanto, un mejor desempeño de la aplicación.

PROGRAMACIÓN PARALELA

POOL DE THREADS

POOL DE THREADs

- La creación y destrucción repetida de threads desperdicia ciclos de procesamiento para la aplicación (overhead).
- Por ejemplo, en el modelo maestro/esclavo donde los esclavos son creados en cada iteración.
- Una solución para no perder ciclos de procesador en cada iteración creando y destruyendo threads es la utilización de un Pool de Threads:
  - Se crea un conjunto de threads los cuales esperan en una variable de condición por una señal.
  - Cuando la señal es recibida comienza su procesamiento.
  - Al finalizar la tarea vuelven a esperar en una variable de condición a la espera de una señal.
- Otra solución es el uso de barriers.
- De esta forma se evita estar creando y destruyendo continuamente threads.

LOCKS ESTRUCTURADOS EN DATOS

```
while (1) {
   pthread_mutex_lock(&mtx);
   pthread_cond_wait(&cond,&mtx);
   if (salir) {
     pthread_mutex_unlock(&mtx);
     return NULL;
   }
   pthread_mutex_unlock(&mtx);
    ...
```

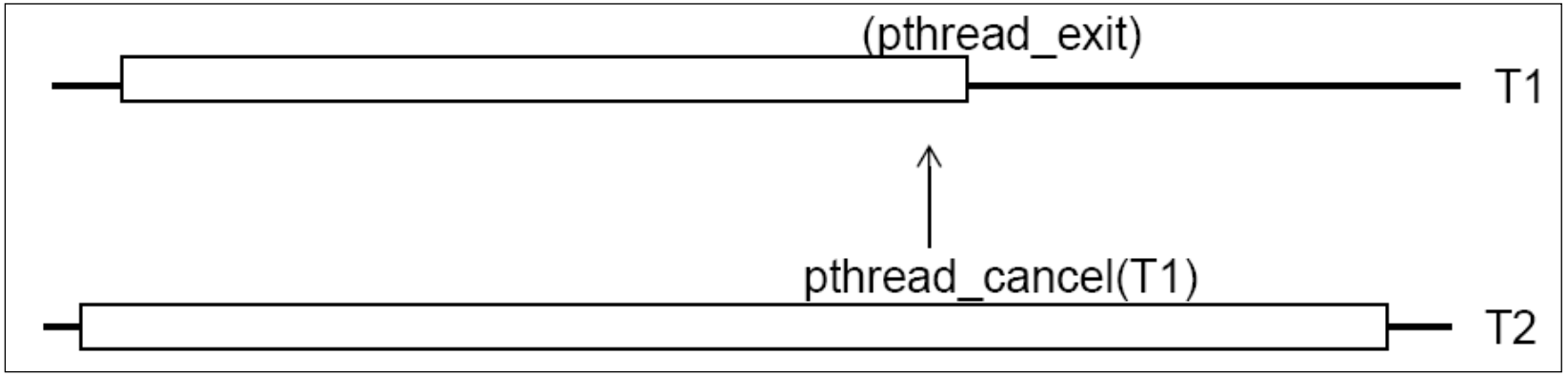
```
while (1) {
   pthread_barrier_wait(&barrier);
   pthread_mutex_lock(&cond,&mtx);
   if (salir) {
     pthread_mutex_unlock(&mtx);
     return NULL;
   }
   pthread_mutex_unlock(&mtx);
   ...
```

PROGRAMACIÓN PARALELA

CANCELACIÓN DE THREADS

CANCELACIÓN DE THREADs

- **En ciertas circunstancias es necesario finalizar un thread que está ejecutando.**



- **La función `pthread_cancel` permite que un thread cancele otro en la mitad de su ejecución.**
- **Si bien parece una tarea sencilla, no lo es.**

CANCELACIÓN DE THREADs

- El thread tiene un estado privado que permite ser cancelado o no.
- Antes de entrar en una sección crítica se puede deshabilitar la cancelación y luego la habilita nuevamente al salir.
- El estado es configurado a través de la función: `pthread_setcancelstate`.
- Se define un tipo de cancelación.
  - Asincrónica (asynchronous).
    - El thread es cancelado inmediatamente.
  - Diferida (deferred) – Defecto.
    - El thread es cancelado cuando consulta si debe fue cancelado por otro thread utilizando la primitiva: `pthread_test_cancel` .
    - También se verifica en funciones de biblioteca que sea un punto de cancelación (cancellation point). Ej: open,read, write, pause, waitpid.

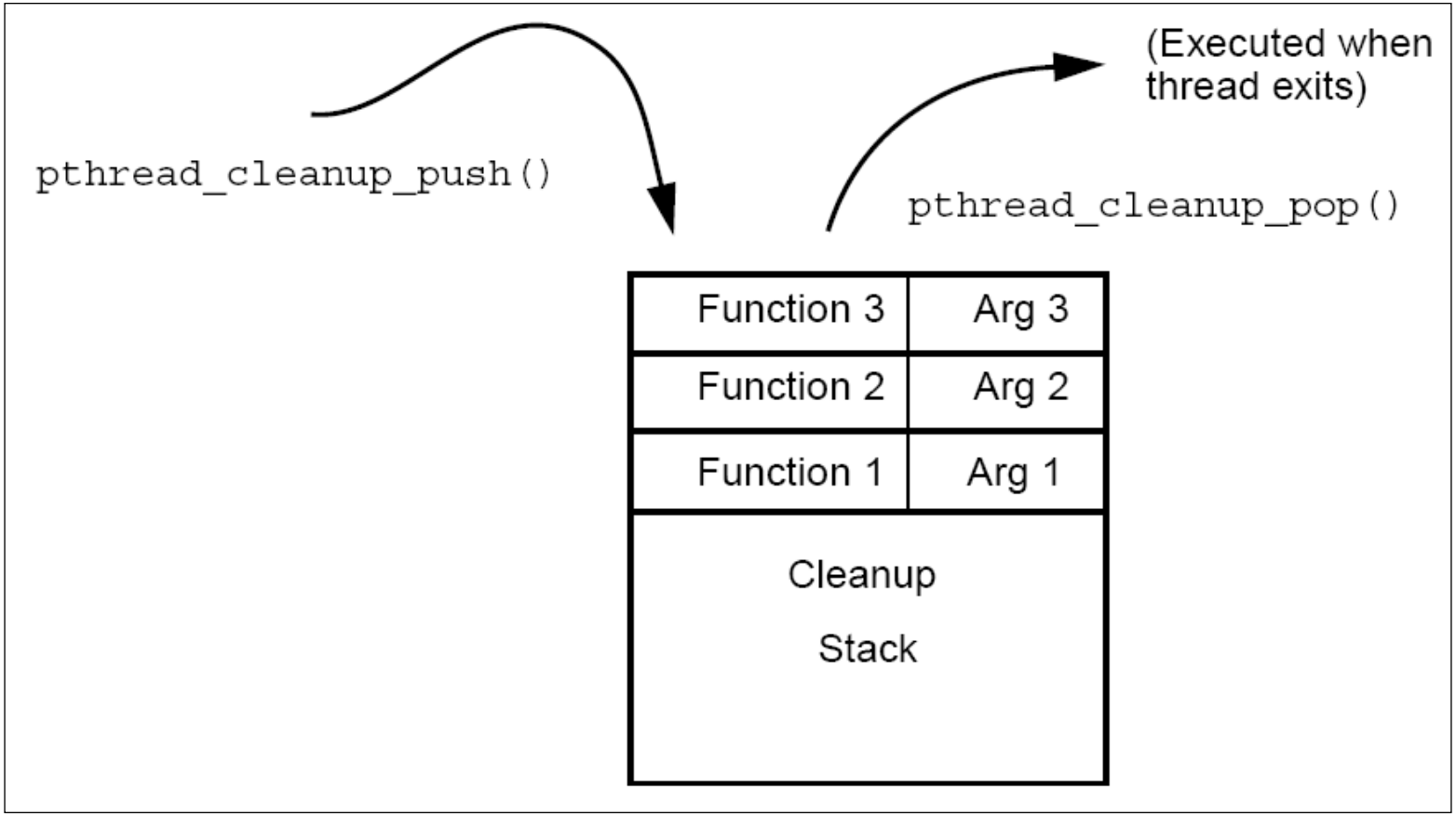
CANCELACIÓN DE THREADs

- La tarea parece sencilla, pero no lo es.
- Problemas:
  - Que pasa con los recursos que tiene ganados.

    Ej: mutex ?.
  - Memoria pedida a través de la función `malloc`.
- Solución: Handlers de limpieza en cancelaciones.
  - Se provee de una estructura LIFO que permite configurar funciones de limpieza del thread antes de finalizar su ejecución.
  - Las funciones de limpieza reciben un parámetro de tipo `void *`.
- Handlers de limpieza:
  - `pthread_cleanup_push(void *).`
  - `pthread_cleanup_pop(void *).`

CANCELACIÓN DE THREADs

- **Handlers de cancelación.**



PROGRAMACIÓN PARALELA

SEÑALES

SEÑALES

- Asignación de señales al igual que a un proceso tradicional.
- Cada thread cuenta con una mascara que le permite filtrar ciertas señales.
- Existe herencia: Un thread creado por otro hereda la mascara de señales.
- Las señales de cada threads son manipuladas a través de la siguiente función:

```
pthread_sigmask(int how, const sigset_t *new, sigset_t *old);
pthread_sigmask(SIG_SETMASK, &new, &old); /*set new mask */
pthread_sigmask(SIG_BLOCK, &new, &old); /* blocking mask */
pthread_sigmask(SIG_UNBLOCK, &new, &old); /* unblocking */
```

- Si el proceso recibe una señal, se ejecutara la rutina (handler) para todos los threads que no la tengan deshabilitada.
- Se permite generar una señal para un thread específico a través de la primitiva:

```
pthread_kill(pthread_t, int signal)
```