Auxiliar 2

Sincronización de Threads

Profesores: Rodrigo Arenas, Luis Mateu y Lucas Torrealba
Auxiliares: José Astorga, Vicente González y Pablo Jaramillo
Semestre: Otoño 2024

Resumen

Pthreads

- Iniciar un thread

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

  La cual **retorna** 0 si la creación fue exitosa y donde:
  - `thread`: Es donde queda guardado el thread.
  - `attr`: Son atributos especiales para crear el thread.
  - `start_routine`: Es la función a ejecutar por el thread, la cual **recibe** un único puntero genérico (`void *`) como argumento.
  - `arg`: Argumento pasado a la función.

- Enterrar un thread

```c
int pthread_join(pthread_t thread, void **return_value);
```

  La cual **retorna** 0 en caso de éxito.

  IMPORTANTE: Todo thread debe ser enterrado, si no queda en estado de zombie y no liberará sus recursos.

- Pasar argumentos al thread

```c
// Ejemplo de valores
typedef struct {
  double x;
  char *s;
  int n;
  unsigned res;
} Args;
```

  En general, es mejor guardar el resultado de la operación dentro de la misma estructura en vez de usar el retorno de la función.

Estrategia general

1. Descubrir / diseñar qué parte del algoritmo podemos paralelizar efectivamente.
2. Lanzar threads con argumentos correspondientes.
3. Esperar que el trabajo paralelo sea realizado (Quizás es necesario realizar trabajo en el thread principal).
4. Enterrar los threads lanzados y recolectar los resultados.

Mutex

- Características:
  - Es una herramienta de EXclusión MUTua.
  - Evita que dos o más threads entren a una zona crítica a la vez.
  - Tiene dos estados: abierto o cerrado.
  - Un proceso que solicite un mutex cerrado, queda esperando a que se abra.
  - Evita problemas de hambruna, *datarraces* y *race conditions*.
- Iniciar un mutex

```c
// Global
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

// Local
pthread_mutex_t mutex;
void fun(...) {
  ...
  pthread_mutex_init(&mutex, NULL);
  ...
}
```

- Solicitar un mutex (dentro de la función del thread):

```c
int phtread_mutex_lock(phtread_mutex_t *mutex);
```

  Y luego para liberarlo:

```c
int phtread_mutex_unlock(phtread_mutex_t *mutex);
```

Condiciones

- Características:
  - Se utiliza en conjunto a un mutex dentro de la zona crítica.
  - Permite que un thread quede esperando una condición.
  - Al esperar que una condición se cumpla, libera el mutex; al despertar, espera a que el mutex este disponible.
  - Evita el *busy-waiting*.
- Iniciar una condición:

```c
// Global
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

// Local
pthread_cond_t mutex;
void fun(...) {
  ...
  pthread_cond_init(&cond, NULL);
  ...
}
```

- Esperar la condición:

  ```c
  int phtread_cond_wait(pthread_cond_t *cond, phtread_mutex_t *mutex)
  ```

  Luego, en otro thread, se «despierta» a todos los que esperan con:

  ```c
  int phtread_cond_broadcast(pthread_cond_t *cond)
  ```

  Para despertar a uno sólo:

  ```c
  int phtread_cond_signal(pthread_cond_t *cond)
  ```

Preguntas

P1. Quicksort en N cores

La función de abajo es una implementación del algoritmo de *quicksort* para ordenar un arreglo de enteros.

```
void quicksort_seq(int a[], int i, int j){
  if (i < j) {
    int h = particionar(a, i, j);
    quicksort_seq(a, i, h-1);
    quicksort_seq(a, h+1, j);
  }
}
```

Considere que usted tiene a su disposición la función `particionar`, la cual se encarga de seleccionar un elemento del arreglo como "pivote", dejando a su lado izquierdo los valores menores y a su lado derecho los valores mayores. La función retorna la posición final en la que se encuentra el "pivote".

Usted debería paralelizar la función `quicksort` para una máquina de multi-core, siendo el encabezado de la función el siguiente:

```
void quicksort(int a[], int i, int j, int n);
```

P2. Colecta

Se necesita crear un sistema para juntar exactamente una cantidad X de dinero:

(a) Definir el tipo de datos `Colecta`

(b) Programar la función `Colecta *nuevaColecta(double meta)` que crea y retorna una colecta para juntar **$meta**.

(c) Programar la función `double aportar(Colecta *c, double monto)`, que es invocada desde múltiples threads para contribuir con **$monto**. El valor de retorno de la función es el mínimo entre monto y lo que falta para llegar a la meta. **La función debe retornar una vez que la meta se cumpla.**

P3. (Propuesto) PSS C2 2018-P

La función *imprimir* sirve para imprimir documentos desde múltiples threads. Evita que 2 threads accedan a la impresora simultáneamente. Recibe el documento que se debe imprimir y un entero entre 0 y 9 que representa la prioridad. La siguiente es una implementación incompleta de esta función porque no considera la prioridad.

```
pthread_mutex_t m= PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond= PTHREAD_COND_INITIALIZER;
int ocup= 0;
void imprimir(Doc *doc, int pri) {
  lock(&m);
```

```
  while (ocup)
    wait(&cond, &m);
  ocup= 1;
  unlock(&m);
  doPrint(doc); // imprime de verdad
  lock(&m);
  ocup= 0;
  broadcast(&cond);
  unlock(&m);
}
```

Complete esta implementación de modo que una impresión con prioridad p no pueda comenzar mientras exista una impresión pendiente con prioridad $q > p$.

Ayuda: Necesitará usar un arreglo global que contabilice las impresiones pendientes para cada prioridad. Programe una función que dada una prioridad p retorne verdadero si existen impresiones pendientes con mayor prioridad, o falso en caso contrario. Impresiones con la misma prioridad pueden realizarse en cualquier orden.