Sistemas Operativos

Primer Cuatrimestre de 2010

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico N°2

pthreads

Grupo

Integrante	LU	Correo electrónico
Mataloni Alejandro	706/07	amataloni@gmail.com
Mancuso Emiliano	597/07	emiliano.mancuso@gmail.com

1. Ejercicio 1

Para empezar, creamos un archivo nuevo **servidor_multi.c** con el fin de desarrollar el servidor multi-thread para realizar el TP, partiendo del código original del **servidor_mono.c**

Primero leímos (desde internet) como usar *pthreads* en nuestra application. Analizamos el código y decidimos que con sólo hacer la función atendedor_de_alumno multi-thread, podríamos realizar el procesamiento de varios clientes simultaneamente.

Como la función **atendedor_de_alumno** tenía la siguiente aridad:

```
void *atendedor_de_alumno(int socket_fd, t_aula *el_aula)
```

Lo que hicimos fue crear un struct para pasarle los parámetros:

```
typedef struct {
        int socket_fd;
        t_aula *el_aula;
} t_arguments;
```

Luego para crear el thread:

```
rc = pthread_create(&threadId, NULL, atendedor_de_alumno, (void *) &arguments);
```

Una vez resuelto el tema de los threads, pasamos al control de acceso a las variables compartidas.
En principio tenemos que controlar aquellas variables que son accedidas desde todos los threads. Si no se hiciera esto podríamos caer en una *race condition*.

Para solucionar esto utilizamos **mutexes**
Estas variables son:

1. un_aula
2. rescatistas_disponibles

Asignamos el *mutexAula* para garantizar el acceso de a un thread a la vez, para cuando se consultan o modifican las posiciones o la cantidad de personas de la misma. El *mutexRescatista* se encarga de sincronizar el acceso a la cantidad de rescatistas disponibles. Cuando se solicitan más rescatistas que los disponibles, el alumno que lo requiere se queda esperando.
Para no hacer **busy wait** (e irnos al infierno que nos conto Fernando Schapachnic) utilizamos las variables de condición. La variable de condición encargada de sincronizar la cantidad de rescatistas disponibles es *rescatista_cv* y nos permite mandar a esperar a los threads que todavía no dispongan de un rescatista. Cuando la liberación de un alumno ocurre, se libera un rescatista y al mismo momento se produce una **signal** que *despierta* algún thread en espera.

2. Pruebas realizadas

Para poder probar que los rescatistas liberaban a los alumnos concurrentemente, creamos un script que lanzaba 10 instancias distintas del *server_tester* que nos facilitó la cátedra con 20 alumnos cada instancia.

Como configuración inicial, elegimos tener 2 rescatistas para poder controlar facilmente el comportamiento del servidor multithread. Además introdujimos una espera en los rescatistas (con la funcion **usleep**) para demorarlos y poder simular el comportamiento en paralelo, dado que las operaciones a realizar demoraban muy poco tiempo.

Por último levantamos el servidor multithread en una pc, y con otras 2 computadoras corrimos el mismo script, insertando alumnos en el servidor, para que la concurrencia no se vea afectada por el *scheduler* del sistema operativo del servidor. De todas formas, no pudímos observar diferencias dado que los threads eran de corta duración por las operaciones a realizar.

3. Dificultades encontradas

1. Cuando empezamos a hacer pruebas, en algunos casos nos fallaba al enviar los primeros datos (nombre, columna y fila). Luego de observar y discutir en el laboratorio con el resto de los alumnos nos dimos cuenta que habia un parametro pasado a la función *sscanf*, dentro de *biblioteca.c*, el cual hacía que se 'rompa' el programa. El problema era que se limitaba el nombre del país a 20 caracteres lo cual a veces no se cumplía. La solución fue cambiar ese parametro por un número mayor donde la entrada (dada por el archivo **paises.py**) coincida.

2. Otra dificultad que tuvimos fue que en un principio en la función **main** creabamos un solo struct para pasar los parámetros a la función de thread por lo tanto era una única variable que era utilizada por todos los thread (sólo en el principio pero podía traer complicaciones). La solución fue que antes de crear un thread hacemos una llamada a **malloc** para reservar (y no alocar) memoria para una nueva variable de ese tipo y utilizarla para crear el nuevo thread.