DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Sistemas Operativos
Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Caravario, Martín	470/12	martin.caravario@gmail.com
Hosen, Federico	825/12	fhosen@hotmail.com
Vuotto, Lucas	385/12	lvuotto@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

El objetivo de este trabajo práctico es implementar un nuevo diseño de una simulación de simulacro de evacuación.

Se desea pasar de un modelo *mono-thread* a uno *multi-thread*, para poder simular que varios alumnos se muevan por el aula simultaneamente.

Para esta nueva implementación se pide utilizar la biblioteca *Pthreads*, restringiéndonos únicamente a los mutexes y variables de condición provistas por ésta.

En este informe detallaremos la implmentación realizada y justificaremos las decisiones que fuimos tomando a lo largo de la resolución del trabajo práctico.

Dicha explicación y justificación estarán en la sección que se encuentra a continuación, en forma de un único texto.

2. Detalles de implementación

La idea del nuevo diseño es tener un hilo principal que gestione las conexiones y delegue el trabajo de atender a un cliente en un nuevo thread. Para hacer ésto, el programa forkea, siendo el hijo el encargado de ser el hilo principal y el padre solamente espera a que el hijo termine.

Ésto lo hicimos para destruir todas las variables de tipo mutex y de condición que utilizamos para modelar al aula. El trabajo del padre es simplemente esperar a que el servidor termine y luego destruir al aula.

Es el hijo quien se encarga de ser el servidor. Ésto lo hace llamando a la función `servidor`.

Aquí se encuentra el primer cambio, se modularizaron las funcionalidades, estando ahora en la función *servidor* el código que antes (en el servidor mono) se encontraba en el *main*.

La forma de establecer la conexión con el cliente es la misma que antes, se sigue utilizando un socket de tipo *INET* con el protocolo *TCP*.

La diferencia en la gestión de las conexiones se ve en el ciclo. Una vez que la conexión con el cliente fue establecida, se inicializan los atributos necesarios y los argumentos para crear al *worker thread* que atenderá al cliente.

Para poder pasar los argumentos creamos una estructura de nombre *args* que tiene dos miembros, *fd* y *aula*, donde el primer corresponde al file descriptor del socket del cliente[1] en cuestión, y el segundo es un puntero a la estructura aula (la cual detallaremos más adelante).

Configuramos los atributos con la constante `PTHREAD_CREATE_DETACHED`. Ésto lo hicimos pues no es necesario en nuestra implementación hacer un *join* a un thread y al estar *detached* el thread libera automáticamente sus recursos, ahorrándonos de tener estructuras que guarden los *ID* de los threads y ahorrándonos también de tener algún tipo de estrategia de cuándo ó cómo liberar los recursos del thread.

Una vez que el thread sea creado, el servidor (hilo maestro) destruirá el atributo creado.

El thread empieza a correr con la función `start_routine`, que simplemente castea el puntero pasado como parámetro a la estructura `thread_args` para luego pasar los parámetros a la función `atendedor_de_alumno`.

Antes de analizar los cambios en la forma de atender a los alumnos, veamos cómo cambia la estructura `t_aula`.

En la versión *mono-thread* bastaba con saber cuántas personas estaban en cada posición del aula, la cantidad de personas totales y los rescatistas disponibles.

En nuestra implementación de la versión *multi-thread* tenemos lo siguiente:

```
typedef struct {
  int posiciones[ANCHO_AULA][ALTO_AULA];
  int cantidad_de_personas;
  int rescatistas_disponibles;
  int para_salir;
  bool saliendo;
  pthread_mutex_t m_posiciones[ANCHO_AULA][ALTO_AULA];
  pthread_mutex_t m_rescatistas;
  pthread_mutex_t m_estado;
  pthread_cond_t vc_rescatistas;
  pthread_cond_t vc_estado;
} t_aula;
```

Se mantienen los datos usados anteriormente, pero ahora necesitamos asegurarnos que no más de un alumno accederá a los recursos al mismo tiempo, pues esto generaría condiciones de carrera.

Es por esto que se agregan los distintos mutexes y variables de condición. Los usos son los siguientes:

- `m_posiciones[][]`, que cada posición i, j de la matriz tiene un mutex que regula el acceso a la posicion i, j de `posiciones`.
- `m_rescatistas`, que coordina el acceso a la variable `rescatistas_disponibles`.
- `m_estado`, que regula el acceso a las variables `cantidad_de_personas`, `para_salir` y `saliendo`.

[1] En términos de este trabajo práctico, nos referiremos a alumno y cliente indistintamente.

- `vc_rescatistas`, que sirve para simular el comportamiento de semáforos, tiene relacionada la variable `rescatistas_disponibles` y el mutex `m_rescatistas`.
- `vc_estado`, variable de condición que tiene relacionada la variable entera `para_salir` y el mutex `m_estado`.

Veamos ahora `atendedor_de_alumno`. La estructura de la función es similar a la de la versión del servidor *mono-thread*, es decir, la conexión se establece de la misma forma y los pasos que debe seguir el alumno para salir del aula son los mismos.

Lo que sí cambia es la forma de realizar las acciones, pues hay que soportar la concurrencia de varios alumnos en un mismo aula.

La primera modificación en las acciones del alumno se da en `t_aula_ingresar`. Como ésta modifica la cantidad de personas totale, y la cantidad de alumnos en una posición, se debe garantizar acceso único a dichas variables. Por eso para incrementar la cantidad de personas totales se pide por el mutex `m_estado`,y para aumentar la cantidad de personas en una posición se pide por `m_posiciones[i][j]`, siendo i, j la posición inicial del alumno.

Para recibir e interpretar el mensaje del cliente y para contestarle se siguen usando las funciones provistas por `biblioteca.h`.

Una modificación importante se da al momento de intentar moverse. El algoritmo es (conceptualmente) el mismo, lo que cambia es que ahora se necesita pedir por dos recursos compartidos por todos los alumnos. Éstos son, la posición en la cual está parado el alumno, llamemosla i, j, y la posición a la cual se quiere mover, llamemosla k, l.

Como debemos asegurarnos que ambas posiciones se acceden y actualizan al mismo tiempo, tenemos que pedir los mutex de ambas posiciones antes de cambiar alguna de las dos.

Esta acción podría generar deadlock si no se establece algún criterio en el orden de los pedidos. Por este motivo, establecimos un orden para pedir los mutexes, eliminando así la espera circular.

El orden consiste en primero pedir la posición *más chica*, considerando a las posiciones con una relación de orden basada primero en la *fila* y después en la *columna*. Es decir,

$$(i, j) < (k, l) \iff i < k \vee (i = k \wedge j < l)$$

De este modo, se pide por `m_posiciones[i][j]` primero y luego por `m_posiciones[k][l]` si $(i, j) < (k, l)$ y si no se piden al revés.

Una vez que el alumno consigue salir del aula tiene que ponerse la máscara y luego esperar a formar un grupo para efectivamente ser evacuado.

Ésta primera acción ya requiere de una coordinación de recursos más sofisticada. En la versión *mono-thread*, al haber sólo un alumno, siempre había un rescatista disponible; en cambio, ahora podría pasar que un alumno tenga que esperar a que un rescatista se libere.

Aquí es donde usaremos la variable de condición `vc_rescatistas`, pues el alumno tendrá que esperar a que `rescatistas_disponibles` sea mayor a 0.

Para hacer ésto, utilizamos el esquema visto en clase. Es decir, pedimos por el mutex asociado, y con un `while` esperamos que `rescatistas_disponibles` sea mayor a 0, haciendo `pthread_cond_wait` sobre las variables mencionadas. Luego, una vez que tenemos acceso a los rescatistas y sabemos que hay al menos uno libre, decrementamos la cantidad y liberamos el recurso. Se procede a*poner la máscara* y pedimos otra vez por el recurso para incrementar la cantidad de rescatistas disponibles, liberandolos finalmente.

Puesta la máscara, ahora falta hacer que los alumnos salgan formando un grupo de 5, o un grupo menor si no hay suficientes alumnos.

Entonces, básicamente tenemos que implementar una barrera, para que cuando se alcance un valor determinado, todos los alumnos que tenían que salir salgan y los que no podían ponerse a formar grupo se queden esperando a que terminen de salir los demás.

Con ésto en mente, el booleano `saliendo` nos indicará si hay un grupo que efectivamente está saliendo o no.

Entonces el alumno primero espera a que el grupo que estaba saliendo (si lo hay) termine de salir, esto se hace verificando `saliendo` con un `while`, utilizando la variable de condición `vc_estado`.

Una vez que el alumno sabe que no hay un grupo saliendo, empieza a formar uno nuevo, actualizando las variables `cantidad_de_personas` y `para_salir`. Cada alumno verifica si es el quinto en anotarse o si es el último alumno del aula, si pasa cualquiera de las dos situaciones, prende la variable `saliendo`, para que los demás alumnos con intención de salir se queden esperando y para que los alumnos que estaban formando grupo sepan que pueden empezar a salir.

Una vez que el alumno sale, decrementa `para_salir` y luego verifica si él era el último del grupo en salir. Si lo era, entonces ya no hay gente saliendo, con lo cual pone en falso la variable `saliendo`.

Por último, el alumno manda un `pthread_cond_broadcast` a la variable de condición `vc_estado`, despertanto a todos los alumnos y luego libera el mutex `m_estado`. Habilitando así a los demás alumnos del grupo a salir (si quedaba alguno) y habilitando a los que están esperando para formar un nuevo grupo que lo hagan.

Una vez hecho ésto el alumno ya se considera libre, con lo cual se le avisa y el *thread* retorna a la función `start_routine` para luego terminar retornando `NULL`.

3. Paralelismo

Para obtener el mayor grado de paralelismo y minimizar las estructuras bloquedas en exclusión mutua, creamos una matriz de mutexes, en la cual hay un mutex por cada metro cuadrado del aula, en contraposición a utilizar un mutex para el aula entera. De este modo, se pueden realizar múltiples movimientos de alumnos, siempre y cuando no coincidan las posiciones origen y destino de los distintos alumnos. Si en cambio se utilizase un solo mutex para *todo* el aula, no podrían realizarse **nunca** múltiples movimientos en simultáneo, pues habría que esperar a cada uno de los movimientos se complete, perdiendo así la mayoría del sentido de la implementación *multi-threading*.

4. Deadlock

Para analizar si el sistema se encuentra libre de deadlock veremos cada pieza de código que utiliza algún mecanismo de sincronización y mostraremos que dicho código está libre de deadlock.

```
void t_aula_ingresar(t_aula *un_aula, t_persona *alumno)
{
  pthread_mutex_lock(&un_aula->m_estado);
  un_aula->cantidad_de_personas++;
  pthread_mutex_unlock(&un_aula->m_estado);

  pthread_mutex_lock(&un_aula->m_posiciones[alumno->posicion_fila][alumno->posicion_columna]);
  un_aula->posiciones[alumno->posicion_fila][alumno->posicion_columna]++;
  pthread_mutex_unlock(&un_aula->m_posiciones[alumno->posicion_fila][alumno->posicion_columna]);
}
```

Ver que `t_aula_ingresar` está libre de deadlock es trivial, pues ni si quiera hay *hold and wait*, es decir, una vez adquirido el recurso, realiza las acciones necesarios y libera el recurso, sin pedir ningún otro recurso en el medio.

```
void t_aula_liberar(t_aula *un_aula, t_persona *alumno)
{
  pthread_mutex_lock(&un_aula->m_estado);
  un_aula->cantidad_de_personas--;
  pthread_mutex_unlock(&un_aula->m_estado);

  pthread_mutex_lock(&un_aula->m_posiciones[alumno->posicion_fila][alumno->posicion_columna]);
  un_aula->posiciones[alumno->posicion_fila][alumno->posicion_columna]--;
  pthread_mutex_unlock(&un_aula->m_posiciones[alumno->posicion_fila][alumno->posicion_columna]);
}
```

Ver que `t_aula_liberar` está libre de deadlock también es trivial, pues de nuevo no hay *hold and wait*, ya que esta es la operación función realiza la operación inversa a `t_aula_ingresar` y ya vimos que esa función no produce deadlocks.

```
  if (...) {
    /* ... */

    pthread_mutex_lock(&aula->m_posiciones[i1][j1]);
    pthread_mutex_lock(&aula->m_posiciones[i2][j2]);
    aula->posiciones[fila][columna]++;
    aula->posiciones[alumno->posicion_fila][alumno->posicion_columna]--;
    pthread_mutex_unlock(&aula->m_posiciones[i2][j2]);
    pthread_mutex_unlock(&aula->m_posiciones[i1][j1]);
  } else {
    pthread_mutex_lock(&aula->m_posiciones[alumno->posicion_fila][alumno->posicion_columna]);
    aula->posiciones[alumno->posicion_fila][alumno->posicion_columna]--;
    pthread_mutex_unlock(&aula->m_posiciones[alumno->posicion_fila][alumno->posicion_columna]);
  }
```

Esta pieza de código corresponde al de `intentar_moverse`. En esta situación sí nos encontramos con un *hold and wait*, con *no preemtion* y con *exclusión mutua*. Veamos que no hay *espera circular*. Como vimos durante la cursada, si establecemos una relación de orden a la hora de pedir por los recursos podemos eliminar la espera circular.

Si bien en el código citado no está, los valores de los índices para acceder a la matriz son los que marcarán qué mutex de la matriz se pedirá primero, y cual después.

La explicación sobre la relación de orden establecida ya fue explicada anteriormente.

Es por ésto entonces que dicha pieza de código se encuentra libre de *deadlock*.

```
void colocar_mascara(t_aula *el_aula, t_persona *alumno)
{
  printf("Esperando rescatista. Ya casi %s!\n", alumno->nombre);
  pthread_mutex_lock(&el_aula->m_rescatistas);
  while(el_aula->rescatistas_disponibles <= 0)
    pthread_cond_wait(&el_aula->vc_rescatistas, &el_aula->m_rescatistas);
  el_aula->rescatistas_disponibles--;
  pthread_cond_signal(&el_aula->vc_rescatistas);
  pthread_mutex_unlock(&el_aula->m_rescatistas);

  alumno->tiene_mascara = true;

  pthread_mutex_lock(&el_aula->m_rescatistas);
  el_aula->rescatistas_disponibles++;
  pthread_mutex_unlock(&el_aula->m_rescatistas);
}
```

Ver que `colocar_mascara` está libre de deadlock es sencillo, pues no se produce *hold and wait*, dado que se pide sólo un recurso, el cual se libera inmediatamente después de ser utilizado y esta es la única función que interactúa con este recurso.

```
  /* Si hay personas saliendo, espero sin modificar las cantidades. */
  pthread_mutex_lock(&el_aula->m_estado);
  while (el_aula->saliendo) {
    pthread_cond_wait(&el_aula->vc_estado, &el_aula->m_estado);
  }
```

```
/* Una vez que sé que no hay gente saliendo, me anoto para salir. */
el_aula->cantidad_de_personas--;
el_aula->para_salir++;

/* Si soy el quinto o el último, empezamos a salir. */
el_aula->saliendo = el_aula->para_salir == 5 ||
                    el_aula->cantidad_de_personas == 0;

while (!el_aula->saliendo) {
  pthread_cond_wait(&el_aula->vc_estado, &el_aula->m_estado);
}
el_aula->para_salir--;
pthread_cond_broadcast(&el_aula->vc_estado);
/* Si somos 0, habilitamos a la gente a entrar, migo. */
el_aula->saliendo = el_aula->para_salir != 0;
pthread_mutex_unlock(&el_aula->m_estado);
```

Para ver que en esta pieza de código no hay deadlock, debemos ver que en ningún momento el *thread* queda esperando un suceso causado por otro *thread* que nunca va a llegar.

En primera instancia la variable `saliendo` se inicializa en falso, con lo cual el primer alumno que llega no se queda *trabado* en el while inicial. Luego, libera el mutex al hacer `pthread_cond_wait` sobre la variable de condición y el mutex.

Si no hay más alumnos en el aula, la variable `saliendo` estará en verdadero, con lo cual el alumno no entrará en el ciclo y terminará de ejecutar el código analizado.

Si por el contrario hay más alumnos dentro del aula, `saliendo` estará en verdadero, y el alumno quedará esperando en el segundo `while` hasta que se junte su grupo, piediendo y liberando el mutex mediante el `pthread_cond_wait`.

Una vez que el grupo se forma, se envían señales a todos los alumnos que estaban esperando por la variable de condición, pero recién se libera el mutex asociado una vez que son actualizadas las variables y `saliendo` es seteado con el valor correcto.

Con lo cual, una vez que el alumno termina, sea cual fuera su situación, manda señales a los alumnos que están esperando, y libera el mutex, quedando así esta pieza libre de deadlock.

Habiendo analizado todos los códigos que contienen mutexes y variables de condición, y habiendo visto que cada uno de ellos se encuentra libre de deadlock, podemos afirmar que el sistema en su conjunto se encuentra libre de deadlock.

5. Escalabilidad

Los algoritmos encargados de sincronizar los *threads* para soportar concurrencia soportan cualquier número de clientes, sin embargo nos encontramos con problemas en otras secciones de la implementación.

El problema más grande ocurre al tratar de gestionar un número tan grandes de conexiones, ya que si la cantidad supera un número dado, el servidor empieza a perder los pedidos de conexión de los clientes.

En primera instancia, se necesita una cola de conexiones pendientes de mayor tamaño, idealmente tan grande como la cantidad de conexiones que quiero atender. Pues en el peor caso, todos quieren conectarse en el mismo instante.

El problema con esta solución es que aunque el valor pasado como parámetro a `listen` sea muy grande, si éste es mayor al valor descripto en `/proc/sys/net/core/somaxconn`, entonces es truncado a dicho valor [2].

Entonces, una posible solución (de software) podría ser aumentar el valor descripto por la variable para tener una cola de conexiones pendientes de mayor tamaño. De todas formas, habría que evaluar si el sistema permite tales modificaciones.

[2] Información provista por el man en la entrada correspondiente a `listen`, en la sección de notas, en el último párrafo.

Sea cual fuere la solución, el problema es claro, dada esta implementación, el servidor no soporta tantas conexiones de forma concurrente, ya que la mayoría de las conexiones se pierde. De hecho, investigando el problema, encontramos que es un problema común en el desarrollo de servidores, principalmente servidores web, que son propensos a recibir un alto caudal de conexiones concurrentes a diario.

Dicho problema se encuentra descripto en `http://kegel.com/c10k.html`. También encontramos alguna información relevante en `http://www.acme.com/software/thttpd/notes.html#listen`, en el ítem *On the listen queue length*, donde describe que usando el tamaño de cola por defecto (5) la cantidad de conexiones aceptadas por segundo es de 3, en cambio, aumentando el tamaño de la cola a 32, el valor aumenta a $10/20$ conexiones por segundo.

Esto lo pudimos ver en nuestra implementación. Corriendo un test hecho por nosotros [3] con 1000 clientes intentando conectarse de forma concurrente con el tamaño de la cola en 5 se perdían conexiones; en cambio, aumentando el tamaño 128 (el máximo provisto por el sistema) ya no se generaban conexiones perdidas.

De esta forma vemos como el cuello de botella está dado a la hora de establecer las conexiones con los clientes.

Sin embargo, este podría no ser el único problema. Con un millón de conexiones, el servidor crearía esa cantidad de *threads*, y cada thread necesita una cantidad de recursos determinada, memoria entre otras cosas. Podría requerir una cantidad de memoria que el sistema no tenga disponible.

Si éste problema se da, una posible solución (de hardware) es agregar tanta memoria como necesiten un millón de threads. Otra solución, que requeriría modificar el diseño del servidor (y en consecuencia, su implementación), sería montarlo en un *cluster*, y cada vez que se atiende a un cliente, derivarlo a otra computadora, manteniendo una distribución equilibrada en las distintas computadoras.

[3] `tester_concurrente.py`