Trabajo Práctico de Sistemas Operativos

23 de noviembre de 2014

Universidad de Buenos Aires - Departamento de Computación - FCEN

Integrantes:

- Castro, Damián L.U.: 326/11 ltdicai@gmail.com
- Toffoletti, Luis L.U.: 827/11 luis.toffoletti@gmail.com
- Zanollo, Florencia L.U.: 934/11 florenciazanollo@gmail.com

Índice

1. Introducción

En los sistemas operativos tradicionales, cada proceso tiene un espacio de dirección y un único hilo (o thread) de control. De hecho, esta es casi la definición de proceso. Sin embargo, en muchas situaciones, es deseable tener múltiples threads de control en el mismo espacio de dirección corriendo quasi-paralelamente, como si fueran (casi) procesos diferentes (excepto por el espacio de direcciones compartido).

La razón principal para utilizar estos microprocesos, llamados threads, es que en muchas aplicaciones hay multiples actividades ocurriendo al mismo tiempo. Algunas de ellas pueden bloquearse de vez en cuando. Al descomponer tal aplicación en multiples threads sequenciales que corren quasi-paralelamente, el modelo de programación se vuelve más simple.

Un segundo argumento de por qué tenerlos es que, al ser más livianos que los procesos, es más fácil (y por lo tanto más rápido) crearlos y destruirlos.

La tercer razón para usarlos tiene que ver con rendimiento. Los threads no mejoran el rendimiento si todos ellos están ligados a un CPU, pero cuando hay una gran cantidad de CPU y dispositivos de I/O, tener threads permite a las actividades superponerse entre sí, mejorando así el rendimiento.

Por último, los threads son útiles en sistemas con multiples CPUs, dónde es posible el paralelismo real.

Historicamente, los proveedores de hardware implementaban su propia versión de threads. Estas implementaciones diferían sustancialmente entre sí, haciendo dificil para los programadores desarrollar aplicaciones portables que utilicen threads.

Para hacer posible la portabilidad de programas, la IEEE definió un estandar. Las implementaciones que adhieren a este estandar son referidas como POSIX threads o **Pthreads**. Los Pthreads están definidos como una librería del lenguaje C, conteniendo de tipos de datos y llamadas a procedimientos. La mayoría de los sistemas UNIX la soportan. El estandar define más de 60 funciones.
Cada pthread tiene las siguientes propiedades:

- Identificador.
- Set de registros (incluído el *program counter*).
- Set de atributos (que son guardados en una estructura e incluyen el tamaño del *stack* entre otros).

Pthreads también proporciona un número de funciones que pueden ser usadas para sincronizar threads. El mecanismo más básico usa una variable de exclusión mutua o **mutex**, que puede ser bloqueada o desbloqueada para proteger una sección crítica.
Si un thread quiere entrar en una sección crítica primero trata de bloquear el mutex asociado a esa sección. Si estaba desbloqueado entra y el mutex es bloqueado automáticamente, previniendo así que otros threads entren. Si ya estaba bloqueado, el thread se bloquea hasta que el mutex sea liberado.
Si múltiples threads están esperando el mismo mutex, cuando éste es desbloqueado, sólo le es permitido continuar a uno de ellos. Estos bloqueos no son mandatorios. Queda a cargo del programador asegurar que los threads los utilizan correctamente.

Además de los mutex, Pthreads ofrece un segundo mecanísmo de sincronización: **variables de condición**. Éstas permiten a los threads bloquearse debido al incumplimiento de alguna condición. Casi siempre estos mecanismos son utilizados conjuntamente.

2. Resolución

2.1. Variables utilizadas

2.1.1. Compartidas

t_aula aula. Estructura que contiene:

- Una matríz de enteros donde cada posición equivale a 1 m^2 del aula.
- Cantidad de personas dentro del aula.
- Cantidad de rescatístas.

2.1.2. Privadas

t_persona alumno. Estructura que contiene:

- Nombre
- Posición en la que se encuentra (fila y columna)
- Un booleano indicando si salió y otro si tiene la máscara.

t_salidas salidas. Estructura que contiene:

- Cantidad de personas en el pasillo. El pasillo es una zona intermedia entre el aula y el grupo. En él se encuentran los alumnos con máscara esperando para poder ingresar en un grupo.
- Cantidad de personas en el grupo. En el grupo los alumnos esperan ser cinco para poder ser evacuados.

2.1.3. Caso especial

t_parametros. Estructura que contiene:

- Identificador del socket que utilizará el pthread para comunicarse con el cliente. Este socket es único para cada pthread.
- Variable compartida *el_aula*.
- Variable privada *salidas*.

La implementación de pthreads sólo admite el pasaje de un parámetro y debe ser una estructura, por eso t_parametros es necesaria. Al crearla utilizamos malloc para alojarla en memoria, de no hacerlo de esta forma la variable 'muere' al final del *scope* y el pthread estaría accediendo a posiciones inválidas (contienen basura).

2.2. Mutexes y Variables de Condición

Para sincronizar los procesos utilizamos:

- Mutexes

Nombre del mutex	**Para control de acceso a...**
mutex_posiciones (matriz de tamaño ancho del aula por alto del aula)	las posiciones del aula.
mutex_cantidad_de_personas	la cantidad de personas en el aula.
mutex_rescatistas	la cantidad de rescatistas en el aula.
mutex_pasillo	la cantidad de personas en el pasillo, es decir, fuera del aula (con máscara) pero no en el grupo.
mutex_grupo	la cantidad de personas en el grupo, es decir, esperando formar grupo de 5 para salir.

- Variables de condición

Nombre de la variable	Condición de bloqueo	Explicación
cond_hay_rescatistas	Cantidad de rescatistas disponibles = 0	Espera a que haya un rescatista libre.
cond_grupo_lleno	Cantidad de personas en el grupo >= 5	Espera a que haya espacio en el grupo de evacuacion.
cond_estan_evacuando	Cantidad de personas en el grupo $\neq$ 5 y hay más gente en el aula o en el pasillo	Espera a que sean 5 en el grupo de evacuación a menos que sea la última persona en salir.
cond_salimos_todos	Cantidad de personas en el grupo >0	Espera a que termine de salir todo el grupo.

2.3. Implementación

Código del archivo servidor_multi.c

```
1: procedure T_AULA_INICIAR_VACIA(aula)
2:     for i=0 to ancho del aula do
3:         for j=0 to alto del aula do
4:             aula.posicion[i][j] = 0
5:         end for
6:     end for
7:     aula.cantidad_de_personas = 0
8:     aula.rescatistas_disponibles = Cantidad inicial
9: end procedure
```

Inicializa un aula vacía con cierta cantidad de rescatistas.

```
1: procedure T_AULA_INGRESAR(aula, alumno)
2:     LOCK(mutex_cantidad_de_personas)
3:         aula.cantidad_de_personas ++
4:     UNLOCK(mutex_cantidad_de_personas)
5:     LOCK(mutex_posiciones[alumno.fila][alumno.columna])
6:         aula.posicion[alumno.fila][alumno.columna] ++
7:     UNLOCK(mutex_posiciones[alumno.fila][alumno.columna])
8: end procedure
```

Ingresa un alumno al aula en la posición indicada por él. Para que no haya problemas de concurrencia se piden los mutex que actúan sobre la cantidad de personas del aula y de la posición indicada.

```
1: procedure T_AULA_LIBERAR(aula, alumno)
2:     LOCK(mutex_cantidad_de_personas)
3:         aula.cantidad_de_personas - -
4:     UNLOCK(mutex_cantidad_de_personas)
5: end procedure
```

Retira un alumno del aula. Por la misma razón que antes se pide el mutex de la cantidad de personas.

```
1: procedure TERMINAR_SERVIDOR_DE_ALUMNO(socket_fd, aula, alumno)
2:     close(socket_fd)
3:     t_aula_liberar(aula, alumno)
4:     EXIT()                                          ▷ Cierra el pthread.
5: end procedure
```

Termina la conexión con el cliente.

```
1: function INTENTAR_MOVERSE(aula, alumno, direccion)
2:     Calculo la nueva_posicion = (fila, columna)
3:     if nueva_posicion es la salida then
4:         alumno.salio = true
5:         pudo_moverse = true
6:     end if
7:     LOCK(mutex_posiciones[nueva_posicion])
8:     if nueva_posicion esta entre límites and aula.posicion[nueva_posicion] <Máx.cant. personas then
9:         pudo_moverse = true
10:    end if
11:    UNLOCK(mutex_posiciones[nueva_posicion])

12:    vieja_posicion = (alumno.fila, alumno.columna)

13:    if pudo_moverse then
14:        if nueva_posicion.fila <vieja_posicion.fila or (nueva_posicion.fila == vieja_posicion.fila
15:        and nueva_posicion.columna <vieja_posicion.columna) then
16:            if !alumno.salio then
17:                LOCK(mutex_posiciones[nueva_posicion.fila][nueva_posicion.columna])
18:            end if
19:            LOCK(mutex_posiciones[vieja_posicion.fila][vieja_posicion.columna])
20:            orden_locks = 0
21:        else
22:            LOCK(mutex_posiciones[vieja_posicion.fila][vieja_posicion.columna])
23:            if !alumno.salio then
24:                LOCK(mutex_posiciones[nueva_posicion.fila][nueva_posicion.columna])
25:            end if
26:            orden_locks = 1
27:        end if
28:        if !alumno.salio then
29:            aula.posicion[nueva_posicion] ++
30:        end if
31:        aula.posicion[vieja_posicion] - -
32:        actualizar posicion del alumno
33:        if orden_locks == 0 then
34:            UNLOCK(mutex_posiciones[vieja_posicion.fila][vieja_posicion.columna])
35:            if !alumno.salio then
36:                UNLOCK(mutex_posiciones[nueva_posicion.fila][nueva_posicion.columna])
37:            end if
38:        else
39:            if !alumno.salio then
40:                UNLOCK(mutex_posiciones[nueva_posicion.fila][nueva_posicion.columna])
41:            end if
42:            UNLOCK(mutex_posiciones[vieja_posicion.fila][vieja_posicion.columna])
43:        end if
44:    end if
45:    return pudo_moverse
46: end function
```

Intenta mover al alumno dentro del aula hacia la dirección indicada. Para esto debe tomar los mutex de la posición vieja y nueva (si es que no salió), se toman en determinado orden para evitar espera circular (ver sección 2.4). Luego devuelve si el alumno pudo moverse o no.

```
1: procedure COLOCAR_MASCARA(aula, alumno)
2:     alumno.tiene_mascara = true
3: end procedure
```

Coloca la máscara en el alumno.

```
1: procedure ATENDEDOR_DE_ALUMNO(socket_fd, aula, salidas, alumno)
2:     if no se pudo recibir el nombre y la posicion then
3:         terminar_servidor_de_alumno(socket_fd, NULL, NULL)
4:     end if
5:     t_aula_ingresar(aula, alumno)
6:     for ever do
7:         if no se pudo recibir la direccion then
8:             terminar_servidor_de_alumno(socket_fd, aula, alumno)
9:         end if
10:        pudo_moverse = intentar_moverse(aula, alumno, direccion)
11:        enviar_respuesta(socket_fd, pudo_moverse)
12:        if alumno.salio then
13:            break
14:        end if
15:    end for

16:    LOCK(mutex_rescatistas)
17:    while aula.rescatistas_disponibles == 0 do
18:        COND_WAIT(cond_hay_rescatistas, mutex_rescatistas)
19:    end while
20:        aula.rescatistas_disponibles - -
21:    UNLOCK(mutex_rescatistas)

22:    colocar_mascara(aula, alumno)

23:    LOCK(mutex_rescatistas)
24:        aula.rescatistas_disponibles ++
25:    UNLOCK(mutex_rescatistas)
26:    COND_SIGNAL(cond_hay_rescatistas)

27:    LOCK(mutex_pasillo)
28:        t_aula_liberar(aula, alumno)
29:        salidas.cant_personas_pasillo ++
30:    UNLOCK(mutex_pasillo)

31:    LOCK(mutex_grupo)
32:    while salidas.cant_personas_grupo >= 5 do
33:        COND_WAIT(cond_grupo_lleno, mutex_grupo)
34:    end while
35:    LOCK(mutex_pasillo)
36:        salidas.cant_personas_pasillo - -
37:        salidas.cant_personas_grupo ++
```

```
38:     UNLOCK(mutex_pasillo)
39:     LOCK(mutex_pasillo)
40:     LOCK(mutex_cantidad_de_personas)
41:     if salidas.cant_personas_grupo == 5 or (salidas.cant_personas_pasillo == 0
42:     and aula.cantidad_de_personas == 0) then
43:         UNLOCK(mutex_cantidad_de_personas)
44:         UNLOCK(mutex_pasillo)
45:         COND_BROADCAST(cond_estan_evacuando)
46:     else
47:         UNLOCK(mutex_cantidad_de_personas)
48:         UNLOCK(mutex_pasillo)
49:         COND_WAIT(cond_estan_evacuando, mutex_grupo)
50:     end if
51:     salidas.cant_personas_grupo - -
52:     if salidas.cant_personas_grupo >0 then
53:
54:         COND_WAIT(cond_salimos_todos, mutex_grupo)
55:     else
56:         COND_BROADCAST(cond_salimos_todos)
57:         COND_BROADCAST(cond_grupo_lleno)
58:     end if
59:     UNLOCK(mutex_grupo)

60:     enviar_respuesta(socket_fd, LIBRE)
61: end procedure
```

Si no se puede recibir el nombre y la posicion se supone una conexión fallida y se corta dicha conexión. Luego se reciben direcciones hasta que el alumno logra salir del aula. En caso de no recibir una dirección, de nuevo, se supone una conexión fallida y se corta dicha conexión.

Cuando llega a la salida queda a la espera de un rescatista. En cuanto consigue uno le es colocada la máscara y luego el rescatista vuelve a quedar libre. Notar que para sincronizar la variable rescatistas_disponibles es pedido el mutex antes y después de modificarla, además en vez de esperar de forma activa (*busy waiting*) utilizamos una variable de condición que despierta cuándo hay rescatistas disponibles.

Una vez que el alumno tiene la máscara puesta, entra a un lugar entre el aula y la salida al cuál decidimos llamar 'pasillo'. Recién entonces decimos que el alumno salió efectivamente del aula (linea 28). La cantidad de personas en el pasillo también está sincronizada con un mutex.

En este momento el alumno debe esperar a que haya lugar en el grupo de evacuación. De nuevo utilizamos una variable de condición que despierta si esto pasa. Ahora el alumno pasa del pasillo al grupo, si son 5 o es el último en salir avisa a los demás que pueden evacuar. De lo contrario queda a la espera de la señal.
De a uno se retiran del grupo (se descuentan de la cantidad de personas en el grupo, variable que está sincronizada). En cuanto todos se hayan retirado del grupo salen del edificio juntos.

```
1: function MAIN(void)
2:     Inicializar conexion con sockets y permitir 5 conexiones en espera
3:     t_aula_iniciar_vacia(aula)
4:     Inicializar mutexes
5:     Inicializar variables de condicion
6:     for ever do                                    ▷ Loop de atencion al cliente
7:         if Se conecto un nuevo cliente then
```

```
8:              Inicializar pthread y sus parametros
9:              Crear pthread con la funcion 'atendedor_de_alumno'
10:         end if
11:     end for
12: end function
```

Crea un nuevo pthread por cada cliente que se conecta al servidor, es decir, uno por cada alumno que ingrese al aula. Lo crea con la función atendedor_de_alumno la cuál está explicada más arriba.

2.4. Deadlock

Un grupo de procesos están en estado de *deadlock* si cada uno de ellos está esperando un evento que sólo otro proceso del grupo puede causar. Vamos a analizar el código para demostrar que está libre de deadlocks.

Antes de eso veremos cuáles son las condiciones que debe cumplir un sistema para tener la posibilidad de llegar a un estado de deadlock, llamadas condiciones de Coffman:

- Exclusión mutua: cada recurso está asignado exactamente a un proceso o está disponible.
- Hold-and-wait: Los procesos que tienen asignado un recurso pueden requerir otro/s recurso/s.
- No-preemption: Los recursos asignados a procesos no pueden ser removidos por la fuerza.
- Espera circular: Debe haber una lista de dos o más procesos, cada uno de ellos esperando un recurso que tiene el anterior.

En nuestra implementación no se cumple la última condición de Coffman, por ende está libre de deadlocks. Entonces veamos por qué no hay espera circular:

Para asegurarnos de que no haya espera circular en cuanto a las posiciones del aula lo que hicimos fue disponer un orden de bloqueos. De esta forma, si dos clientes deben bloquear las posiciones 1 y 2. Ambos lo van a hacer en el mismo orden. Entonces si 1 <2, uno de ellos tomará el lock de la posición 1 y el otro se quedará esperando a que se libere. Núnca sucede que uno de ellos toma el 2 sin tener el 1.

Otro momento donde hay hold and wait es cuando el cliente tiene el mutex del grupo y pide el del pasillo. Pero no hay ningún lugar en el código donde pueda llegar a pedir primero el pasillo y luego el grupo. Notar que sí hay un pedido del mutex del pasillo sin el del grupo, pero en ese caso el cliente eventualmente desbloqueará el pasillo, permitiendo a los demás seguir su curso. Grupo-pasillo-aula, explicar porq no hay deadlock con eso Más grafiquitos, los está haciendo Dami

Una variante de deadlock es livelock, ambos son formas de inanición. Excepto que el estado de los procesos envueltos en el livelock está cambiando constantemente, aunque sin avanzar. Al suponer que los clientes son personas lógicas evitamos caer en un livelock, ya que no tiene sentido moverse en círculos dentro del aula en vez de salir.

2.5. Paralelismo

El objetivo de esta implementación es maximizar el grado de paralelismo, por esta razón decidimos utilizar semáforos independientes para las posiciones del aula y sus atributos.

En un principio nuestra solución utilizaba un unico mutex para restringir el acceso a la estructura del aula, si bien esta solución funcionaba no se permitian movimientos paralelos dentro del aula, aunque esas posiciones fueran totalmente distintas, lo cual quita paralelismo.

Además se buscó reducir las secciones críticas lo mayor posible. Así, por ejemplo, mientras a un alumno le están poniendo una máscara otro puede bloquear el mutex de los rescatistas ya que no está siendo modificada la cantidad de ellos.

3. Escalamiento

En caso de un aumento drástico en la cantidad de alumnos el problema a tener en cuenta es la memoria.

Un pthread es creado por cada alumno (o cliente) que se conecta al servidor. Esto implica un nuevo stack, el tamaño del stack es variable. El valor por defecto sobre un sistema de 32 bits (y lo vamos a asumir para este caso) es de 2MB.

Entonces para un servidor con 10 clientes (10 pthreads) se necesita[1] una memoria de $10 * 2$ MB $= 20$ MB.
Y para un servidor con 1000 clientes se necesita una memoria de 2GB.

La solución obvia mediante hardware es agregar más memoria principal al servidor de manera que alcance para la cantidad de clientes. Esto conlleva un costo muy alto y puede llegar a volverse inmanejable.

Soluciones mediante software:

Sin pthreads: Se podría implementar un servidor capáz de atender a distintos clientes en 'simultaneo'. Con un mecanísmo de scheduling tal y como se simula paralelismo de procesos contando con solo un CPU. El problema de esta solución es que puede tardar mucho en atender a los clientes (depende estríctamente de la cantidad y el mecanísmo usado). Incluso podría haber *starvation*.

Con pthreads: Existe una solución simple que no modifica la implementación original y otra, compleja, que si lo hace.

- **Simple:** El tamaño del stack de cada pthread se puede modificar. Entonces se podría utilizar un stack más pequeño para no tener tantos problemas de memoria. La contraparte de esta técnica es que se debe definir un tamaño de stack apropiado, debe tener espacio para contener todas las variables privadas del pthread.
- **Compleja:** En vez de tener un pthread por cada cliente, se pueden compartir. Si se crea un pthread que atiende a X cantidad de clientes entonces la cantidad total de memoria requerida para N clientes es de $\frac{N}{X} * 2\ MB$.
  Se debe elegir un X adecuado ya que si es muy chico se consume más memoria pero los clientes se atienden más rápido y viceversa de ser muy grande.
  Por ejemplo para X = 10 la cantidad total de memoria requerida para 1000 clientes es de $\frac{1000}{10} * 2$ MB $= 200$ MB. Se reduce en un 90 %.

[1] En realidad si la memoria no alcanza se podría usar *swapping* pero esto generaría otro problema: *thrashing* (se cargan y descargan sucesiva y constantemente partes de la imagen de un proceso desde y hacia la memoria virtual)