

Trabajo Práctico de Sistemas Operativos

13 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	3
2. Resolución	3
2.1. Variables utilizadas	3
2.2. Mutexes	4
2.3. Deadlock	4
2.4. Implementación	4
3. Escalamiento	4

1. Introducción

Para hacer posible la portabilidad de programas con threads, la IEEE definió un estandar, la librería que define estos threads es llamada **Pthreads**. 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 (incluido 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 mecanismo 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

Compartidas: t_aula el_aula. Es una estructura que almacena los datos de una reserva. Contiene:

- Una matriz de enteros donde cada posición equivale a $1m^2$ del aula.
- Cantidad de personas dentro del aula.
- Cantidad de rescatistas.

Privadas: t_persona alumno. Contiene:

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

Caso especial: t_parametros. Es una estructura que contiene lo siguiente:

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

Es necesario tener una estructura para pasar los parámetros porque la implementación de pthreads sólo admite el pasaje de un parámetro y debe ser una estructura.

Al crearlo utilizamos malloc para alojarlo 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

2.3. Deadlock

2.4. Implementación

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. Normalmente (y lo vamos a asumir para este caso) es de 2MB.

Entonces para un servidor con 10 clientes (10 pthreads) se necesita¹ una memoria de $10 * 2 \text{ MB} = 20 \text{ MB}$.

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 capaz de atender a distintos clientes en ‘simultaneo’.

Con un mecanismo 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 estrictamente de la cantidad y el mecanismo usado). Incluso podría haber *starvation*.

Con pthreads: En vez de tener un pthread por cada cliente se pueden compartir. Es un poco mezclar la idea anterior con pthreads.

Si se crea un pthread que atiende a X cantidad de clientes entonces la cantidad total de memoria requerida para N clientes es de $\frac{N}{X} * 2 \text{ MB}$. 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 $\frac{1000}{10} * 2 \text{ MB} = 200 \text{ MB}$. Se reduce en un 90 %.

¹En realidad si la memoria no alcanza se podría usar *swapping* pero esto generaría otro problema: *thrashing*