Trabajo Práctico de Sistemas Operativos

18 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 el_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

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 la condición de Coffman que no se cumple es la última, por ende está libre de deadlocks. Entonces queremos probar que 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 uno de ellos...

Libre de deadlock! ¿por que? ¿qué condicion de don Coffsy no se cumple? Explicar las condiciones de Cofsy (muy poco - citarlo)

No hay espera circular porque imponemos un orden a como tomar los locks de las posiciones del aula. Agregar grafiquito de la matriz con sus flechitas de mayor, menor o casos.

Hold and wait sin espera circular: nadie que tenga el pasillo va a pedir el grupo entonces siempre se pide el grupo en algún momento se libera el pasillo. Nunca se da la situación de que alguien tiene el grupo y esta esperando el pasillo mientras otro tiene el pasillo mientras espera el grupo. Orden determinado, primero grupo, dsp pasillo, dsp aula ponele. Más grafiquitos, los está haciendo Dami

Una variante de deadlock es livelock, algo de explicación, no pasa porq asumimos q los clientes son personas lógicas!

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)