DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Sistemas Operativos

Trabajo práctico 2

Pthreads

Resumen

Integrante	LU	Correo electrónico
Acosta, Javier Sebastian	338/11	acostajavier.ajs@gmail.com
Gomez, Fernando Nahuel	695/11	fernando.gmz12@gmail.com
Russo, Christian Sebastián	679/10	christian.russo@gmail.com

Palabras claves:

Threads

Índice

1. Introducción

La nueva implementacion del servidor, capaz de atender a varios clientes simultáneamente, se encuentra en el archivo *servidor_multi.c*. Ningún otro archivo fue modificado.

En las secciones siguientes explicaremos la implementacion realizada, el porque nuestro sistema esta libre de Deadlock y como podría modificarse el servidor para un incremento masivo en la cantidad de clientes.

2. Explicación de la implementacion

A continuación daremos una explicación detallada de el funcionamiento de cada una de las funciones implementadas en el servidor y de las estructuras utilizadas.

2.1. Estructura t_aula

Esta estructura representa el aula y esta formada por:

- Una matriz de enteros que representa cada posición del aula y cuantas personas hay en dicha posición.
- Un entero que indica la cantidad total de personas en el aula y un mutex para esta variable.
- Un entero que indica la cantidad de rescatistas disponibles y un mutex para esta variable.
- Una matriz de mutexes, del mismo tamaño que la matriz del aula, donde cada mutex sirve para bloquear la posición correspondiente de la matriz que representa el aula. Esta matriz nos permite minimizar la cantidad de posiciones del aula que bloqueamos al mover a un alumno a solo dos, en lugar de tener que bloquear la matriz completa si usáramos un solo mutex para toda la matriz.
- Una variable de condición que utilizamos para saber si hay al menos un rescatista disponible.
- Una variable de condición que utilizamos para saber si ya tenemos formado un grupo de alumnos en la salida.
- Un booleano que indica si ya se formo un grupo de alumnos en la salida del aula, un entero que indica cuantos alumnos del grupo actual *(si es que ya se formo un grupo)* salieron del aula y un mutex para ambas variables *(el mismo para las dos)*.

2.2. Estructura thdata

Esta estructura contiene los datos requeridos por los threads que atienden a cada cliente. Estos datos son:

- El file descriptor de la conexión con el cliente (alumno).
- Un puntero a la estructura del aula en la que se encuentra el alumno.

2.3. Funcion t_aula_iniciar_vacia

Se encarga de inicializar la estructura t_aula pasada por parametro.

En esta función, primero colocamos cero en todas las posiciones de la matriz que representa el aula, indicando que no hay ningún alumno en ellas e inicializamos el mutex correspondiente a cada posición en *No tomado (Null, 1, o el valor que use la representación interna)*.

Luego, inicializamos la cantidad de personas en el aula en cero y la cantidad de rescatistas disponibles en el valor de la constante RESCATISTAS. También inicializamos los mutex correspondientes a ambas variables en *No tomado*.

También inicializamos en *No tomado* el mutex de la cantidad de personas en el aula y el de rescatistas disponibles.

Inicializamos la variable booleana que indica que se formo un grupo de alumnos en la salida en *false* y el entero que indica la cantidad de alumnos que salieron del grupo actual en cero.

Finalmente, inicializamos las variables de condición relacionadas con los rescatistas disponibles y con la salida del aula a su valor por defecto.

2.4. Funcion t_aula_ingresar

Se encarga de ingresar a un alumno al aula.

Aqui, primero, pedimos el mutex correspondiente a la variable de la estructura t_aula que indica el número de personas, luego la incrementamos en uno y liberamos el mutex.

Luego, pedimos el mutex de **la posición en el aula** que corresponde a la posición inicial del alumno, incrementamos la cantidad de personas en esa posición del aula en uno y liberamos el mutex.

2.5. Funcion t_aula_liberar

Se encarga de sacar a un alumno del aula.

Lo primero que hacemos en esta función es incrementar el numero de alumnos en la salida en uno.

Luego, si ya se formo un grupo de alumnos en la salida *(es decir, ya hay mas de 5 alumnos es la salida)* el alumno actual deberá esperar a que dicho grupo abandone el aula.

En caso contrario el alumno deberá esperar a que se forme un grupo de cinco alumnos en la salida.

Si el alumno es el ultimo necesario para completar el grupo, entonces avisa a los otros cuatro alumnos que están esperando que ya pueden salir y estos salen.

El ultimo alumno del grupo en salir se encarga de resetear las variables de la estructura del aula que indican que un grupo esta saliendo y de avisar a los otros alumnos que querian salir *(si es que hay)*, que ya pueden formar otro grupo.

2.6. Funcion terminar_servidor_de_alumno

Termina la conexión con el cliente del alumno, lo saca del aula y termina la ejecución del thread en el servidor encargado de atender a dicho cliente.

2.7. Funcion intentar_moverse

Se encarga de actualizar la posición del alumno en el aula realizando el movimiento solicitado.

Lo primero que hacemos aqui es pedir el mutex de la posición destino para determinar si en esta hay espacio suficiente para el alumno y luego liberamos el mutex.

Luego determinamos si el alumno pudo moverse, es decir, si este llegó a la salida o si la posición destino está dentro del aula (es decir, que el alumno no atravesó las paredes del aula) y hay espacio para una persona más en dicha posición.

Si el alumno no puede moverse la función termina retornando false (es decir, el alumno no pudo moverse).

En cambio, si el alumno si puede moverse, pedimos el mutex de su posición actual y de la posición destino de modo de actualizar ambas posiciones simultáneamente.

Aqui, además, chequeamos ciertas condiciones sobre las posiciones de forma de determinar en qué orden pedir los dos mutex. Esto lo hacemos para evitar que se produzca deadlock y lo explicamos con detalle en la sección **Deadlock**.

Una vez que incrementamos en uno el valor de la posición destino y disminuimos en uno el valor de la posición actual, liberamos ambos mutex, actualizamos la posición del alumno y la función termina retornando true (es decir, el alumno pudo moverse).

2.8. Funcion colocar_mascara

Se encarga de colocar la máscara al alumno, una vez que el alumno llegó a la salida del aula.

Primero pedimos el mutex de los rescatistas, luego el thread queda a la espera *(duerme)* si no hay ningún rescatista disponible y continua su ejecución cuando esta condición cambia *(cond_signal)*, es decir, hay al menos un rescatista disponible.

Luego disminuimos la cantidad de rescatistas en uno, indicando que hay un rescatista más ocupado poniendo una máscara y por último liberamos el mutex. A continuación le colocamos la máscara al alumno.

Nuevamente pedimos el mutex de los rescatistas, incrementamos la cantidad de rescatistas en uno y liberamos el mutex.

Finalmente realizamos un signal sobre la variable de condición *(cond_signal)* de la estructura que representa el aula para indicar que hay un rescatista más disponible.

2.9. Funcion atendedor_de_alumno

Es la función principal que ejecuta cada thread.

Al comienzo, esta función recibe del cliente el nombre del alumno y su posición en el aula y llama a la función **t_aula_ingresar**, para colocar al alumno dentro de la estructura que representa al aula en la posición indicada.

Luego, la función entra en un ciclo que solo termina si se corta la comunicación con el alumno o si este llega a la salida del aula.

En cada iteración del ciclo, el cliente indica en qué dirección se desea mover el alumno. En respuesta, el servidor llama a la función **intentar_moverse** y envía al cliente el mensaje **OK** en caso de que se haya podido realizar el movimiento requerido o el mensaje **OCUPADO**, en caso contrario.

En cada movimiento se chequea si el alumno llegó a la salida del aula, en caso afirmativo se procede a llamar a las funciones **colocar_mascara** *(para que un rescatista le coloque una máscara al alumno)* y a **t_aula_liberar** *(para sacar al alumno del aula)*.

Finalmente, se envía al cliente el mensaje **LIBRE** y se termina la ejecución del thread del servidor encargado de atender a dicho cliente.

En caso de que el alumno no llegue a la salida al moverse, en thread queda a la espera de que el cliente que debe atender le indique una nueva dirección en la que mover al alumno.

2.10. Funcion crear_thread_servidor

Se encarga de crear un thread en el servidor para atender exclusivamente a la conexión con el cliente (alumno) pasado como parámetro.

Primero creamos un thread que se encargará de ejecutar la función **atendedor_de_alumno**. A dicho thread le pasamos como parámetro un puntero a una estructura thdata que contiene la siguiente información: el file descriptor del cliente y el aula.

Estos datos son requeridos por el thread para conocer la conexión de la cual debe estar pendiente (es decir, el alumno al cual atiende) y el aula en la cual se encuentra el alumno.

2.11. Funcion main

Es la función encarga de crear los threads encargados de atender las conexiones de los clientes.

El único cambio realizado a esta funcion con respecto a la función utilizada por *servidor_mono.c* es en el ciclo FOR que se encuentra al final.

Dicho cambio consiste únicamente en llamar la función **crear_thread_servidor** *(explicada anteriormente)* cada vez que se acepta una nueva conexión con un cliente en el servidor. Este cambio es, en esencia, el que permite al servidor atender a más de un cliente simultáneamente.

3. Sistema libre de Deadlock

A continuación explicaremos por que nuestro servidor esta completamente libre de *Deadlocks*.

Primero, recordemos las cuatro condiciones de Coffman para que haya Deadlock en un sistema:

1. Exclusión mutua.
2. Retención y espera.
3. No preemption.
4. Espera circular.

Ahora queremos ver que nuestro servidor no cumple con, al menos, una de estas condiciones.

A priori, no podemos saber si en el sistema que se ejecutara el servidor existe un mecanismo arbitrario que desaloja threads, de modo que vamos a suponer que la tercer condición si se cumple.

Ahora, analizaremos de forma incremental las funciones del servidor, es decir, suponemos que el servidor no posee ninguna función e iremos añadiendo nuestras funciones de modo que ninguna tenga Deadlock ni genere un Deadlock en conjunto con funciones que ya eran parte del servidor.

Primero, notemos que las funciones **t_aula_iniciar_vacia**, **terminar_servidor_de_alumno**, **atendedor_de_alumno**, **crear_thread_servidor** y la función principal del servidor, es decir, **main** no hacen uso de mutexes ni de variables de condición, por lo cual podemos estar seguros que ninguna de estas funciones introduce un Deadlock en el sistema.
Hasta aquí, nuestro servidor no cumple con las condiciones 1, 2 y 4 de Coffman.

Luego, las funciones **t_aula_ingresar** y **colocar_mascara**, aunque si hacen uso de mutexes, solo utilizan **un mutex a la vez** cada una.

La primera función pide el mutex de la cantidad de personas en el aula, lo libera y luego pide el mutex de una posición de la matriz, incrementa dicha posición y lo libera.

La segunda funcion solo utiliza el mutex de los rescatistas y es la única función del servidor *(completo)* que lo utiliza. Primero hace una espera en una variable de condición enlazada con el mutex de los rescatistas hasta que se cumple que hay al menos un rescatista disponible y disminuye la cantidad de rescatistas disponibles en uno. Mas adelante, la función vuelve a tomar el mutex de los rescatistas y a incrementar el valor de los rescatistas disponibles.

Hasta este punto, nuestro servidor no cumple las condiciones 2 y 4 de Coffman.

Ahora añadimos al servidor la función **t_aula_liberar**. Esta función utiliza el mutex de la salida *(única función del servidor que utiliza este mutex)* y el mutex de la cantidad de personas en el aula. Esta función hace que nuestro servidor cumpla con la segunda condición de Coffman *(retención y espera)* ya que, al tener el mutex de la salida es necesario, a veces, pedir también el mutex de las personas para saber cuantas personas quedan en el aula y, así, saber a cuantas personas se debe esperar al querer salir. Pero **la situación inversa nunca se da**, es decir, en ningún momento un pthread que tenga el mutex de las personas en el aula va a necesitar pedir el mutex de la salida. Esto podemos afirmarlo porque el mutex de la salida solo es utilizado por la función **t_aula_liberar** y porque en esta función los mutex siempre se toman de la primer forma descrita, es decir, primero el de la salida y luego el de las personas.

Entonces, podemos concluir que esta función no produce que la cuarta condición de Coffman *(espera circular)* se cumpla en nuestro servidor y, por lo tanto, hasta este punto podemos afirmar que el servidor esta libre de Deadlock.

Solo nos queda analizar la función **intentar_moverse** y ver que esta no hace que nuestro servidor cumpla con la cuarta condición de Coffman *(espera circular)*.

Como dijimos anteriormente *(en la sección donde explicamos el funcionamiento de cada función)* en esta función es necesario tomar en un orden determinado los mutex de la posición del aula en la que se encuentra el alumno y de la posición del aula a la que queremos ir, ya que sino se podría dar un caso como el siguiente:

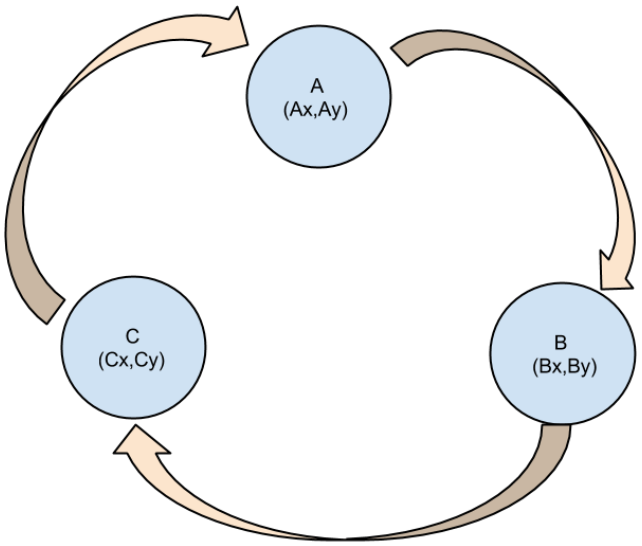

Figura 1: Espera circular

Lo que sucede aquí es que el alumno A esta parado en la posición (A_x,A_y), tiene el mutex de dicha posición y quiere ir a la posición (B_x, B_y) en la cual se encuentra el alumno B, el cual tiene el mutex de dicha posición, y a su vez este quiere ir a la posición (C_x, C_y) en la cual se encuentra el alumno C, que tiene el mutex de dicha posición y que quiere moverse a la posición (A_x, A_y) que, como dijimos, el alumno A posee su mutex.

En este caso, claramente, tenemos espera circular y, dado que nuestro servidor ya cumplía las otras tres condiciones de Coffman, tenemos Deadlock.

Para solventar este problema lo que hicimos fue **imponer un orden** en la forma en que los alumnos deben tomar los mutex para moverse en el aula. El orden impuesto es **de arriba hacia abajo y de izquierda a derecha**, siempre pensando que la posición (0, 0) del aula es la que se encuentra mas arriba y mas a la izquierda.

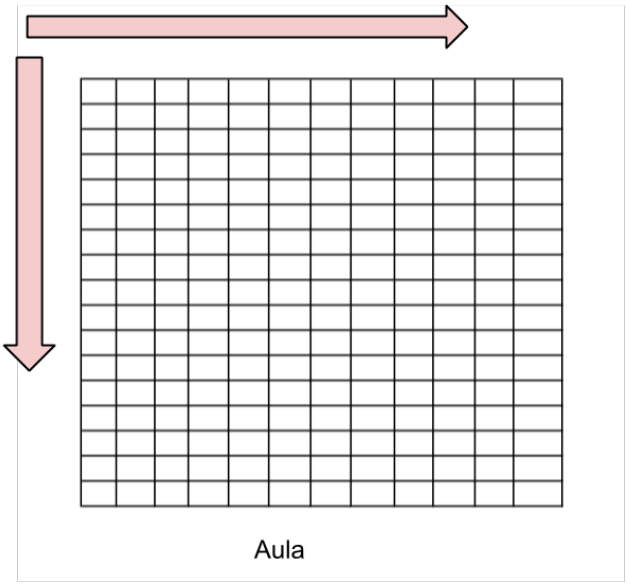

Figura 2: Orden utilizado para tomar los mutexes

Luego, según a que dirección quiera moverse el alumno, determinamos en que orden debe tomar los mutexes de su posición y de la posición destino:

- **DERECHA**: Primero toma el mutex de su posición y luego el de la posición destino.
- **IZQUIERDA**: Primero toma el mutex de la posición destino y luego el de su posición.
- **ABAJO**: Primero toma el mutex de su posición y luego el de la posición destino.
- **ARRIBA**: Primero toma el mutex de la posición destino y luego el de su posición.

Este orden nos asegura que no se puedan dar casos como el explicado anteriormente, donde $N > 2$ alumnos retienen el mutex de su posición y, a su vez, quieren obtener el mutex de su posición destino ya que, dependiendo de la relación que exista entre los valores de las posiciones de los alumnos estos deberán, en algún caso, tomar primero el mutex de su posición destino y, luego, el de su posición actual.

Por ejemplo, si tenemos un alumno A en la posición (1, 3) que quiere ir a la posición (1, 4) y en esta posición tenemos un alumno B que quiere ir a la posición (1, 3) se producirá alguno de los siguientes escenarios:

Escenario 1:

- A toma el mutex de su posición impidiendo que B lo tome.
- A toma el mutex de la posición de B, el cual B no puede tomar hasta que tengo el mutex de la posicion de A.
- A se mueve a la posición de B y libera ambos mutexes.

- B toma el mutex de la posición previa de A.
- B toma el mutex de su posición.
- B se mueve y libera ambos mutexes.

Escenario 2:

- B toma el mutex de la posición de A impidiendo que A lo tome.
- B toma el mutex de su posición, el cual A no puede tomar hasta que tenga el mutex de la posición de B.
- B se mueve y libera ambos mutexes.
- A toma el mutex de la posición previa de B.
- A toma el mutex de su posición.
- A se mueve y libera ambos mutexes.

Este ejemplo se puede extender a N alumnos *(pthreads)* y, en todos los casos, resultara que, al menos, un alumno podrá moverse sin que otro se lo impida *(a menos que el aula este repleta y no haya lugar donde moverse, pero eso escapa del control de los mutexes)*.

Finalmente, podemos concluir que el orden impuesto a la toma de los mutexes elimina la espera circular en la función y que podemos añadirla al servidor sin temor a que nos produzca Deadlock.

Luego, añadimos al servidor todas las funciones que utilizamos y llegamos a la conclusión de que este conjunto de funciones no cumple la cuarta condición de Coffman *(espera circular)* y, por lo tanto, nuestro servidor esta libre de Deadlock.

4. Escalamiento a un millón de usuarios

Si el servidor debe atender concurrentemente a un millón de clientes o más, entonces será necesario realizar modificaciones a este para solucionar los siguientes problemas:

Hay un único proceso aceptando las conexiones de los clientes. Como lo explicamos anteriormente, el thread principal del servidor es el encargado de recibir los pedidos de conexión de los clientes y, a partir de dichos pedidos, lanzar un thread que se encargara de atender a dicho cliente en exclusividad. Este mecanismo se puede convertir en un importante *cuello de botella* del servidor cuando la cantidad de pedidos de conexión crece desmesuradamente, como en el caso de tener un millón de clientes solicitando conectarse a la vez, ya que los clientes deberán esperar mucho tiempo antes de ser atendidos.

Para solucionar este problema se podrían utilizar varios procesos encargados de atender las conexiones de los clientes, en lugar de sólo tener un proceso. Incluso se podría ejecutar a cada uno de estos procesos en una computadora distinta y distribuir las conexiones de los clientes de forma equitativa, para así disminuir el tiempo de espera.

Será necesario ejecutar un millón o mas de threads. Esto es debido a que el servidor lanza un thread por cada cliente que solicite una conexión.

El primer problema que acarrea el hecho de tener tantos threads ejecutándose en el sistema esta relacionado con el scheduling de los mismos. Sin lugar a dudas, el tener que compartir el procesador con un millón de otros procesos hace que el tiempo de espera por cliente y el throughput del sistema se vean afectados. Una forma de mitigar este problema es incrementando la cantidad de núcleos de procesamiento disponibles en el servidor, ya sea con un mejor procesador o utilizando varias computadoras para distribuir el procesamiento. También sería aconsejable evaluar el desempeño del servidor con un número de clientes similar al esperado bajo distintas políticas de scheduling para así elegir la más adecuada en función de la métrica (tiempo de espera, throughput, etc) que se desee priorizar.

El segundo problema que se genera en el servidor está relacionado con la implementación de threads utilizada *POSIX threads*. La cantidad máxima de threads y el tamaño por defecto de la pila de cada thread son dos parámetros muy importantes de dicha implementación.

Por un lado, la cantidad máxima de threads es un limitante directo a la cantidad de clientes que el servidor podrá atender concurrentemente, puesto que cada cliente representa un thread. En particular, en el sistema operativo Linux, el máximo número de threads permitidos (cat /proc/sys/kernel/theads-max) se define, por defecto, en función de la cantidad de memoria disponible (más memoria implica más threads) y el tamaño de la pila que se asigna a cada thread (menor tamaño de pila implica más threads), aunque este valor puede ser modificado *(se necesitan permisos de root)*. También es posible que el sistema operativo posea un límite para la cantidad de threads que un proceso puede crear, en cuyo caso, este valor también debe ser modificado de manera acorde.

Por otro lado, el tamaño de la pila de cada thread es un limitante indirecto, ya que si queremos tener un millón de threads ejecutándose y el tamaño de la pila está configurado en 2 MB (valor por defecto en sistemas de 64 bits) entonces necesitaremos 2 TB de memoria. Claramente, no es necesario poseer 2 TB de memoria RAM ya que también entra en juego la memoria virtual del sistema (swap), pero queda en evidencia la gran importancia que tiene establecer un valor correcto al tamaño de la pila de cada thread y el hecho de que la memoria disponible es un factor limitante. Una solución obvia a estos problemas es la de instalar más memoria en el servidor o utilizar varias computadoras para distribuir los requerimientos de recursos entre cada una. Una solución mas sofisticada implicaría realizar un análisis exhaustivo del uso de la pila que realizan los threads en el sistema para así configurar de manera acorde este valor y, así, hacer un uso correcto de los recursos disponibles.

El tercer problema que se genera esta relacionado con la red que conecta al servidor con los

millones de clientes. Por más grande que sea, el ancho de banda de una red es limitado, lo cual podría afectar la comunicación del servidor con los clientes. Además, los componentes de hardware que componen la red (routers, placas de red, etc) podrían ser un factor limitante en la cantidad de conexiones concurrentes que el sistema soporte. Estos problemas solo es posible solucionarlos obteniendo componentes de hardware que sean capaces de manejar los requisitos de la red del servidor.

Otro problema que se presenta es que cada conexión con un cliente ocupa un file descriptor en el servidor. Los file descriptors no son infinitos y, por lo tanto, este valor debe ser configurado de manera acorde en el sistema.

Los datos del aula serán utilizados por un millón de clientes a la vez. Esto se traduce en que un millón de procesos estarán compartiendo la estructura del aula en memoria generando un problema de contención de recursos y que, por ejemplo, si el aula esta casi o completamente llena entonces la cantidad de clientes (alumnos) que podrán moverse será ínfima en comparación con la cantidad total de clientes (alumnos) en el aula y, dependiendo de la política de scheduling utilizada, el tiempo que tomará que todos salgan del aula puede ser arbitrariamente grande. Esta situación no es un deadlock sino un livelock o también se la puede considerar como una situación de inanición, ya que efectivamente algunos clientes (alumnos) pueden realizar movimientos pero, por las limitaciones de espacio del aula (impuestas por el problema), muchos clientes no podrán realizar acción alguna. Es claro que mientras mas núcleos de procesamiento haya en el servidor más rápido se ejecutarán aquellos clientes que efectivamente pueden moverse, ya que esto aumenta el grado de paralelismo en el servidor. Otra forma de mitigar este problema (en caso de que se lo desee) podría ser implementando en el servidor un algoritmo que detecte estas situaciones, detecte a los clientes que sí pueden moverse y les otorgue una mayor prioridad frente al scheduler para así disminuir el tiempo que les tome a estos salir, dándole a los otros clientes la oportunidad de hacer algo.