Trabajo Práctico 2 - Pthreads
Grupo 10
Aronson, Alex
Ravasi, Nicolás

En este TP debimos crear un servidor que se encargue de manejar diferentes threads que representen a diferentes alumnos tratando de evacuar un aula. Para esto contamos con el código de un servidor que puede manejar solamente un cliente, y debemos a partir de este adaptarlo para cumplir la consigna.

Para esto, lo primero que debemos hacer con dicho código es hacer que soporte varios clientes, esto lo hacemos modificándolo de manera tal que cuando llegue una conexión nueva se lance un nuevo thread. Para esto, en cada pedido definimos un $pthread_t$, en la creación se van a necesitar argumentos que van a ser los argumentos de la función a la cual vamos a hacer apuntar dicho thread para que se ejecute cuando se lance (*atendedor_de_alumno*), así que para esto creamos un struct nuevo (usando malloc, ya que es no es una variable local sino que va a vivir a lo largo de toda la ejecución del thread) que va a tener esos parámetros. Una vez hecho esto podemos usar la funcion *pthread_create* con todos los parámetros para ejecutar el thread.

La nueva función *atendedor_de_alumno* ya no recibe los parámetros anteriores, sino que recibe un puntero a void que debemos encargarnos de castear para que estos sean asignados a las variables correspondientes. Luego de todo el manejo, simplemente usamos *pthread_exit* para terminar el thread.

Ahora bien, todos estos threads se ejecutan simultáneamente, y todos acceden a la misma matriz de posiciones y a los mismos rescatistas, por lo tanto debemos asegurar que el acceso a cada una de estas variables compartidas se realiza concurrentemente y no existen inconsistencias. Para ver la necesidad de implementar mutexes, probamos lanzar el cliente python (que lanza 22 clientes) sin protección en las variables, y restringiendo la capacidad de cada posición a 1, y aún así vimos que los clientes se movían todos a la misma posición sin importar que se excediera el límite. Por lo tanto, era necesario implementar algún método que garantizara que el acceso a variables compartidas fuera exclusivo de un thread. Para esto, contamos con los mutex. Diferenciamos dos, uno para el acceso a la grilla de posiciones, que determina cuánta gente hay en cada posición, y otro para los rescatistas, que permite ver cuántos rescatistas hay disponibles.

Para el primer mutex, *mutexmatriz*, lo iniciamos al principio de la ejecución del servidor (ya que es independiente de los threads), y cada vez que se vaya a acceder a la matriz posiciones, independientemente de si se va a leer o escribir, se debe pedir el mutex. Esto es, al ver si puede moverse, al moverse, al salir del aula y al entrar, se debe garantizar que se tiene el mutex con *pthread_mutex_lock* para poder ejecutar esa sección crítica, devolviendo el mismo con *pthread_mutex_unlock*. Podemos ver que no se3 produce deadlock con este recurso puesto que no se cumple la condición de Coffman de espera circular, puesto que quien toma el mutex hace algunos cálculos y lo devuelve, sabiendo que dichos cálculos no dependen de ninguna otra variable de condición, sabemos que este thread en algún momento termina con su ejecución en la sección crítica y devuelve el mutex sin problemas.

Para los rescatistas es diferente porque si bien la lectura de la cantidad de rescatistas debe requerir un mutex (*mutexrescat*), debemos tener en cuenta que se pueden asignar más de un rescatista (si los hay) al mismo tiempo, por lo tanto el mutex no es suficiente. Para esto, creamos una variable de condición (*vcondresc*). Así, al llegar al punto de salida, el cliente pide el mutex para ver cuántos rescatistas hay disponibles, y se queda ciclando hasta que haya uno. Dentro de este ciclo, se espera por una señal sobre la variable de condición, con *pthread_cond_wait*, cuando se haga un signal sobre la misma, se va a despertar un thread y va a poder seguir ejecutando. Una vez que está garantizado que hay al menos un rescatista disponible (porque no entra en la guarda del while), decrementamos la cantidad de rescatistas disponibles, devolvemos el mutex (puesto que no vamos a tocar la variable, así que dejamos que otros threads la puedan ver, si no hay más rescatistas, tendrán que esperar). Una vez colocada la máscara, pedimos el mutex nuevamente para liberar el rescatista, y luego hacemos un signal sobre la variable de condición para que algún thread que esté esperando (si lo hay) se despierte y pueda ejecutar.

Con todo esto programado, lo único que queda por ver es que realmente estas medidas permiten que los clientes ejecuten sus acciones concurrentemente sin provocar deadlocks ni pisar variables compartidas

indebidamente. Ya dijimos que sin implementar el mutex para la matriz de posiciones, la cantidad de clientes en una posición supera al máximo permitido lo cual es incorrecto, pero al poner el mutex, nunca se sobrepasa dicho máximo, aún cuando este lo bajemos tanto como querramos, lo cual significa que la implementación es correcta.

En el caso de los rescatistas, la forma de testear que funciona correctamente es bien diferente puesto que el script de python no se puede usar para testearlo ya que no hace un switch entre thread y thread cuando uno está esperando al rescatista. Es por esto que tuvimos que lanzar los clientes nosotros mismos simultáneamente desde varias consolas. Dado que como estaba programado no causaba una escasez de rescatistas, ya que la colocación de máscara era inmediata, era necesario imponer un delay que provocara que para un determinado momento existieran más alumnos esperando ser salvados que rescatistas disponibles para atenderlos.

Para esto, consideramos la función *usleep* que permite poner a dormir el thread por cierto intervalo de tiempo, es importante remarcar que usamos *usleep* y no *sleep* porque esta última pondría a dormir a todo el proceso y no al thread, lo cual no nos serviría de nada para testear. Con un *usleep* de 10 segundos, que lo seteamos sólo para algunos clientes (aquellos cuyo nombre empieza con A), de manera tal que sean estos los que demoran el rescate, podemos ver que no se produce deadlock, esto se evidencia por el hecho de que no se cumple la condición de Coffman de *holdandwait*, ya que cuando un thread toma el mutex de rescatistas, si hay disponibles, toma un rescatista y devuelve el mutex, tomándolo de vuelta para liberar al mismo; en caso de no haber rescatistas disponibles, se queda esperando sobre la variable de condición pero devuelve el mutex, por lo tanto no puede pasar que un thread se quede esperando por un recurso mientras mantiene otro en su haber.