Sistemas Operativos

Trabajo Práctico Nº2

Grupo 20

Daniel Grosso	694/08	dgrosso@gmail.com
Mariano De Sousa Bispo	389/08	marian_sabianaa@hotmail.com

Noviembre 2010

1. Explicación servidor_multi

Para realizar el `servidor_multi` partimos del código del `servidor_mono`. `A` diferencia con el `servidor_mono`, para poder atender a varios clientes en simultáneo, el `el servidor_multi` lanza un *thread* por cada cliente atendido. Cada *thread* ejecuta la función `atendedor_de_alumno` la cual representa al rescatista asignado a cada alumno. Para crear el *thread* se llama a la función `pthread_create` pasándole una estructura inicializada con los datos que necesita `atendedor_de_alumno` (*file descriptor* del alumno y un puntero al aula). La función `atendedor_de_alumno` es exactamente igual a la misma función del `servidor_mono`. Los únicos cambios fueron introducidos en la función `intentar_moverse`. Ésta verifica que la petición a la que se quiere mover el cliente sea válida. En caso de ser válida, mueve a la persona de un casillero al otro, devolviendo el valor de verdadero.

Cada metro cuadrado del aula, es representando como una posición en una matriz, en esta, se guarda la cantidad de personas que se encuentran en ese lugar. Se deduce de esta estructura, que sólo dos posiciones por cada alumno son potencialmente modificables en cada pedido. Al poder moverse varios clientes en simultáneo, no se puede asegurar que no exista dos o más que deseen realizar operaciones sobre alguna posición particular de la matriz. Para solucionar este problema, hace falta bloquear tanto lectura como escritura de las celdas intervinientes en el movimientos, las cuales potencialmente, pueden ser todas. Hemos encontrado dos soluciones para este problema: la primera es tener un *mutex* que restrinja el acceso a toda la matriz, es decir, sólo un *thread* puede accederla a la vez. La segunda, consiste en tener un *mutex* por cada celda, y que el *thread* bloquee sólo las posiciones que utiliza (posición actual, posición destino). Esta última opción nos pareció la más adecuada, ya que a pesar de consumir una cantidad mayor de memoria, posibilita el procesamiento en paralelo y no bloquea la ejecución de los otros hilos si se reciben pedidos que involucran celdas disjuntas.

Cada *thread* entonces, inicialmente utilizará en cada pedido del cliente, dos *mutex*, que luego de utilizarlos los libera. Esta solución no es libre de *deadlock*. Supongamos que tenemos dos clientes `A` y `B` en celdas consecutivas. El cliente `A` pide moverse a la posición de `B`, y viceversa. Supongamos además que `A` se atiende primero, y según nuestro algoritmo, se bloquea la posición de `A`, seguido de esto, `B` es atendido y logra hacer un bloqueo de su posición actual. `A` entonces no podrá acceder a la posición de `B` ya que esta bloqueada, y a su vez, `B` no podrá acceder a la posición de `A` ya que también lo está.

Para solucionar este problema, forzamos mediante un nuevo *mutex* a realizar ambos *locks* de manera atómica. Esta nueva sección crítica comienza antes del primer *lock* y termina después del segundo. Cada *lock* fue estratégicamente ubicado de forma tal que no bloqueara posiciones que no utilizará luego.

En conclusión, cada movimiento de los clientes, puede realizarse en simultáneo, con la excepción de que ambos deseen moverse y/o estén en una misma posición.

Otro problema a solucionar consistía en la asignación de un rescatista a cada alumno para que le coloque la máscara. La cantidad de rescatistas es limitada, por lo tanto es necesario llevar la cuenta de cuántos rescatistas se disponen en el momento que sale del aula cada alumno. Como varios alumnos (*threads*) podrían pedir un rescatista disponible al mismo tiempo, se podroduce una condición de carrera. Esta condición de carrera se controla mediante una variable de condición y un *mutex* asociado a dicha variable. Cuando cada alumno sale del aula, espera (mediante `wait`) a que no haya ningún otro alumno pidiendo un rescatista. Al pedir el rescatista (el *thread* asociado al alumno decrementa la variable compartida que cuenta los rescatistas disponibles), avisa (mediante `signal`) que ya fue asignado un rescatista para colocarle la máscara. De esta manera, los rescatistas se asignan de forma ordenada.