DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo práctico 2

Sistemas Operativos


| Integrante | LU | Correo electrónico |
|---|---|---|
| Fernando Gasperi Jabalera | 56/09 | fgasperijabalera@gmail.com |
| Martín Matías Monti | 727/10 | martinmatiasmonti@gmail.com |
| Diego Alejandro Amil | 68/09 | amildie@gmail.com |






**Facultad de Ciencias Exactas y Naturales**
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

# Índice

# 1. Implementación

En el presente trabajo modificamos un servidor que operaba con un único thread para que utilice múltiples. El objetivo es conseguir que el servidor atienda los pedidos de manera concurrente para poder realizar el trabajo requerido en menos tiempo. Además, si se contara con un hardware apropiado podremos ejecutar las tareas de forma paralela aumentando aún más la performance del servidor.

## 1.1. Mono a Multi

El trabajo particular de este servidor consiste en recibir conexiones de clientes, aceptarlas y luego procesar sus peticiones hasta que se desconecten. La función que se encargaba de procesar las peticiones de los clientes se llama *atendedor_de_alumno* y en la implementación original era llamada, una vez que la conexión había sido aceptada, con el file descriptor correspondiente al cliente. Lo que hicimos fue implementar el patrón de concurrencia *thread pool* que cuenta con un thread que se encarga de distribuir el trabajo y otros threads, comúnmente denominados *workers*, sobre los cuales delega las tareas que hay que procesar. Por lo tanto, el esquema final se compone por un lado de el thread principal que recibe las conexiones y las acepta, esta parte no sufrió modificación alguna, y por otro, nuestro agregado, la generación de un nuevo *worker* thread, por cada alumno que acepta, en el cual delega la función *atendedor_de_alumno* que es la que procesa todas las peticiones. La generación de los thread no nos presentó inconvenientes, simplemente tuvimos que agregar un struct *thdata* para poder pasar los argumentos que recibe *atendedor_de_alumno*.

## 1.2. Sincronización

Como pasamos de tener un thread a tener uno por cada alumno que se conecta al servidor, más el servidor principal que genera los *workers*, necesitamos implementar políticas sincronización porque todos van a estar leyendo y escribiendo la misma memoria, ya que como son threads la memoria es compartida.

### 1.2.1. Básicos

Las variable *cantidad_de_personas* pertenece al struct que representa al aula y nos dice la cantidad de personas que se encuentran dentro del aula. La variable es inicializada en cero porque en un principio el aula está vacía y es modificada cuando un alumno ingresa en el aula, mediante la función *t_aula_ingresar*, y cuando alcanza la salida. Cuando ingresa el alumno la variable se incrementa en uno y cuando alcanza la salida se decrementa en uno, en la función *t_aula_liberar*. Nos bastó utilizar un mutex para asegurarnos que el acceso a esa variable sea exclusivo.

### 1.2.2. Aula

El aula está representada por una matriz de enteros. Cada posición de la matriz se corresponde con una posición posible para los alumnos dentro del aula y el valor de la posición representa la cantidad de alumnos en esa ubicación. Como el aula comienza vacía todas las

posiciones de la matriz son inicializadas en cero. Ésta matriz será modificada por todos los alumnos ya que cada vez que quieran moverse de una ubicación a otra necesitan decrementar el valor de la celda en la que se encontraban e incrementar el valor de la celda a la cual se desplazaron, sólo si el desplazamiento solicitado por el alumno haya sido posible dadas las dimensiones del aula y el umbral de capacidad máxima de personas con el que cuentan las posiciones del aula. Como queremos asegurarnos que la posibilidad de concurrencia sea máxima debemos pedir acceso exclusivo a la estructura más pequeña posible. Lo mínimo que necesitamos es la celda correspondiente a la casilla en la que nos encontramos actualmente y la celda a la casilla a la que nos queremos desplazar. Para asegurarnos que no haya deadlock romperemos una de las condiciones necesarias de Coffman: la espera circular. La romperemos asegurándonos que los accesos exclusivos sobre las celdas siempre sean solicitados en el mismo orden. Nosotro agregamos al struct *t_aula* una matriz *locks* de igual tamaño que *posiciones* pero todas con un mutex. De esta forma tenemos un mutex por cada celda de *posiciones*. El orden que impondremos será que siempre se solicite primero la de menor fila y ante empates la de menor columna. Lo explicaremos con un ejemplo. Supongamos que hay dos alumnos que quieren operar sobre las dos mismas celdas $(3, 5)$ y $(4, 5)$. Como la primer celda cuenta con una fila menor a la segunda, $3 < 4$, entonces los dos solicitarán primero la celda $(3, 5)$. El código que toma esta decisión se encuentra dentro de la función *intentar_moverse*.

### 1.2.3. Rescatistas

La administración de los rescatistas fue la primer tarea que requirió la utilización de variables de condición. Cuando los alumnos salen del aula, antes de intentar conformar un grupo de 5 para emprender la salida final, un rescatista debe colocarles una máscara. La máscara es colocada por la función llamada *colocar_mascara*. Se cuenta con un número limitado de rescatistas para colocarles las máscaras a los alumnos, por lo tanto, si un alumno logró salir del aula y necesita que le coloquen la máscara debe esperar a que haya al menos un rescatista disponible. Una vez que lo consiga éste debe colocarle la máscara y luego queda libre para continuar atendiendo a otros alumnos. Para lograr esto utilizamos tres variables globales:

**rescatistas** variable que representa la cantidad de rescatistas disponibles. Es inicializada con la cantidad total de rescatistas.

**cv_rescatistas** variable de condición asociada a la variable rescatistas.

**mutex_rescatistas** mutex asociado a la variable de condición de la variable rescatistas.

```
pthread_mutex_lock(&mutex_rescatistas);
while (rescatistas == 0) {
  pthread_cond_wait(&cv_rescatistas, &mutex_rescatistas);
}
--rescatistas;
pthread_mutex_unlock(&mutex_rescatistas);

alumno->tiene_mascara = true;

pthread_mutex_lock(&mutex_rescatistas);
++rescatistas;
pthread_mutex_unlock(&mutex_rescatistas);

pthread_cond_signal(&cv_rescatistas);
```

Primero esperamos a que la cantidad de rescatistas sea mayor a cero. Se nos notificará de cualquier cambio en la variable rescatistas porque se puede ver que lo último que realiza la función, luego de volver a incrementar la variable rescatistas, es enviar una señal a la variable de condición. Una vez que sabemos que hay al menos un rescatista disponible y tenemos el mutex tomado decrementamos la variable rescatista para indicar que un rescatista estará ocupándose de nosotros. Luego de que nos coloca la máscara incrementamos nuevamente la variable rescatistas para indicar que el rescatista está una vez más disponible y enviamos una señal a la variable de condición para notificar del cambio.

### 1.2.4. Grupo de salida

La formación el grupo de salida es realizada dentro de la función *t_aula_liberar*. En primer lugar actualizamos las variables *cantidad_de_personas*, la cual decrementamos porque el alumno al haber salido del aula ya no se encuentra en ella, y *afuera*, una variable global que utilizamos para indicar la cantidad de alumnos que se encuentran fuera del aula pero todavía no completaron su evacuación, es incrementada. Luego utilizamos 3 variables globales para coordinar la conformación de los grupos de salida:

**grupo_de_salida** representa la cantidad de personas en el grupo que se está formando para salir. Son alumnos a los que ya se les colocó la máscara y sólo están esperando que se termine de formar el grupo de salida. Tiene un mutex y una variable de condición asociados.

**salieron** representa la cantidad de personas que salieron del grupo formado. Tiene un mutex asociado.

**hay_grupo_de_salida** que indica si el grupo el grupo de salida que está formado está saliendo o todavía está esperando. También amparada por el mutex de *grupo_de_salida*.

```
pthread_mutex_lock(&mutex_grupo_de_salida);
while(hay_grupo_de_salida) {
  pthread_cond_wait(&cv_grupo_de_salida, &mutex_grupo_de_salida);
}
```

En esta primera parte nos cercioramos que el grupo de salida no esté saliendo. Si el grupo está saliendo esperamos a que termine de salir para poder sumarnos al próximo que se conforme.

```
++grupo_de_salida;
bool somos_los_ultimos = (un_aula->cantidad_de_personas == 0) &&
  (grupo_de_salida == afuera);
bool grupo_lleno = (grupo_de_salida == 5);
bool soy_el_ultimo = grupo_lleno || somos_los_ultimos;
while (grupo_de_salida < 5 &&
    (un_aula->cantidad_de_personas > 0 || grupo_de_salida < afuera)
    && !hay_grupo_de_salida) {
  pthread_cond_wait(&cv_grupo_de_salida, &mutex_grupo_de_salida);
}
```

En esta segunda parte primero incrementamos *grupo_de_salida* ya que acabamos de ingresar al mismo. Luego vamos a esperar a que el grupo de salida se complete, lo cual puede suceder por dos causas:

1. la cantidad de personas en el grupo de salida llega a 5.
2. el aula está vacía y todos los que estamos afuera ya ingresamos en el grupo de salida.

El último que llegue al grupo y verifique que las condiciones se cumplen para puedan completar la evacuación será el encargado de avisarle al resto con un *broadcast*. Aquí se aprecia la importancia de contar con la variable *hay_grupo_de_salida*, a la cual se le asigna verdadero una vez que una persona entra al grupo y las condiciones se cumplen para que evacúen. Si no contáramos con esta variable podría suceder lo siguiente: si el grupo estuviera listo para evacuar porque no hay más personas en el aula y todos los que están afuera pertenecen al grupo de salida pero no llegan a sumar 5 personas, el que sale del while y va a dar el broadcast puede ser desalojado y mientras tanto entrar muchas personas al aula. En ese caso cuando los que recibieron el broadcast se despierten van a evaluar nuevamente la guarda del while y ver que se cumple por lo cual van a volver a dormir. Sin embargo, como además el último asigna verdadero a la variable *hay_grupo_de_salida* ya queda determinado que todos los que estaban en el grupo hasta el momento van a salir ahora. Con esta variable en verdadero todos los que estaban en el grupo se despiertan y efectivamente salen del while pero ningún otro va a poder entrar al grupo porque necesitan que sea falsa. De esta forma la consistencia prevalece.

```
pthread_mutex_lock(&mutex_salieron);
++salieron;
if (salieron == grupo_de_salida) {
  salieron = 0;
  pthread_mutex_lock(&mutex_grupo_de_salida);
    grupo_de_salida = 0;
    hay_grupo_de_salida = false;
  pthread_mutex_unlock(&mutex_grupo_de_salida);
  pthread_cond_signal(&cv_grupo_de_salida);
}
pthread_mutex_unlock(&mutex_salieron);
```

Finalmente, resta ver que los que pertenecían al grupo salgan y el último en salir, aquel que se encuentre que los que salieron son un cantidad exactamente igual al cardinal del grupo de salida, será el encargado de restablecer el valor de todas las variables utilizadas: *grupo_de_salida* y *salieron* en 0 y *hay_grupo_de_salida* en falso.

### 1.3. Experimentación

Utilizamos el script provisto por la cátedra, *server_tester*, para ver el comportamiento de nuestro servidor. Lo único que modificamos en el mismo durante las experimentaciones fue la cantidad de clientes y el retardo entre peticiones consecutivas, tiene un sleep luego de cada request. Primero, probamos que corra exitosamente con 1 y hasta 5 *server_testers* simultáneamente realizándole peticiones. Todos los alumnos evacuan sin problemas y los *server_tester* finalizan sin inconvenientes. Además, probamos reducir el tiempo que esperaba y aumentar la cantidad de *server_tester* utilizados para ver si efectivamente se comportaba correctamente en situaciones donde tuviera muchas peticiones por segundo. En particular, nos interesaba ver si las impresiones por pantalla que agregamos a las funciones del servidor lograban superponerse entre diferentes threads, lo cual sería una muestra bastante convincente de la concurrencia alcanzada. Pudimos lograrlo con 7 *server_tester* y hasta con corridas en las que no reducimos el sleep de los mismos, inclusive vimos solapamiento de las impresiones entre threads en la función *t_aula_liberar*, que es la que cuenta con una sincronización más complicada, y en todos los casos las ejecuciones fueron exitosas. La única cuenta pendiente que nos quedó fue experimentar más precisamente situaciones en las que hay muchas personas cerca de la salida, afuera, formando el grupo de salidaa y al mismo tiempo muchas personas entrando al aula, lo cual era uno de los casos más importantes de la función *t_aula_liberar*.

## 2. Escalamiento

Si el número de clientes aumenta a $10^6$ lo primero que debemos preguntarnos es cuáles son las implicancias de esto en nuestro servidor, es decir, qué va a aumentar de nuestro lado. Lo que aumentará será:

1. número de sockets utilizados. Cada cliente utiliza un socket diferente para recibir las respuestas del servidor.
2. número de threads utilizados. Las peticiones de cada cliente son procesadas por un thread dedicado únicamente para ese cliente.
3. cantidad de memoria utilizada. Si bien la memoria es compartida entre threads el stack no. Cada thread utiliza poca memoria pero con $10^6$ stack frames no importa cuan pequeños sean existe la posibilidad de que se produzca un stack overflow.

El número de sockets y de threads disponibles en general están limitados por el sistema operativo. Dependiendo el sistema operativo en el que corramos nuestro servidor tenemos que verificar la máxima cantidad de threads que soporta y de sockets. De no soportar los necesarios podemos cambiar de sistema operativo o crecer horizontalmente agregando nuevas máquinas que corran el servidor y una principal que sólo se dedique a recibir las conexiones y balancear la caraga entre las demás. Veamos que una vez más se repite el mismo esquema que planteamos entre el thread principal y los workers sobre los cuales delegaba el trabajo.

Pero hay que tener en cuenta que esto implicaría cambiar por completo la implementación porque pasaríamos a contar con un sistema distribuido lo cual requiere nuevas formas de comunicación. La memoria principal de las máquinas puede ser expandidad comprando nueva. Sin embargo, cabe mencionar que el tamaño del aula debería reconsiderarse ya que si el aula es demasiado pequeña y todos los clientes esperados se conectan simultáneamente puede darse el caso de que todas las posiciones de la misma se saturen y ninguna persona pueda moverse por el límite que tiene cada celda.

Hay elementos que no aumentan y sin embargo, es necesario incrementar su número ya que si no lo hacemos pueden convertirse en un cuello de botella, es el caso del puerto utilizado por el servidor para recibir las conexiones. Si sólo contamos con un puerto de escucha la cola de espera puede volverse arbitrariamente larga. Aumentando el número de puertos disponibles para recibir conexiones la cola de espera se reduciría exactamente en $\frac{encolados}{colas}$ aproximadamente asumiendo que implementamos alguna política de balanceo de carga.