

Filesystems, IPCs y Servidores Concurrentes

September 12, 2011

TP especial SO - 1

Integrantes:

1. Castiglione, Gonzalo - 49138
2. Gomez, Horacio - 50825
3. Orsay, Juan Pablo - 49373

1 Resumen

Para este trabajo se pedia la realizacion de una simulacion de empresas, la cuales tienen aviones a su mando, encargados de distribuir sus medicamentos por una serie de ciudades con conecciones entre ellas limitadas.

2 Modelo

Para llegar a la estructura de la simulacion actual, primero se estudio la diferencia entre procesos y threads. Los threads presentaban la ventaja de ser *light weight* y que no necesitan comunicacion mediante un *IPC* para pasarse informacion dado que corren en la misma zona de codigo. Mientras que los procesos realizan una copia de todas sus variables cuando son creados y corren independientes de los cambios que el proceso padre haga. Teniendo estos detalles en cuenta, se decidio que cada *Compania* debia ser un proceso, ya que no tiene porque compartir su informacion con nadie mas. Y, dado que los aviones comparten todo con su empresa, nos resulto conveniente optar por *Threads* para estos.

Al tomar este tipo de estructura, surgio el problema sobre tipos de mensajes entre estos procesos y cual era la forma optima de comunicarlos dado que el pasaje de informacion es mucho mas complejo que la comunicacion entre threads, ya que requiere tanto sincronizacion como una zona de memoria ya preparada para la lectura y escritura.

Tomada esta decision de diseño, se debia resolver ademas el problema de:

1. Mostrar en pantalla los cambios de cada *Compania* luego de cada turno.
2. Reflejar en las demas companias los cambios producidos en el mapa por la compaia *X* antes que la compania *Y* intente hacer un cambio sin haber recibido esta notificacion.

La primer solucion propuesta fue la de crear un zona de memoria compartida la que involucraria tener en cuenta los siguientes aspectos:

- Todo aquel proceso que quiera modificar esta zona de memoria, tendria que hacerlo siempre bloqueando previamente un mutex (o semaforo en su defecto) y luego de realizados los cambios liberar el mutex. Lo cual obligaria a los demas procesos a “esperar” en una cola a acceder a esta memoria. El problema con esta implementacion es que si se hubiese implementado, no se se hubiese respetando la consigna de usar los diferentes tipos de ipcs para la comunicacion entre procesos. Sin embargo es una solucion agradable ya que evitaba el tener que andar notificando a otras companias mediante paquetes especiales sobre cambios realizados.

La segunda solucion propuesta involucraba un proceso *servidor* que se ecar-garia de la administracion de turnos y recursos para cada compania.

- Este presentaba la ventaja de tener una implementacion muy sencilla ya que solamente se ocuparia de levantar semaforos y bajarlos para que las companias toquen el mapa en forma sincronizad y ademas asegurar que ninguna valla a jugar dos veces seguidas (luego veremos que, como en toda buena idea, trajo sus complicaciones).

- A su vez se podía saber cuando todas las compañías habían terminado un turno; por lo que actualizar la UI por turno resultaría muy sencillo.
- La única contra que encontramos es que la comunicación con las compañías se volvía un tanto compleja.

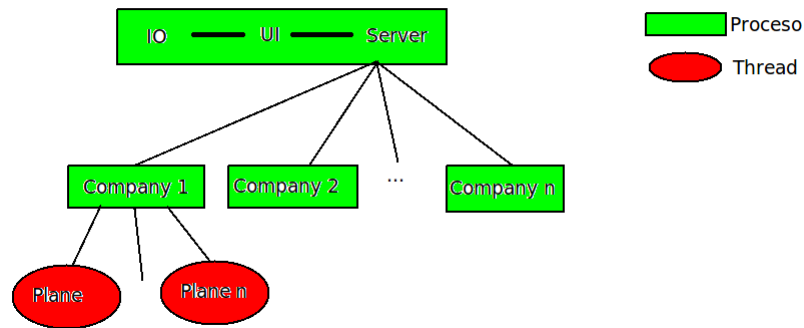


Figure 1: Estructura de procesos y threads de la simulación

Al iniciar el programa, lo primero que se realiza es el parseo de la información (de compañías, mapas), inicialización del entorno (IPCs, semaforos, handlers de las señales, inicialización de los nuevos procesos...) y luego de terminado todo esto, se procede a inicializar el proceso de UI/Servidor.

Uno de los primeros (y mas grandes) problemas que se presentaron al aplicar este diseño era de como reflejar los cambios hechos por una compañía en todas las demás. Inicialmente, se decidió que el servidor tendría una (y la única) instancia del mapa y que se pasaría al principio del turno a la compañía, esta lo modificaría y luego se lo pasaría nuevamente al servidor con los cambios. Y así para cada compañía.

Pero esta solución era muy costosa, ya que además de tener que mandar el mapa, se debía mostrar por pantalla (en forma ordenada) toda la información de la empresa; por lo que además de tener que pasar el mapa dos veces, se tendría que sumar toda la compañía, lo cual implicaría muchísimo procesamiento y uso de memoria que podría ser ahorrado!.

Lo que nos llevo a proponer una segunda solución alternativa, la cual implicaba que tanto el servidor como las compañías tendrían una instancia del mapa(inicial) y este se iría actualizando mediante paquetes “*updates*” que se enviarían desde el servidor luego de cada turno.

Esto presenta la ventaja que la comunicación entre los procesos se reduciría a únicamente sus cambios! pero la contra esta en que requería de una clase *serializer* muchísimo mas completa que la que se tenía en mente. Sin embargo, luego de discutirlo se llego a que se esta era la mejor implementación ¹

¹No se presentan calculos de cuanto mas se mejora dado que es muy facil notar la cantidad

Teniendo en cuenta todas estas consideraciones, la logica del *servidor* seguiria el siguiente comportamiento:

Por cada compania:

- Se le da un turno.
- Se espera a que finalice su turno.
- Se hace un broadcasting de los updates leidos a todas las demas companias.

Actualizar UI.

De esta manera, resulta facil imaginar la logica de una *compania*:

- Despierto a todos mis aviones.
- Cuando todos movieron, actualizo a un nuevo target a todos aquellos que llegaron a destino.
- Escribo los paquetes con los cambios realizados por cada uno al servidor.
- Me escribo a mi misma.

A continuacion se presenta un esquema de como se penso a una compania:

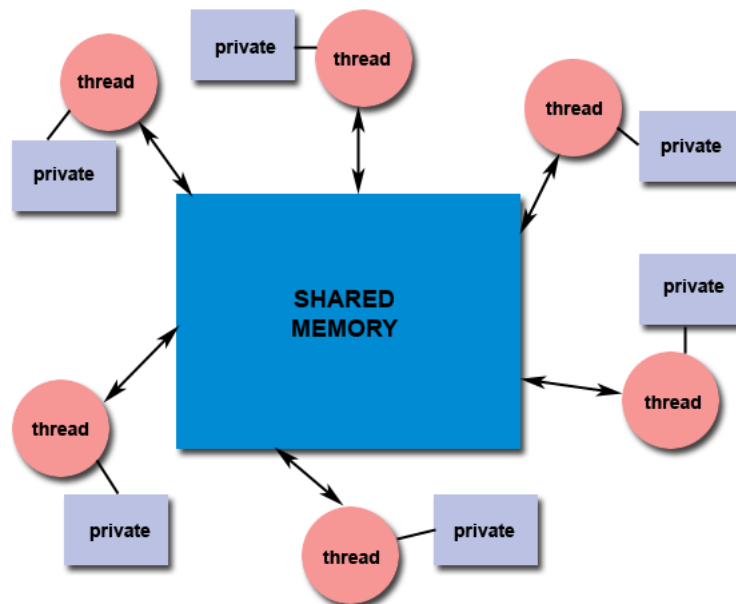


Figure 2: Esquma de una compania
(figura tomada del sitio: <http://www.chuidiang.com>)

de infromacion que se ahorra por cada envio de informacion

Tal como se explico anteriormente, cada *Thread* representa a un avion activo, en donde este tiene su memoria propia (items, posicion, target, ...), una memoria compartida (el mapa) y un comportamiento. Cuando un avion no tiene movimientos posibles, se mata al proceso, y este ciclo sigue hasta que ningun avion tenga movimientos disponibles. En cuyo caso se envia un paquete de tipo *company status update* al servidor y se encargara de hacer lo que sea necesario.

3 IPC

Al principio se comenzo experimentando comunicacion entre procesos con pipes. Luego de varias complicaciones de llego a una primera version de metodos de IPC formada por diversos metodos. Esta nos sirvio como base para ir experimentando como funcionaban los semaforos y mutex en linux. Luego, a medida que el programa iba tomando forma y sentido, se comenzo a mejorarla y se implemento Fifos, MsgQueues, Sockets y Shared Memory.

Uno de los mayores inconvenientes al momento de implementacion de los IPCs era la sincronizacion, ya que, por mas pruebas que se les halla a cada antes de ser montados al codigo de la simulacion, se tenia el problema que cuando no se leia en forma ordenada, este decia que no habia mensajes cuando se espereaba que los hubiera, lo que llevaba a mucha perdida de tiempo en debuggeo de codigo.

Una de las complicaciones que tuvimos con los IPCs era el tamaño de los datos a escribir, dado que para algunos IPCs era mas facil trabajar con tamaños fijos y con otros daba igual, se opto por paquetes de tamaño fijo. Esto facilito mucho la programacion de la interfaz de comunicacion, pero tiene el problema importante que si se quiere serializar una gran compania con muchos aviones e items, puede resultar que este string serializado resultante sea mas largo que el tamaño default. Lo que provocaria que el paquete no se mande y quede inconsistente el resultado de la simulacion.

De todas maneras, si esto llegase a suceder, un mensade de error se mostrara en el *log.txt*. En cuyo caso lo unico que se tiene que arreglar es el *DATA_SIZE* definido en *communicator.h*.

Otro inconveniente que trae esta decicion que tamaño fijo para mensajes es el desperdicio de memoria innecesario por ejemplo para mandar un pequeño update. Sin embargo, no es un gran problema, si el dia de mañana se necesitase que esto no sea asi, bastara con simplemente cambiar la implementacion del IPC correspondiente (no afectaria de ninguna manera a la simulacion).

3.1 Shared Memory

La implementacion que actualmente se tiene para shared memory, es aprovechar el hecho que los tamaños de los mensajes son fijos y reservar en memoria bloques con un tamaño de M mensajes por cada proceso que quiera escribir. Es decir, dividimos un pedazo fijo de memoria en $N \times N$ lugares, en donde N esta dado por la cantidad de luagres especificados en *ipc_init*. Esta implementacion ofrece la ventaja de ser muy eficiente en cuando acceso a un mensaje, ya que tiene una complejiad $o(1)$, pero la gran contra es que es mucha la memoria que necesita tener reservada.

Dentro de cada lugar de esta matriz, se tiene un array con los M luagres ya mencionados y adelante de todo un char indicando cuantos se tienen en la lista.

3.2 Tipos de paquetes

Actualmente la comunicacion entre *servidor* \iff *compania* se realiza mediante tres tipos de paquetes:

- City update: Este paquete contiene informacion de que item fue modificado en que ciudad y en que cantidad. Si muchas ciudades fueron modificadas en un turno, uno por cada cambio va a ser enviado al servidor.
- City status update: Este paquete se envia cuando una compania detecto que sus aviones no pueden abastecer a mas ciudades y quiere “darse de baja” en el servidor. Es importante esto ya que, una vez que una compania se desactiva en el servidor, automaticamente se deja de encolarle actualizaciones. Lo que resulta es un ahorro importante en llamadas al serializer y de uso de recursos del IPC para mantener mensajes que nunca son leidos.
- Company update: este es el paquete mas grande que se tiene, aqui es donde se guarda la informacion de todos los aviones que tiene una compania. El problema es que crece mucho a medida que se agregan mas items y/o aviones la misma. Por lo que quedaria en el “*wish list*”, partir este paquete en otros mas pequenos y en lo posible con tamaño constante, para de esta manera poder reducir el DATA_SIZE que se necesita tener reservado para usar el IPC. Que en estos momentos seria lo que mas recursos consume de la simulacion. Se tenia pensando para estos, realizar lo mismo que para las ciudades, solamente mandar pequenos paquetes updates en vez de repetir una y otra vez la misma informacion que no sufrio cambios.

4 Otras consideraciones

Uno de los problemas mas dificiles que constantemente se presentaba durante el desarrollo del trabajo era sobre la dura eleccion entre *tiempo vs memoria*. Si se debia hacer cierto calculo al momento de neceitarlo o bien guardarlo previamente y simplemente actualizarlo cuando sea necsario.

Al principio, cada vez que un avion llegaba a una ciudad, se realizaba un DFS para encontrar el camino a las ciudad mas corta que sea abastecible. El problema es que este mismo DFS se estaba realizando por cada avion, por cada compania en cada turno. Lo que nos parecia que se realizaba el mismo calculo innecesariamente muchisimas veces. Por lo que se propuso una matriz en donde, por cada compania se colocaria un indice diciendo a donde hay que ir para acceder a la ciudad X (para cualquier ciudad) y a que distancia se encuentra. Esto reduciria la llamda al *DFS* a desrefenciar una matriz ($o(1)$), esta fue una de las luchas mencionadas anteriormente, sobre la utiliacion de memoria espacial o de tiempo².

Otro cambio de la misma indole que se propuso durante la realizcion del trabajo es que cada ID de un elemento representaba a la pocision en el array que ocuparia. Es decir, el elemento con Id 5, se lo encontraria en el array de elementos en la quinta posicion. Esto trae consecuencias como tener un array de 20 elementos solo para tener un elemento con ID 20. Sin embargo, para la busqueda de items (la cual se reliza constantemente) reduce un algortimo de $o(n)$ a uno de $o(1)$ con el precia de agregar un extra en la cantidad de memoria necesaria para su funcionamiento.

4.1 Otras problemas

Un problema al que todavia no pudimos encontrar explicacion es porque los semaforos de *System V* a veces no se inicializaban para ciertos valores de claves. Se intento eliminarlos, cambiar los flags de creacion, reiniciar la comutadora pero todo sin suerte! Mas raro resultaba aun que al sumarle un magic number al key, estos se creban y sin problema. Por lo que nuestra unica alternativa fue la de pasar a semaforos de Posix. Los cuales resultaron increiblemente mas simples. Lo mismo sucede a veces con la creacion de los *IPC*s, que a veces no logran cerrarse correctamente al finalizar el programa o bien el recuso del SO se encuentra ocupado y se rehusa a crear los IPS terminando en resultados sin sentido, por lo que se recomienda que si esto sucede se utilize el comando *ipcs* para verificar que no se tenga una excesiva cantidad de estos andando (se soluciona tambien reiniciando ya que para todo tipo de almacenamiento temporal, se utiliza la carpeta */tmp* del SO).

Se queria aclarar que si se testea la simulacion de archivos que tienen formatos invalidos, el programa resultara en *segmentation fault*. Esto no fue arreglado ya que supusimos que lo importante de este trabajo practico no era ver quien hacia el parser mas robusto sino el de demostrar el poder del uso de

²Esta razon de necesitar implementar un DFS surgio en casos cuando solo se podia abastecer a uan ciudad que no era adyacente a la posicion actual del avion.

IPCs. Por lo que se recomienda verificar bien el formato de los archivos antes de iniciar el programa.

5 Conclusiones

Luego de realizado este trabajo practico, se aprendio sobre la potencia a la que se puede llevar un programa al hacerlos que sean capaces de dividir sus tareas en proectos concretos y por sobre todo, al mantenerlos sincronizados.