TP especial SO - 1

September 12, 2011

Part I
Resumen

Para este trabajo se pedia la realizacion de una simulacion de empresas, la cuales tienenaviones a su mando, encanrgados de distribuir sus medicamentos por una seria de ciudades con conecciones entre ellas limitadas.

Part II
Modelo

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 y simplemente actualizarlo cunado se necesite. (listar algunos ejemplos y soluicones como ser el parseo con memoria estatica, pre calculo de disancias antes de la simulacion...))

Otro problema que se presento era sobre que tipo de mesnajes y como debian comuincarse los procesos dado que la comunicacion entre procesos es mucho mas costosa que la comunicacion entre threads, pero al usar procesos se puede aprovechar de tener espacios de memorias separadas. Y es por esto es que nuestro dieño utiliza a las *Companias* como procesos y cada una tiene n threads (una por cada avion).

Tomada esta decision, el problema se presentaba ahora en :

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

- La primer solucion era 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.
- La segunda solucion involucaraba un proceso *servidor* que para 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 sincronizadamente y ademas asegurar que ninguna valla a jugar dos veces seguidas (luego veremos que, como en toda buena idea, trajo sus buenas complicaciones).
  - A su vez se podia saber cuando todas las companias habian hecho una jugada, por lo que actualizar la UI por turno era muy sencilla.

– La unica contra que encontramos es que la comunicacion con las companias se volvia un tanto compleja.

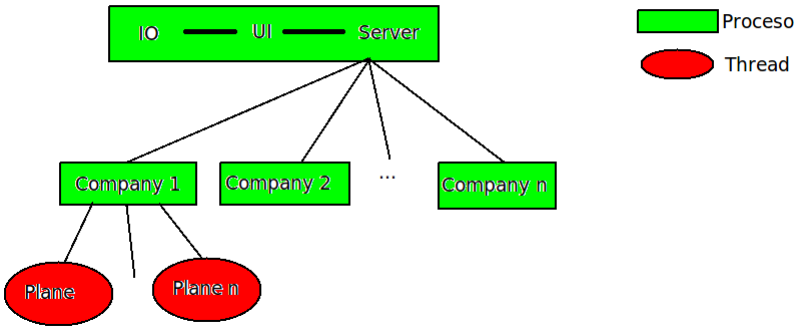

Figure 1: Estructura de procesos activos de la simulacion

Uno de los primeros (y mas grandes) problemas que que se presentaron al aplicar este diseño era de como reflejar los cambios hechos por una compania en todas las demas. Inicialmente, se decidio que el servidor tendria una (y la unica) instancia del mapa y que se *pasaria* al principio del turno a la compania, esta lo modificaria y luego se lo *pasaria* de vuelta al servidor con los cambios. Y asi para cada compania. Pero esto no solucionaba el problema, ya que ademas se debia mostrar por pantalla (en forma ordenada) toda la informacion de la empresa; por lo que ademas de tener que pasar el mapa dos veces, se tendria que sumar toda la compania, lo cual implicaria muchisimo procesamiento y uso de memoria!.

Lo que nos llevo a proponer una segunda solucion; esta implicaba que tanto el servidor como las companias tendrian una instancia del mapa (inicial) y este se iria actualizando mediante paquetes "*updates*" que se enviarian desde el servidor. Esto presenta la ventaja que la comunicacion entre los procesos se reduciria a unicamente sus cambios! pero la contra esta en que requeria de una clase *serializer* muchisimo mas completa. Sin embargo, luego de algunas pruebas y de discutirlo se llego a que se esta era la mejor implementacion. (MOSTRAR ALGUNOS RESULTADOS AQUII).

Por lo que con esta eleccion, la logica del *servidor* seguiria el siguiente comportamiento:

```
Por cada compania:

    Se le da un turno.
    Se hace un broadcasting de los updates enviados por el a
       todas las demas companias.
```

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

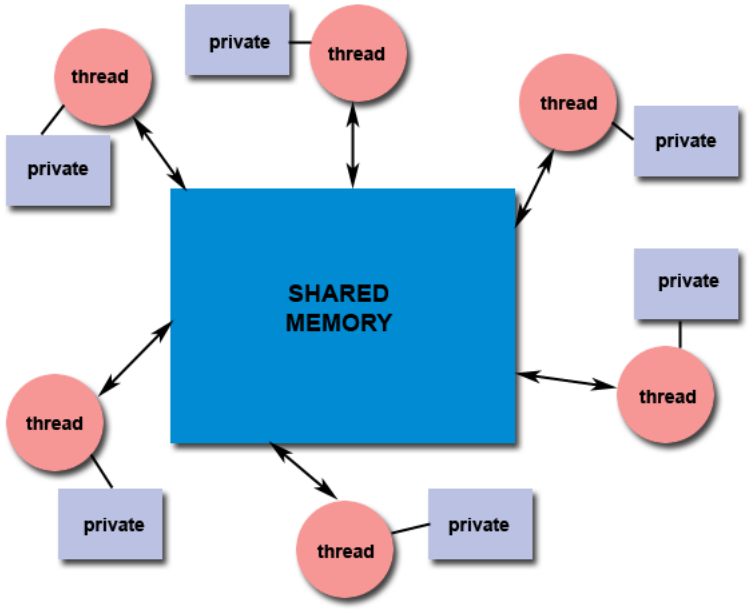

Figure 2: Esquma de una compania
(figura tomada del sitio: http://www.chuidiang.com)

En la figura mostrada, 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 ningun avion 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.

0.1 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[1]

Uno de los mayores inconvenientes al momento de implementacion de los IPCS era la sincronizacion, ya que, por mas que se hallan probado en test antes de ser montados al codigo de la simuacion, se tenia el problema que cuando no se leia en forma ordenada, este decia que no habia mensajes(o bien entregaba mensajes vacios) y cuandos se espereana que los hubiera, lo que llevaba a mucha perdida de tiempo en debugeo de codigo.

[1] Debido a falta de tiempo y experiencia con manejo de memoria compartida en linux, no se logro implementar esta funcionalidad completamente por lo que no esta funcional.

Part III
Consideraciones de tiempo vs memoria

Al principio, cada vez que un avion llegaba a una ciudad, se realizaba un DFS por cada ciudad para encontrar el camino a las ciudad mas corta. 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 innecesariemente 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 ultiliacion de memoria espacial o de tiempo.

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 $o(1)$. Sin embargo, nuvamente, agrega une extra en la cantidad de memoria necesaria disponible.

Part IV
Concusiones

Luego de realizado este trabajo practico, se aprendio sobre la potencia que se puede tener al realizar programas que sean capaces de dividir sus tareas en proecsos concretos y por sobre todo, al mantenerlos sincronizados.