Sistemas Operativos

*Trabajo Práctico 1*

*IPCs y Servidores Concurrentes*

José Ignacio Galindo  
Federico Homovc  
Nicolás Loreti

ITBA 2011

Introducción:

Este trabajo consiste de una aplicación que simula la distribución de medicinas entre ciudades por medio de empresas aerolíneas que utilizan aviones. Cada ciudad posee una necesidad inicial de medicamentos de algún tipo, y cada aerolínea tiene una cierta cantidad de aviones cada uno con uno o más medicamentos iniciales.

        La aplicación está dividida en distintos procesos. El principal es llamado *Map* y es el que inicializa el servidor para poder comunicar todos los procesos. Es el encargado de crear el proceso *IO* (para la salida por pantalla) y un proceso para cada una de las compañías. Cada compañía crea, además, un thread por cada avión. Al iniciarse todos los procesos de las compañías y de IO, éstos se conectan al servidor y mandan un mensaje al mapa para avisar que está listo; una vez que todos están conectados, el mapa inicia la simulación por turnos.

Un turno empieza mostrando el mapa con las ciudades y sus respectivas necesidades. Luego, cada empresa mueve todos sus aviones una unidad de distancia hacia la ciudad que se dirige y, en caso de que arribe, se lo comunica al mapa. El mapa verifica si los aviones de cada compañía pueden descargar todas o alguna de las medicinas y le envía a IO por cada compañía un mensaje con todos los aviones que descargaron. IO muestra dichos aviones en pantalla o, en caso de que ninguno descargue, un mensaje correspondiente. Luego, cada empresa reasigna el destino a sus aviones que hubiesen arribado una nueva ciudad basándose en la carga del avión, las necesidades de la ciudad y la distancia hasta la misma.

Sincronización de procesos:

La sincronización de los distintos procesos se lleva a cabo mediante el pasaje de mensajes , aprovechando la capacidad bloqueante de las llamadas al sistema diseñados para esta tarea en cada IPC. Para lograrlo, todas las lecturas y escrituras de mensajes (mediante las funciones de C *read()* y *write()* ) en el cado de fifo y sockets, msgrcv y msgsnd() en el caso de colas de mensajes, o implementándolo de forma interna en memoria compartida) son bloqueantes, por lo que en más de una ocasión, un proceso debe esperar hasta que otro haya ejecutado cierto fragmento de código y mandado un mensaje de confirmación para poder continuar.

En cuanto a los threads de los procesos de las compañías, se utilizan dos variables de tipo *mutex* y dos *variables de condición*: una de cada tipo para los aviones y una de cada tipo para la compañía. Los aviones inicialmente se crean y se bloquean esperando sobre su variable de condición, luego la compañía hace un *broadcast* sobre esa variable, despertándolos , y luego espera sobre su variable de condición hasta ser despertada una vez por cada avión. De esta forma, se asegura que una vez despierta todos los aviones ya hayan concluido su trabajo. Este mecanismo se repite a través del código de la compañía, que corresponden a controlar las medicinas , descargar las medicinas, y buscar un nuevo destino.

La variable mutex de los aviones sirve para evitar que más de un thread modifique una variable global al mismo tiempo, con lo que se perderían datos sobre qué avión llegó a una ciudad y cuál no. Cada avión debe bloquearla cuando comienza a ejecutar y desbloquearla cuando está esperando. La variable mutex de la compañía sirve para esperar a que un thread se duerma antes de que otro quiera despertarlo, con lo que se producirían *dead locks* y el proceso se bloquearía irremediablemente.

Implementación de IPCs:

Para la implementación de las capa de transporte se decidió utilizar una interfaz unificada para los distintos IPCS. Esta interfaz utiliza el concepto de cliente- servidor , que consideramos adecuada para el modelo de problema que se nos planteó, ya que en nuestro caso se pensó en el mapa como una especie de servidor que no sólo crea a todos los demás procesos, sino que además controla el único recurso compartido entre todos los procesos: el grafo del mapa, procesando las solicitudes de cada uno de los clientes. De esta manera, el mapa es el encargado de inicializar el servidor, que dependiendo de la implementación de IPC, utiliza una implementación del ADT servADT distinta. De esta forma , como map inicializa el server antes de crear a los demás procesos, todos los procesos pueden acceder a la información del server después de las llamadas a fork(). Aquí es conveniente mencionar que en un primer momento se había decido utilizar llamadas a exec() después de hacer el fork() para crear a todas las empresas, pero por la cuestión de compartir la información inicial del servidor, y por considerarlo una solución más simple, se decidió utilizar solamente fork().

Una vez creado un proceso, éste hace una llamada a connectToServer() que recibe justamente como parámetro del ADT del server, y de esa manera obtiene el ADT que identifica a su cliente, con el que le es posible comunicarse. Como se puede pensar, el mapa además de crear el servidor también tiene que conectarse al servidor a través de la función anterior. Una vez conectado, usará el pid de cada uno de sus hijos - que conoce, ya que el mismo los concibió- para así obtener cada uno de los clientes, y así poder comunicarse. Por ende, el mapa envía mensajes a cada uno de sus hijos (  los hijos no se comunican entre si ) y cada uno de los hijos se comunica con él enviándole mensajes a través del cliente del mapa, que puede conseguir, ya que como es su padre, sabe su pid.

A continuación se explica detalladamente como implementó este mecanismo cada IPC, que en algunos casos difiere en gran medida debido a la implementación de threads internos por ejemplo.

**FIFO**  
          
**Sockets**  
          
**Message Queue**

La implementación de colas de mensajes resultó ser la más simple en términos de complejidad y longitud del código. Se decidió implementar el ADT del servidor con un único entero llamado queueID, que como su nombre lo indica, guarda el identificador de la única cola de mensajes creada para que todos los procesos (clientes) puedan comunicarse. En este punto es necesario mencionar que se optó por usar una sola cola de mensajes, sin prioridades (ya que no se consideraron necesarias para el problema planteado) , donde cada “casillero” de la cola de mensajes esté identificado con el pid del proceso comunicado. De esta manera un proceso recibe mensajes desde su cliente ( que justamente está implementado sólo con un entero queueID y su pid ), osea que escucha en su pid, y envía mensajes a otros procesos haciendo un msgsend() con el pid del mensaje destinatario. De este manera, la implementación de colas de mensajes se adaptó de forma idónea al modelo del proyecto, ya que al contar el mapa con los pid de todos sus hijos, simplemente envía mensajes usando msgsend(), utilizando como parámetro el pid de cada hijo; y cada hijo, al conocer el pid del padre, es decir del mapa, envía mensajes al mismo haciendo msgsend() con su pid. Por ende, en cada momento de tiempo en cada “casillero” de la cola de mensajes de cada hijo hay un solo mensaje, pero en el “casillero” del mapa, los mensajes se van encolando, llegando a haber en la cola tantos mensajes como aerolíneas hubiese.

En cuanto a la creación del servidor, representado como se mencionó anteriormente por serverADT, se realiza dentro de la función createServer(), y básicamente se obtiene la identificación de la cola(queueid), usando la función msgget(), utilizando el flag IPC\_CREAT – para que sólo se cree la cola si no existe previamente- y una clave (queueKey) , que consta de un numero elegido arbitrariamente.

Para conectarse al servidor, se utiliza la función connectToServer(), que simplemente, crea el espacio para un nuevo cliente, repensentado por clientADT, copia el valor del queueID y luego inicializa el pid del cliente con su pid.

El funcionamiento de getClient() es similar al de connectToServer() , con la diferencia que en vez de inicializar el pid del ADT con el suyo, lo hace con el que le es pasado como parámetro.

Para el envío y la recepción de mensajes se usa otra estructura denominada msgQueue que, contiene un long mtype , que guarda el pid del mensaje enviado (para indicarle a la cola de mensajes que el mensaje se debe guardar en la “ranura” dado por el valor de pid), y un arreglo de chars llamado mtext que es inicilizado con un tamaño prefijado dado por la constante MSG\_SIZE.

En las funciones disconnectFromServer() y terminateServer(), sólo se destruyen las estructuras del cliente y del servidor, correspondientemente, haciendo un free() de los mismos. También en terminateServer() se destruye la cola de mensajes haciendo una llamada a la función msgctl().

**Shared Memory**

En el caso de memoria compartida, se pensó en un momento en hacer una sóla llamada shmget(), de manera de obtener un solo bloque que memoria, que a su vez iba a ser dividido en varios bloques, uno por cada cliente, partiendo desde una cantidad de clientes máxima prefijada. Cada uno de estos bloques iba a estar compuesto por una lista circular de mensajes, y de esta manera la implementación del ADT del cliente iba a tener un puntero a void para apuntar a la memoria compartida, un entero usado como offset- para indicar el comienzo de su bloque- y dos enteros adicionales para indicar el comienzo y el fin de la cola dentro del bloque. De esta manera, se suponía que un si un cliente se quería comunicar con otro iba a escribir un mensaje en la lista circular dentro del bloque el mensaje destinatario, y para recibir un mensaje un cliente tendría que leer de la lista circular de su propio bloque. Así podrían convivir varios mensajes en un mismo momento para cada cliente, siempre manteniendo un tamaño fijo para cada mensaje.

Finamente se optó por descartar la implementación anterior, encontrando una forma considerablemente más simple de resolver el problema presentado en el diseño del IPC. Se decidió descartar las lista circulares y ,en vez que hacer un sola llamada a shmget(), hacer dos, de manera de obtener dos bloques de memoria compartida diferentes; uno que se usaría para los mensajes en sí, y otro que se usaría para guardar la información de los clientes conectados al servidor. De esta manera, una memoria compartida quedaría integrada por un arreglo de estructuras “shmMessage”, formadas principalmente por arreglos de caracteres de un tamaño predefinido para almacenar los mensajes, y otra que quedaría conformada por un arreglo de estructuras clientCDT y que guardaría la información de cada cliente conectado al servidor.

Para concretar este diseño se decidió implementar el tipo abstracto serverADT de la siguiente manera:

struct serverCDT {

int semid;

int shmidClients;

int shmidMessages;

void \* clients;

void \* memory;

};

Con esta estructura, el server puede guardar toda la información necesaria para lograr de forma efectiva la comunicación entre distintos clientes. El campo semid, se usa para guardar el identificador de un arreglo de 2 semáforos system V (implementados en el archivo semaphore.c), los cuales se usan para garantizar la exclusión mutua entre clientes en las memorias compartidas. De forma análoga, los campos shmidClients y shmidMessages, contienen los dos identificadores de las memorias compartidas utilizadas, de manera que cada cliente (proceso) pueda vincularse la memoria compartida. En cuanto a las punteros a void restantes, se utilizan para que el servidor se vincule a las memorias compartidas, pudiendo así inicializarlas en 0, mediante la función cleanUP(), tarea que se realiza al final de la función createServer().

En lo que se refiere a la implementación del ADT de clientADT, fue realizada de la siguiente manera:

struct clientCDT {

pid\_t id;

int used;

int semid;

int shmidMessages;

int offset;

void \* memory;

};

El campo id guarda el pid del proceso presente en la comunicación y es la forma de asociarlo con el cliente que le corresponde. El campo used, como su nombre lo indica, se utiliza para señalizar si el cliente en cuestión está siendo utilizado por un proceso o no. Este campo no se utiliza para la comunicación en sí, sino que lo utiliza el servidor para saber si una determinada posición dentro del vector de memoria compartida de clientes ya se encuentra tomada por un proceso. El campo semid, así como en el servidor, contiene el identificador del arreglo de dos semáforos que se utilizarán para garantizar la exclusión mutua cuando se trata de escribir un mensaje en la zona de memoria compartida destinada para dicha finalidad. Cabe destacar que cuando se usa este campo, dentro de las funciones sendMessage() y rcvMessage(), se utilizarán la funciones de incremento y de decremento del semáforo, con el argumento SEM\_MEMORY, para indicar que se usa el semáforo destinado para la exclusión mutua correspondiente a la memoria compartida de mensajes( un semáforo del arreglo sirve para cada memoria ). Los campos restantes son usados por las funciones de envío y recepción de mensajes para escribir en el espacio destinado para los mensajes del cliente utilizado los mensajes correspodiente, valiéndose de offset para este objetivo

En forma general, la implementación en cuestión funciona de la siguiente manera: al llamar a la función createServer() se crea un nueva instancia del tipo abstracto del servidor, obteniendo a partir de las claves destinadas este uso los identificadores de tanto los semáforos como las memorias compartidas que se utilizarán para la comunicación. Luego se obtiene los puntros a dichas memorias y se las inicializa todas en cero llamando a la función cleanUp(). Una vez obtenido el serverADT cada uno de los procesos involucrados llamará a la función connectToServer(), para así obtener el clientADT a través del cual podrá comunicarse. Esta función utilizará el semáforo destinado para la memoria compartida correspondiente al arreglo de clientes para asegurarse que no hayan problemas de exclución mutua, buscando dentro del arreglo de clientes en la memoria compartida el primer cliente no inicializado. Una vez ubicado el cliente, y si la cantidad máxima de clientes lo permite, inicializará el cliente en cuestión en la memoria compartida con los datos del proceso y luego , creando un nuevo clientADT para devolverlo como valor de retorno, copiará en el los datos del cliente recién inicializado en la memoria compartida. Al inicializar el cliente, también colocará su campo offset apuntando al espacio de memoria compartida de mensajes que le corresponde.

Una vez que el proceso ya tiene su clientADT, es decir que se encuentra conectado al servidor, deberá hacer una llamada a getClient() , para así poder comunicarse con otro proceso, simplemente indicando su pid. De esta forma, en la función getClient() se buscará en la memoria compartida para los clientes , el cliente cuyo pid coincida por el indicado por el que llamó el cliente, y si lo encuentra, inicializará un nuevo clientADT, copiará los datos y devolverá. Teniendo así el cliente con el cual puede comunicarse con otro proceso, llamará a la función sendMessage() que simplemente, recibiendo la información del cliente destinatario, buscará en el arreglo de memoria compartida para el uso de los mensajes, el lugar donde se encuentre el espacio reservado para los mensajes de dicho cliente ,y así, utilizando memcpy, guardará en mensaje enviado, colocando en el flag isWritten en true. Aquí es importante aclarar que este IPC, a diferencia de los demás , no puede guardar más de un mensaje para un mismo cliente simultáneamente. La manera en que se soluciona este problema es la siguiente: cuando el flag isWritten de la estructura shmMessage correspondiente a un determinado proceso se encuentra activo, el proceso que envía el segundo mensaje, se bloquea en un ciclo hasta que el mensaje en dicha posición sea leído y así el flag sea desactivado. De esta manera se logra implementar este ipc de una manera simple, no impidiendo el correcto funcionamiento del mismo.

Resultados:

Todas las mediciones fueron realizadas dentro de una máquina virtual ( VIrtua Box ) corriendo Ubuntu 11,4 32 bits, utilizando como sistema operativo anfitrión Mac OSX Snow Leopard (10.6.8) y usando como equipo una computadora MacBook Pro , modelo 8,4 , con un procesador Intel Core i7 2Ghz de cuatro núcleos.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Shared Memory | Message Queue | Fifos | Sockets |
| test1 |  |  |  |  |
| test2 |  |  |  |  |
| test3 |  |  |  |  |
| test4 (cities 30, cias 10, planes 5) | 329868,703 | 3896,08 | 6301,9386 | 4477,244667 |
| test5 (cities 40, cias 10, planes 5) | 917,0808 | 194,69633 | 285,813 | 218,2736667 |
| test6 (cities 10, cias 10, planes 10) | 971,8636 | 102,338 | 28,3296 | 13,2854 |
| test7 (cities 20, cias 10, planes 10) | 936,2905 | 106,567 | 40,5378 | 22,6092 |
| test8 (cities 30, cias 10, planes 10) | 10287,817 | 6.336 | 1871,165667 | 1103,511667 |
| test9 (cities 50, cias 10, planes 10) | 31544,552 | 9787,672 | 3214,50633 | 3324,4346 |
| test10 (cities 60, cias 10, planes 10) | 48337,336 | 13402,67 | 4776,688667 | 4475,464333 |