TP II – Unidad V a VII – OBLIGATORIO

Objetivo: Desarrollar una serie de programas que permitan comprobar distintas funcionalidades vinculadas con la comunicación entre procesos utilizando FIFO’s e IPC (Inter Process Communication) provistas por el kernel del SO.

1. Realizar un chat entre procesos independientes usando fifo’s. Cada proceso independiente recibe por línea de comandos 2 parámetros: el nombre del fifo de escritura y el nombre del fifo de lectura. Puede resolver todo el problema programando un único programa (chatfifo.c) y creando dos procesos a partir de un mismo programa, ejecutándolo desde dos terminales distintas con distintos argumentos. Establezca un protocolo para salir del chat, por ejemplo, cuando se tipea la palabra “bye” el proceso deja de enviar y recibir mensajes a través de los fifo’s.

Utilice línea de comandos para crear los fifo’s, ejemplo:

```
$ mkfifo fifo1
$ mkfifo fifo2
```

Abrir dos terminales y ejecutar:

```
$ ./chatfifo fifo1 fifo2
$ ./chatfifo fifo2 fifo1
```

Luego puede destruir los fifos creados:

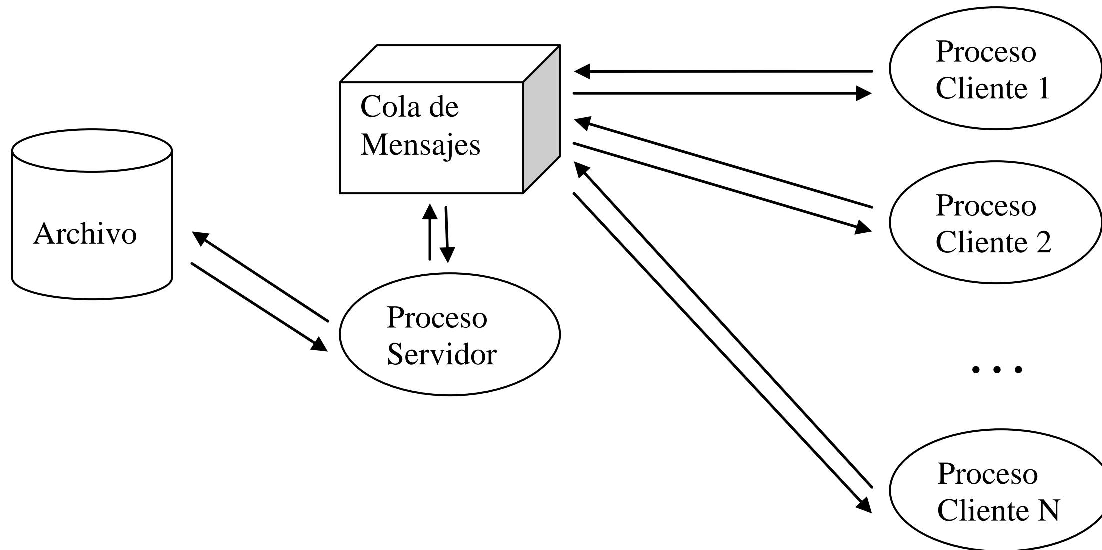
```
$ unlink fifo1
$ unlink fifo2
```

2. Desarrolle un programa de chat usando Cola de Mensajes. Repita el mismo esquema de procesos que en el punto anterior (2 procesos independientes con un hijo cada uno). Puede resolverse con una única cola de mensajes en donde se escriben mensajes de distinto tipo (al menos requerirá mensajes de 2 tipos distintos). Puede resolverlo con un único programa, del cual crea dos procesos independientes y recibe argumentos por línea de comandos.

3. Desarrolle un programa de chat usando Memoria Compartida. Repita el mismo esquema de procesos que en el punto anterior (2 procesos independientes con un hijo cada uno). Puede resolverse con una única área de memoria con espacio para dos mensajes (digamos, de 256 bytes cada una, ésta será la limitación del programa, no acepta mensajes de más de 256 caracteres). Se requerirá de dos semáforos para que, cuando un proceso escriba un mensaje no permita que otro proceso lo lea hasta que la escritura no se haya completado y cuando un proceso lea, impedir que otro proceso escriba el mensaje que se esta leyendo.

4. Desarrolle un programa de chat usando archivos. Elija las estructuras de datos y organización de archivos que mejor se adapte a la resolución de este problema. Realice el diseño de la aplicación y luego su implementación.

5. Se pretende compartir un archivo de acceso directo entre múltiples procesos utilizando una cola de mensajes como medio de comunicación bidireccional entre un proceso server y múltiples procesos clientes. El esquema es el siguiente:



El protocolo entre los procesos clientes (todos iguales) y el proceso servidor es el siguiente:
-Todos los procesos cliente utilizan la misma cola de mensajes (0xA)
-Para simplificar, el formato del registro del archivo de acceso directo es el siguiente:

```
Estado,1 byte (0 es registro libre, 1 registro ocupado, 2 registro borrado)
Descripción,100 bytes, cadena de caracteres q incluye \0 al final de la cadena
```

Y esta creado, inicializado con 1000 registros con estado 0 y descripción con ceros binarios.
-Los procesos clientes son todos iguales e interactivos, con opción de menú para agregar registros, modificar registros, borrar registros, leer o consultar registros y salir.
-Todos los procesos clientes envían mensajes de tipo 1 a la cola 0xA en este formato:

```
<pid>,<número de registro>,<descripción, char[100]>
```

Dónde:
<pid> es un número entero que representa el id de proceso
<número de registro> es el número de registro objetivo dentro del archivo, numero entero entre 0 y N-1 registros tenga el archivo de acceso directo, en caso de ser "-1" significa "agregar al final del archivo"
<descripción> cadena de caracteres de máximo 100 bytes (incluyendo \0) que representa el contenido de campo Descripción del archivo de acceso directo. Cuando Descripción es "borrar" significa que se pretende borrar ese registro y <numero de registro> debe estar entre 0 y N-1. Cuando Descripción es "leer" significa que se pretende leer ese registro y <numero de registro> debe estar entre 0 y N-1.
Ejemplo para proceso cliente con pid=1234:

```
1234,-1,hola
1234,45,q tal
1234,54,borrar
1234,45,leer
```

El último ejemplo significa "leer el registro 45", el anterior "borrar el registro 54".

-El proceso servidor siempre va a responder al proceso cliente utilizando la misma cola de mensajes

(0xA)
-Si el proceso servidor no responde a un proceso cliente, éste quedará bloqueado.
-Si el proceso servidor recibe CTRL+C (señal de interrupción), significa que debe terminar de atender al cliente actual (si es que alguno se está atendiendo en este momento) y finalizar su trabajo.
-El proceso servidor recibe por línea de comandos el nombre del archivo de acceso directo, si el archivo existe, lo abre de R/W, si no existe, lo crea e inicializa y lo deja abierto de R/W listo para trabajar.
-El tipo de mensaje que grabará en la cola el proceso servidor, como respuesta a una petición de proceso cliente será de tipo <pid> (es decir, coincide con el pid del proceso cliente que luego lo leerá)
-El formato de respuesta del proceso servidor en la cola es el mismo (en cuanto a tipos de datos y longitud) que el enviado por el cliente:

```
<retorno>,<numero registro>,<descripción>
```

Dónde:
<retorno> es un número entero, 0 significa error y el campo descripción contiene la descripción del error; 1 significa que no hubo error y el campo descripción contiene el campo descripción del registro.
<numero registro> es un número entero que indica el registro del archivo de acceso directo afectado por la operación que el proceso cliente envió
<descripción> cadena de 100 bytes máximo (incluye \0) que contiene la el campo descripción del registro insertado, borrado, modificado, leído. En caso de que <retorno> sea 0, contiene una descripción de error.
Ejemplo de respuestas de proceso servidor para peticiones de proceso cliente con pid=1234:

```
1,1000,hola
1,45,q tal
0,54,registro 54 esta vacio
1,45,q tal
```

En este ejemplo se asume que el archivo de acceso directo fue creado con 1000 registros vacíos (de 0 a 999), por lo tanto, si se pide agregar un registro, éste será el registro 1000. No se puede borrar un registro que está vacío, como es el caso del registro 54.

Esta versión de este programa no contempla los problemas de acceso concurrente a registros o manejo “transaccional” de registros.

6. Modifique los programas servidor y cliente del punto anterior para trabajar de una forma “transaccional” con los registros. Puede suceder que uno o más procesos clientes podrían ejecutarse desde distintas terminales remotas conectadas a un mismo computador, por lo tanto, podría tratarse de distintos usuarios que están modificando “al mismo tiempo” registros de este archivo de acceso directo. Entonces, para evitar que un usuario interfiera con otro ante la modificación “concurrente” de un registro[1], un usuario podría utilizar un mecanismo de lock/unlock de registro para asegurarse que otro usuario no pueda usar un registro determinado cuando él lo tome como propio. Se supone que el lock dura poco tiempo, un usuario hace lock del registro X, luego puede leerlo, modificarlo, etc y por último tiene que hacer un unlock para permitir que otros usuarios usen dicho registro. Un registro con lock puesto, solo puede ser usado por ese usuario/proceso cliente únicamente. El

[1] El entrecomillado de “al mismo tiempo” y “concurrente” se justifica por el hecho de que el proceso servidor serializa las peticiones de los procesos clientes al utilizar una cola de mensajes.

registro se libera haciendo un unlock. Si se pretende hacer un cambio sobre un registro sin hacer lock previamente, solo se permitirá en caso de que el registro no este previamente con lock.

-Se mantiene el formato de mensajes entre proceso servidor y procesos clientes.
-Ahora un proceso cliente puede enviar mensaje de lock, ejemplo de lock sobre registro 45 enviado por proceso cliente 1234:

```
1234,45,lock
```

Y también puede enviar mensaje de unlock:

```
1234,45,unlock
```

-El proceso servidor responderá –cuando no hay error- ante el mensaje de lock de la siguiente forma (asumiendo que el contenido del registro es “q tal”):

```
1,45,q tal
```

Y ante un mensaje de unlock:

```
1,45,q tal
```

En caso de error, podrá enviar el mensaje correspondiente:

```
0,45,registro ya lockeado por pid 4567
0,45,registro 45 nunca tuvo lock previo
```

De esta forma, el cambio del registro 45 por parte del proceso cliente 1234, sería:

```
1234,45,lock
1234,45,nueva descripción
1234,45,unlock
```

Discuta con su equipo o en clase la forma de implementación de este nuevo protocolo.

7. El desarrollo moderno de aplicaciones de N capas (ver https://www.grch.com.ar/docs/bd/apuntes/ArquitecturasNTier.pdf) por lo general requiere de utilización de lenguajes y frameworks modernos, no obstante, lo subyacente de mucho de esto es la interfaz CGI. En el ejercicio 10 del TP I, hemos propuesto una discusión acerca de la interfaz CGI. Uno de los problemas es el código HTML *hardcoded* con multiples o complejos *printf's* dentro del código C. ¿De qué forma podemos mejorar esto? ¿Cómo podríamos hacer para poner este código HTML de forma externa al programa? ¿Podríamos utilizar archivos en el file system? No, ante miles o millones de peticiones en un sitio web muy concurrido ello tendría un gran impacto en el servidor web ¿entonces, que podemos usar? Memoria Compartida! Evitaríamos el acceso al disco y mejoraríamos mucho el rendimiento y hasta incluso podemos subir info de base de datos a modo de memoria *cache*.
La idea sería poner todo el código HTML en un archivo (para simplificar) y éste archivo subirlo a memoria compartida. Pero.. ¿Qué pasa si múltiples programas CGI requieren distintos códigos HTML? Podríamos poner todo dentro de un mismo archivo de texto y delimitarlo con *tags*, ejemplo de contenido index.html:

```
<cabecera>
<!DOCTYPE html>
<TITLE>Monitoreo Servidor</TITLE>
</cabecera>

<menu>
<h1>Opciones de monitoreo</h1>
```

```
<ul>
  <li><a href="./ps.html">Procesos</a></li>
  <li><a href="./ps2.html">Procesos (version 2)</a></li>
  <li><a href="./disk.html">Disco</a></li>
  <li><a href="./mem.html">Memoria</a></li>
  <li><a href="cgi-bin/abmprod">ABM Productos</a></li>
</ul>
</menu>

<fin>
</BODY>
</HTML>
</fin>
```

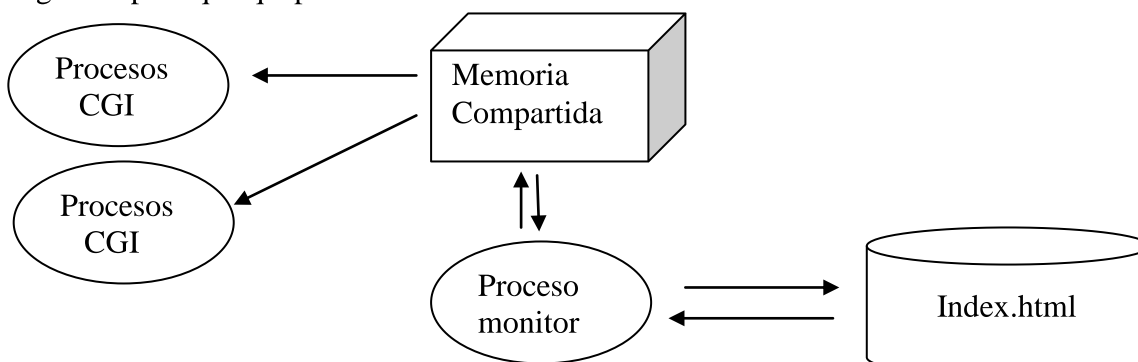
Entonces, de esta forma, dentro de un programa CGI podríamos resolver el problema de esta forma:

```
char *strHtml = obtenerHtml(shmid,”cabecera”);
printf(“%s”,strHtml);
free(strHtml);
```

la función obtenerHtml() devolverá “<!DOCTYPE html>\n<TITLE>Monitoreo Servidor</TITLE>\n”, es decir, lo que delimita el tag <cabecera>..</cabecera>; suponiendo que la función obtenerHtml() utiliza malloc() en su interior. Otro diseño posible (tal vez, más eficiente, sería evitar el malloc() y asumir que el programador tiene memoria estática suficiente para ese código html), el riesgo sería que dicha memoria no sea suficiente (o bien se agregue código html y no se amplíe el espacio de memoria estática), ejemplo:

```
char strHtml[500];
obtenerHtml(shmid,strHtml,500,”cabecera”);
printf(“%s”,strHtml);
```

la función obtenerHtml() no podría cargar más de 499 caracteres más el \0 final para evitar el desborde de la memoria y afectar al programa.
Como Ud sabe, el desarrollo web es muy dinámico y seguramente necesitará modificar el archivo index.html mientras prueba los programas CGI que lo utilizan, la solución a este problema podría ser hacer un programa (monitor.c) que chequee el archivo index.html y cada vez que el mismo se modifique en el disco, vuelva a subir su contenido a memoria compartida, en caso de que el tamaño del archivo supere a al tamaño de la memoria compartida, deberá borrarla y volver a crearla más grande para que quepa el archivo:



Los programas CGI deberán contemplar la posibilidad de que la memoria compartida que pretenden consultar puede no estar disponible si es que la misma se está actualizando o re-creando. Habrá que implementar una forma de reintentos y *delay* ante error en la función shmget() (cuando ésta devuelve -1).
Desarrolle el programa monitor.c el cual recibirá los siguientes datos por línea de comandos:

-archivo a monitorear (ej: index.html)
-cada cuantos segundos verifica el archivo
-tamaño en KB incial de memoria compartida (ej: 1 creará una memoria compartida de 1024 bytes)
-clave hexadecimal de memoria compartida (ej: 0xA)

Sugerencia: utilice la señal de alarma para secuenciar el monitoreo, utilice la función stat() para verificar si el archivo sufrió una modificación desde la carga inicial que hizo el programa en memoria compartida. Detener el programa con CTRL+C. Utilice el comando ipcs para destruir la memoria compartida. También tendrá que diseñar el contenido de la memoria compartida, pues el programa deberá saber el tamaño de la misma y el último tamaño subido del archivo, un formato posible podría ser:
<long tamaño memoria compartida><long tamaño de archivo><contenido del archivo>\0..(hasta el final de la memoria compartida)
Entonces, si el programa monitor se cae, se puede volver a ejecutar y consultar el contenido de la memoria compartida y volver a empezar (e incluso los programas CGI podrían continuar ejecutándose sin problema).
Una vez que la aplicación ha sido desarrollada, este programa ya no sería necesario o en tal caso, habría que hacer un programa que simplemente cargue el contenido del archivo en memoria compartida y finalice, sin hacer ningún monitoreo.
Por último, el desarrollo web puede requerir contenido dinámico, en tal caso, podríamos definir una sintaxis de parámetros o variables dentro del contenido estático html del archivo index.html y estas variables se podrían reemplazar desde el código CGI, ejemplo:

```
<cabecera>
<!DOCTYPE html>
<TITLE>$titulo</TITLE>
</cabecera>

<menu>
<h1>Opciones de monitoreo</h1>
<ul>
  <li><a href="./ps.html">Procesos</a></li>
  <li><a href="./ps2.html">Procesos (version 2)</a></li>
  <li><a href="./disk.html">Disco</a></li>
  <li><a href="./mem.html">Memoria</a></li>
  <li><a href="cgi-bin/abmprod">ABM Productos</a></li>
</ul>
</menu>

<fin>
</BODY>
</HTML>
```

```
</fin>
```

```
char strHtml[500];
obtenerHtml(shmid,strHtml,500,"cabecera","titulo=Monitoreo
Servidor&variable2=contenido2…");
printf("%s",strHtml);
```

la función obtenerHtml() obtiene el código html estático y reemplaza los nombres de variables por el contenido que se indica en la llamada la función.
PARA NO COMPLEJIZAR MAS ESTE TRABAJO PRACTICO NO IMPLEMENTE ESTA FUNCIONALIDAD, se explicó solo a modo de ver las posibilidades que tiene esta tecnología que estamos desarrollando.

8. Desarrolle un programa CGI muy simple que utilice la función obtenerHtml() en combinación con el programa monitor.c y un archivo html subido a memoria compartida, cambie el archivo y observe como monitor sube los cambios y luego el programa CGI muestra el cambio en la página web. Es decir, pruebe todo el desarrollo del punto anterior. Con estas herramientas se puede implementar toda una aplicación web de alto rendimiento, más aún si mantenemos datos en memoria compartida a modo de memoria *cache* de consultas frecuentes que se hacen a la base de datos del sistema, datos de parámetros globales de la aplicación, etc.

Introducción Memoria Compartida y Semáforos

La memoria compartida es una de las técnicas de IPC más utilizadas y eficientes puesto que los datos están disponibles para todos los procesos y no requieren que éstos realicen "copias privadas" de los datos a intercambiar. La implementación de distintos tipos de servidores y soluciones empresariales basadas en memoria compartida plantea el problema de la exclusión mutua, requiriendo de semáforos para el control de concurrencia entre procesos.

En 1965 se publica el tratado de Dijkstra[2] cuyo principio fundamental es: "dos o más procesos pueden cooperar por medio de simples señales, de forma tal, que un proceso pueda ser obligado a parar en un lugar determinado hasta que haya recibido una señal específica. Cualquier requisito complejo de coordinación puede ser satisfecho con la estructura de señales apropiada". Los semáforos son las señales a las cuales se refería Dijstra; enviar una señal al semáforo `s` es `semSignal(s)` y parar hasta recibir la señal del semáforo `s` es `semWait(s)`.

Bibliografía

- Stallings, William, "Sistemas Operativos", 5ta. Edición. Prentice Hall. 2001. Madrid.
- Stevens, Richard, "Advanced Programming in the UNIX Environment", Addison-Wesley Professional Computing Series, 1993, ISBN 0-201-56317-7
- LooseSandra Loosemore, Richard M. Stallman, Roland McGrath, Andrew Oram, Ulrich Drepper, "The GNU C Library Reference Manual", Free Software Foundation, 2007, Boston, USA.

[2] Dijstra, E., "Cooperating Sequential Processes", Technological University, Eindhoven, The Netherlands, 1965.