Sistemas Operativos – Práctica 4

FECHA DE ENTREGA: VIERNES 29 DE ABRIL (HORA LÍMITE DE ENTREGA, 23:00 HORAS).

La cuarta práctica se va a desarrollar en dos fases según el siguiente cronograma:
Semana 1: Colas de Mensajes
Hasta el final: Ejercicio global que incorpora los conocimientos adquiridos a lo largo de las prácticas de Sistemas Operativos.

Los ejercicios correspondientes a esta cuarta práctica se van a clasificar en:

APRENDIZAJE, se refiere a ejercicios altamente recomendable realizar para el correcto seguimiento de los conocimientos teóricos que se presentan en esta unidad didáctica.

ENTREGABLE, estos ejercicios son obligatorios y deben entregarse correctamente implementados y documentados. Para la documentación se recomienda hacer uso de doxygen. Con dicha herramienta podemos generar la documentación de nuestro proyecto en html, pero también es posible obtener dicha documentación en pdf (haciendo make en el directorio latex generado tras ejecutar doxygen) crear páginas man (para lo que hay que igualar GENERATE_MAN a YES en el fichero Doxyfile) de ayuda. En este sentido, resulta de gran ayuda utilizar doxywizard para configurar el fichero Doxyfile de configuración de la herramienta.

SEMANA 1

Colas de Mensajes

Las colas de mensajes, junto con los semáforos y la memoria compartida son los recursos compartidos que pone Unix a disposición de los programas para que puedan intercambiarse información.

En C para Unix es posible hacer que dos procesos sean capaces de enviarse mensajes y de esta forma intercambiar información. El mecanismo para conseguirlo es el de una cola de mensajes. Los procesos introducen mensajes en la cola y se va almacenando en ella. Cuando un proceso extrae un mensaje de la cola, extrae el primer mensaje que se introdujo y dicho mensaje se borra de la cola. Las colas de mensajes son un recurso global que gestiona el sistema operativo.

El sistema de colas de mensajes es análogo a un sistema de correos, y en él se pueden distinguir dos tipos de elementos:

- *Mensajes*: son similares a las cartas que se envían por correo, y por tanto contienen la información que se desea transmitir entre los procesos.
- *Remitente* y *destinatario*: es el proceso que envía o recibe los mensajes, respectivamente. Ambos deberán solicitar al sistema operativo acceso a la cola de mensajes que los comunica antes de poder utilizarla. Desde ese momento, el proceso remitente puede componer un mensaje y enviarlo a la cola de mensajes, y el proceso destinatario puede acudir en cualquier momento a recuperar un mensaje de la cola.

Es posible hacer "tipos" de mensajes distintos, de forma que cada tipo de mensaje contiene una información distinta y va identificado por un entero. Por ejemplo, los mensajes de tipo 1 pueden contener el saldo de una cuenta de banco y el número de dicha cuenta, los de tipo 2 pueden contener el nombre de una sucursal bancaria y su calle, etc. Los procesos luego pueden retirar mensajes de la

cola selectivamente por su tipo. Si un proceso sólo está interesado en saldos de cuentas, extraería únicamente mensajes de tipo 1, etc.

Los procesos acceden secuencialmente a la cola, leyendo los mensajes en orden cronológico (desde el más antiguo al más reciente), pero selectivamente, esto es, considerando sólo los mensajes de un cierto tipo: esta última característica nos da un tipo de control de la prioridad sobre los mensajes que leemos.

Las funciones para trabajar con colas de mensajes en Unix en C están incluidas en las librerías *<sys/types.h>*, *<sys/ipc.h>* y *<sys/msg.h>*. En particular, las funciones que se van a emplear son *msgget()*, *msgsnd()*, *msgrcv()* y *msgctl()*.

La sintaxis de las funciones anteriores es la siguiente:

Creación de colas de mensajes: *int msgget (key_t key, int msgflg);*
Recibe como argumento una clave IPC y los flags similares a los que ya hemos visto con memoria compartida y semáforos, ejemplo: IPC_CREAT | 0660 que crea la cola, si no existe, y da acceso al propietario y grupo de usuarios.
La función devuelve el identificador de la cola. En caso de que la función genere un error, devolverá -1.

Enviar datos a una cola de mensajes: *int msgsnd(int msqid, struct msgbuf *msgp, int msgsz, int msgflg);*

- *msqid* (valor que devuelve la función *msgget()*) es el identificador de la cola,
- *msgp* es un puntero al mensaje que tenemos que enviar. La estructura *msgbuf* tenemos que definirla en ella hay indicar forzosamente el primer campo de la estructura como un long que representará el tipo de mensaje y el resto de los campos de la estructura es el mensaje que enviamos.
  Por defecto, la estructura base del sistema que describe un mensaje se llama *msgbuf* y está declarada en `linux/msg.h`

```
/* message buffer for msgsnd and msgrcv calls */
struct msgbuf {
    long mtype;         /* type of message */
    char mtext[1];      /* message text */
};
```

  El campo *mtype* representa el tipo de mensaje y es un número estrictamente Positivo (>0). El segundo campo representa el contenido del mensaje.

  La estructura msgbuf puede ser redefinida y contener datos complejos; por ejemplo:

```
struct message {
    long mtype;         /* message type */
    long sender;        /* sender id */
    long receiver;      /* receiver id */
    struct info data;   /* message content */
    ...
};
```

- *msgsz* es la dimensión del mensaje, excluyendo la longitud del tipo mtype que tiene la longitud de un long, que es normalmente de 4 bytes. Sobre la estructura anterior *message* la longitud del mensaje sería:

```
length = sizeof(struct message) - sizeof(long);
```

- *msgflg* es un flag relativo a la política de espera referente a cuando la cola está llena. Por defecto, el flag es IPC_WAIT, es decir, en el caso de que la cola esté completa, el proceso se queda bloqueado esperando a que se libere algún hueco. Si *msgflg* es puesta a IPC_NOWAIT y no hubiera espacio disponible, el proceso emisor no esperará y saldrá con el código de error EAGAIN.

Recibir datos de una cola de mensajes: *int msgrcv (int msqid, struct msgbuf *msgp, int msgsz, long mtype, int msgflg);*

La llamada a sistema *msgrcv* lee un mensaje de la cola de mensajes especificada por *msqid* y de tipo *mtype*. El mensaje leído se guarda en *msgp*, eliminándolo de la cola de mensajes. El argumento *msgsz* especifica el tamaño máximo en bytes de la zona de memoria apuntada por *msgp*. En el argumento *msgflg* hay diversas opciones:

- MSG_NOERROR: si el tamaño del mensaje es superior al tamaño especificado en el campo *msgsz*, y si la opción MSG_NOERROR está posicionada, el mensaje se truncará. La parte sobrante se pierde. En el caso en que no esté esta opción, el mensaje no se retira de la cola y la llamada fracasa devolviendo como error E2BIG.
- IPC_NOWAIT: esta opción permite evitar la espera activa. Si la cola no está nunca vacía, se devuelve el error ENOMSG. Si esta opción no está activa, la llamada se suspende hasta que un dato del tipo solicitado entre en la cola de mensajes.

El tipo de mensaje a leer debe especificarse en el campo *mtype*:

- Si *mtype* es igual a 0, se lee el primer mensaje de la cola, es decir, el mensaje más antiguo, sea cual sea su tipo.
- Si *mtype* es negativo, entonces se devuelve el primer mensaje de la cola con el tipo menor, inferior o igual al valor absoluto de *mtype*.
- Si *mtype* es positivo, se devuelve el primer mensaje de la cola con un tipo estrictamente igual a *mtype*. En el caso de que esté presente la opción MSG_EXCPT, se devolverá el primer mensaje con un tipo diferente.

Operaciones de control sobre una cola de mensajes: *int msgctl (int msqid, int cmd, struct msqid_ds *buf);*

Donde *msqid* es el identificador de la cola, *cmd* es la operación que se quiere realizar y *buf* es una estructura donde se guarda la información asociada a la cola en caso de que la operación sea IPC_STAT o IPC_SET. Las operaciones que se pueden realizar son:

IPC_STAT: guarda la información asociada a la cola de mensajes en la estructura apuntada por *buf*. Esta información es por ejemplo el tamaño de la cola, el identificador del proceso que la ha creado, los permisos, etc.

IPC_SET: Establece los permisos de la cola de mensajes a los de la estructura *buf*.

IPC_RMID: Marca la cola para borrado. No se borra hasta que no haya ningún proceso que esté asociada a él.

Devuelve 0 si éxito y –1 en caso de error.

Ejemplo de uso:

Programa 1

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include  <stdio.h>

#define N 33
```

```
typedef struct _Mensaje{
      long id;  /*Campo obligatorio a long que identifica el tipo de mensaje*/

      /*Informacion a transmitir en el mensaje*/

     int valor;
     char aviso[80];
}mensaje;

int main(void){

 key_t clave;
 int msqid;

mensaje msg;

/*
 *  Se obtiene una clave a partir de un fichero existente cualquiera
 *  y de un entero cualquiera. Todos los procesos que quieran compartir la
 * cola de mensaje deben usar el mismo fichero y el mismo entero.
 */

 clave = ftok ("/bin/ls", N);
 if (clave == (key_t) -1)
 {
            perror("Error al obtener clave para cola mensajes\n");
            exit(EXIT_FAILURE);
 }

/*
 * Se crea la cola de mensajes y se obtiene un identificador para ella.
 * El IPC_CREAT indica que cree la cola de mensajes si no lo está.
 * 0600 son permisos de lectura y escritura para el usuario que lance
 * los procesos. Es importante el 0 delante para que se interprete en octal.
 */
 msqid = msgget (clave, 0600 | IPC_CREAT);
 if (msqid == -1)
 {
             perror("Error al obtener identificador para cola mensajes");
            return(0);
 }


 msg.id = 1;  /*Tipo de mensaje*/
 msg.valor= 29;
 strcpy (msg.aviso, "Hola a todos");

/*
 * Se envia el mensaje. Los parámetros son:
 *  Id de la cola de mensajes.
 *  Dirección al mensaje, convirtiéndola en puntero a (struct msgbuf *)
 *  Tamaño total de los campos de datos de nuestro mensaje (parte del envio)
 *  flags. IPC_NOWAIT indica que si el mensaje no se puede enviar
 *(habitualmente porque la cola de mensajes esta llena), que no espere
 * y de un error. Si no se pone este flag, el programa queda bloqueado
 * hasta que se pueda enviar el mensaje.
```

```
 */

 msgsnd (msqid, (struct msgbuf *) &msg,  sizeof(mensaje) - sizeof(long),   IPC_NOWAIT);

/*
 * Se recibe un mensaje del otro proceso. Los parámetros son:
 * Id de la cola de mensajes.
 * Dirección del sitio en el que queremos recibir el mensaje, convirtiendolo en puntero a
(struct msgbuf *).
 * Tamaño máximo de nuestros campos de datos.
 * Identificador del tipo de mensaje que queremos recibir.
 * flags. En este caso se quiere que el programa quede bloqueado hasta
 *  que llegue un mensaje de tipo 2. Si se pone IPC_NOWAIT, se devolvería
 *   un error en caso de que no haya mensaje de tipo 2 y el proceso continuaría ejecutándose.
 */

 msgrcv (msqid, (struct msgbuf *)&msg, sizeof(mensaje) - sizeof(long),  2, 0);

 printf("Recibido mensaje tipo %d \n", msg.id);
 printf("Dato_Numerico = %d \n", msg.valor);
 printf("Mensaje = %s \n", msg.aviso);

/*
 * Se borra y cierra la cola de mensajes.
 * IPC_RMID indica que se quiere borrar. El puntero del final son datos
 * que se quieran pasar para otros comandos. IPC_RMID no necesita datos,
 * así que se pasa un puntero a NULL.
 */

  msgctl (msqid, IPC_RMID, (struct msqid_ds *)NULL);

 exit(EXIT_SUCCESS);

}
```

Programa 2

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>

#define N 33

typedef struct _Mensaje{
 long id; /* Identificador del mensaje*/

/* Informacion que se quiere transmitir*/

 int valor;
 char aviso[80];
}mensaje;
```

```
int main(void)
{

 key_t clave;
 int msqid;

mensaje msg;


 clave = ftok ("/bin/ls", N); /*Misma clave que el proceso cooperante*/
 if (clave == (key_t)-1)
 {
            perror("Error al obtener clave para cola mensajes \n");
           exit(EXIT_FAILURE);
 }

/*
* Se crea la cola de mensajes y se obtiene un identificador para ella.
 * El IPC_CREAT indica que cree la cola de mensajes si no lo está.
 *  0600 son permisos de lectura y escritura para el usuario que lance
 * los procesos. Es importante el 0 delante para que se interprete en octal.
 */
 msqid = msgget (clave, 0600 | IPC_CREAT);
 if (msqid == -1)
 {
           perror ("Error al obtener identificador para cola mensajes \n");
           exit(EXIT_FAILURE);
 }


/*
 * Recepcion de un mensaje
 */

 msgrcv (msqid, (struct msgbuf *) &msg, sizeof(mensaje) - sizeof(long),   1, 0);

 printf("Recibido mensaje tipo %d \n", msg.id);
 printf("Dato_Numerico = %d \n", msg.valor);
 printf("Mensaje = %s \n", msg.aviso);

 /*
  * Envio de un mensaje
   */
 msg.id = 2;
 msg.valor = 13;
 strcpy (msg.aviso, "Adios");

    msgsnd (msqid, (struct msgbuf *) &msg, sizeof(mensaje)-sizeof(long), IPC_NOWAIT);



   msgctl (msqid, IPC_RMID, (struct msqid_ds *)NULL);

   exit(EXIT_SUCCESS);
}
```

Ejercicio 1. **(ENTREGABLE) (3.0 ptos)** Se pretende diseñar e implementar una cadena de montaje usando colas de mensajes de UNIX. La cadena de montaje está compuesta por tres procesos (A, B y C), cada uno especializado en una función. La comunicación entre cada par de procesos (es decir el proceso i y el proceso i+1) se realiza a través de una cola de mensajes de UNIX. En esta cadena de montaje, cada proceso realiza una función bien diferenciada:

- El primer proceso A lee de un fichero f1 y escribe en la primera cola de mensajes trozos del fichero de longitud máxima 4KB.
- El proceso intermedio B lee de la cola de mensajes cada trozo del fichero y realiza una simple función de conversión, consistente en reemplazar las letras minúsculas por letras mayúsculas. Una vez realizada esta transformación, escribe el contenido en la cola de mensajes.
- El último proceso lee de la cola el trozo de memoria y lo vuelca al fichero f2.

El programa principal acepta dos argumentos de entrada, correspondientes al nombre del fichero origen (f1) y destino (f2). Debe ejecutarse de la siguiente manera:

```
$ cadena_montaje <f1> <f2>
```

Se pide:
a)Decisiones de diseño necesarias para la implementación del programa cadena_montaje. Se entrega un archivo cadena_montaje.doc.
b)Código fuente en el lenguaje de programación C del programa cadena_montaje.c

SEMANA 2 y 3

Ejercicio Final

Se propone un ejercicio en el que se van a utilizar la mayor parte de los mecanismos que se han trabajado por separado durante el curso. El ejercicio consiste en la **simulación de la gestión de aulas durante un examen**.

El objetivo será gestionar los espacios de un aula durante la realización de un examen. Para ello los alumnos deberán esperar a la entrada del aula hasta que un profesor le entregue el enunciado del examen y le indique el asiento que ha de ocupar. Existe un total de dos aulas para examen y dos profesores por aula. Los profesores están encargados de colocar a los alumnos y recoger los exámenes una vez haya concluido el tiempo de la prueba de evaluación.

Requisitos.

- El simulador pedirá por pantalla el número de asientos disponibles en cada aula.
- Asimismo preguntará cuántos alumnos entran en la simulación. Para que la simulación sea efectiva el número de alumnos debe ser mayor que el número de plazas disponibles en el aula de mayor tamaño, aunque el número total de alumnos debe ser inferior que el número total de plazas disponibles.
- Al generar nuevos procesos/hilos alumno se asignarán aleatoriamente a un aula, en cuya entrada esperarán a que un profesor les asigne un asiento.
- Cuando el nivel de ocupación de un aula sea del 85%, los alumnos en espera a la entrada intentarán entrar en la otra aula.
- Existirá un proceso gestor de asignación de espacios que será el que controlará al resto de procesos o hilos que se creen.
- Cada alumno será un nuevo proceso hijo o hilo creado en paralelo y dependiente del gestor. El id del alumno es el id del proceso hijo o hilo que se genere.
- Los asientos de las aulas se codificarán mediante memoria compartida debidamente reservada y gestionada

- Existen un control de acceso a la entrada del aula, y un segundo control a la hora de entregar los exámenes. Cuando un alumno entrega un examen deja libre su asiento, que puede ser ocupado por un nuevo alumno (si lo hubiere).
- Los alumnos entrarán al aula con un tiempo aleatorio (menor de 5 segundos).
- Existirán dos procesos "profesor" en cada aula. Estos estarán dormidos hasta que el gestor les mande la necesidad de atender a un alumno que está en la entrada del aula. El gestor asignará una vez a cada profesor el trabajo a realizar informándole de su id de proceso/hilo.
- Cada vez que entre un alumno, se le indicará al profesor el identificador del alumno al que debe asignar un espacio libre. El profesor buscará un asiento libre y, una vez asignado el espacio, registra el identificador del alumno en dicho asiento e informa al gestor de la posición en la que se ha sentado el alumno. Dicha posición queda marcada como ocupada.
- En cada aula existirá otro proceso "profesor_examen" que se encargará de recoger los exámenes. Cada alumno permanecerá realizando el examen un tiempo aleatorio. Tras ese periodo, avisará al proceso profesor_examen despertándole e indicándole su identificador. El profesor_examen recogerá el examen, con lo que el asiento queda libre. Tras ello el profesor_examen informa al gestor del asiento que queda libre.
- Existirá un tiempo máximo de examen de 5 minutos, el cual vendrá codificado mediante una alarma. Pasado ese tiempo el proceso profesor_examen pasa a recoger todos los exámenes.
- El alumno saldrá por la puerta de salida del aula y terminará su proceso/hilo.
- Se pueden bloquear alumnos en la entrada del aula. Puede haber bloqueos también en determinados accesos a memoria pero serán de tiempo muy breve. También en el trabajo de los profesores o profesor_examen puede haber bloqueos que deben mostrarse.
- Para controlar la carga del sistema, se generará una alarma cada 30 segundos en el proceso gestor para que, a través de llamadas a exec de un comando "ps" de sistema se muestren todos los procesos hijos vivos del proceso actual (ayuda "ps –ef | grep ID_PROCESO | grep –v grep"). A través de una llamada a dup2 se generará un fichero llamado SIGHUP_PPID_lista_proc.txt con dicha lista de procesos recuperando la salida que deje el exec
- Es un requisito utilizar todas las herramientas "ipc" trabajadas en las prácticas (memoria compartida, semáforos, señales, colas de mensajes) y se valorará positivamente el uso del mayor número posible de mecanismos estudiados sobre comunicación y control de procesos (exec, dup2, pipe, fork, thread…)
- Cualquier mejora adicional que el estudiante identifique para completar la simulación de los espacios de examen se implementará
- Se debe intentar minimizar el número de recursos utilizados en la implementación final

Se entregará:

- Código de simulación (archivos .c y .h)
- Makefile
- Documentación doxygen generada del código
- Memoria explicativa pdf indicando al menos:
    1. Descripción de la solución.
    2. Listado de archivos, y componentes entregados y sus relaciones.
    3. Trazabilidad de los archivos y componentes con los requisitos solicitados (en qué archivos/funciones se encuentra la solución a cada requisito)

    Pruebas realizadas. Se exige un alto nivel de pruebas en el que se muestre que:
        - Se producen bloqueos en la entrada/salida
        - Otros bloqueos en los procesos gestores (profesor, profesor_examen…)
        - Se invocan todas las señales identificadas
        - Los hilos/procesos realizan su parte del trabajo
        - Se utilizan todas las herramientas ipc
        - Mecanismos de comunicación y gestión de procesos utilizados
        - Hay un control de errores o workaround en caso de fallos