Sistemas Operativos – Práctica 4

FECHA DE ENTREGA: MARTES 9 DE MAYO HASTA 12 DE MAYO (HORA LÍMITE DE ENTREGA,23:00 HORAS).

La cuarta práctica se va a desarrollar en dos fases según el siguiente cronograma:

- Colas de Mensajes
- 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.

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 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 que definir 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:

o 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.

o 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 está 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, convirtiéndolo 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) (2.0 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 el código fuente en el lenguaje de programación C del programa cadena_montaje.c

EJERCICIO FINAL

(8 PUNTOS)

Introducción

En este ejercicio se debe construir un simulador de carreras de caballos que funciona de forma distribuida. El número de caballos participantes es variable, así como la longitud de la carrera.

El proceso inicial (ejecutado por el usuario) es el encargado de arbitrar las carreras, llevando cuenta de la posición de cada uno de los caballos y sincronizando los avances de los mismos. Cada proceso hijo simula uno de los caballos, lanzando uno o más dados para determinar cuánto debe avanzar, y enviando el resultado al proceso principal, que debe sumar dicha cantidad a la

posición del caballo correspondiente.

En las siguientes prácticas se ampliará el funcionamiento de este simulador, por lo que es importante mantener un código limpio y modular.

Estructura

Los caballos se identifican por un número: 1,2,3,4, …

Los apostadores tienen los nombres: “Apostador-1”, “Apostador-2”, …

El esquema de ejecución del simulador será el siguiente:

1- El usuario arranca el proceso principal, introduciendo como parámetros:
   a. el número de participantes (caballos) para la carrera. Máximo 10 caballos.
   b. la longitud de la carrera (el número que, una vez alcanzado por un caballo, le da la victoria)
   c. Número de apostadores. Máximo 10 apostadores.
   d. Número de ventanillas para gestionar las apuestas
2- El proceso principal crea:
   a. Proceso monitor
   b. Proceso gestor de apuestas
   c. Proceso apostador
   d. Tantos procesos hijo como caballos haya y establece los recursos necesarios de comunicación a los caballos (tuberías)
3- El proceso principal esperará un tiempo entre 15 segundos para comenzar la carrera. Entonces marcará la carrera como comenzada.

Simulación de la carrera

1- El proceso principal envía la información acerca de la posición de cada caballo (para determinar el tipo de tirada) a los procesos caballo hijo, y les notifica mediante una señal que queda a la espera de su simulación. Tras esto, espera el resultado de las tiradas de los caballos.
2- Cada proceso hijo lee la información acerca de su posición, determina el tipo de tirada y genera el número correspondiente. Lo envía a través de un mensaje al proceso principal y queda en una espera no ocupada.
3- El proceso principal recoge los mensajes de las tiradas, actualiza las posiciones de cada caballo y determina cómo se harán las próximas tiradas.

4- La carrera terminará cuando el usuario mande una señal de interrupción con Ctrl+C o cuando alguno de los participantes llegue a la “meta”.
5- Se debe esperar 15 segundos para mostrar los resultados de la carrera y los resultados de las apuestas. Finalmente se deben liberar todos los recursos,

prestando especial atención a evitar procesos huérfanos. Notificar la orden de finalización mediante señales.

Proceso monitor

El proceso monitor es el encargado de presentar por pantalla la información relevante. Tiene dos pantallas:

- Hasta que finalice la carrera. Muestra:
    - Estado de la carrera:
        - segundos que faltan para que comience
        - comenzada
    - Posición de los caballos
    - Estado de las apuestas
        - Cotización de cada caballo
- Finalizada la carrera:
    - "Carrera finalizada"
    - Resultados de los tres primeros puestos (cuando la carrera finaliza)
    - Resultados de las apuestas: apostadores ganadores y beneficios

Este proceso comprobará cada segundo la información y actualizará la pantalla.

Proceso gestor de apuestas

1- Inicializa las apuestas:
    a. Total dinero apostado a cada caballo = 1.0
    b. Cotización de cada caballo = <total dinero apostado a todos los caballos> / <dinero apostado al caballo>
    c. Dinero a pagar a cada apostador para cada caballo = 0
2- Inicializa tantos threads como ventanillas de gestión de apuestas
3- Recibe mensajes de apuestas en una cola
4- Los mensajes de apuestas que se van recibiendo son procesados por los threads "ventanilla"
5- Sólo se procesan apuestas hasta el comienzo de la carrera. Está prohibido procesar ninguna apuesta una vez comenzada la carrera.
6- Cada ventanilla:
    a. Asume uno de los mensajes de apuesta
    b. Comprueba el caballo de la apuesta
    c. Se le asigna al apostador la cantidad que se le pagara en caso de que el caballo gane = <dinero apostado> * <cotización del caballo>
    d. Se actualiza la cotización de los caballos:
        i. <Cotización de un caballo> = <total dinero apostado a todos los caballos> / <total dinero apostado al caballo>

Proceso apostador

Generará de forma aleatoria apuestas.

Cada 0,1 segundos enviará un mensaje de apuesta.

Estructura de información en el mensaje:

- Nombre del apostador: char[20]
- Número de caballo: int
- Cuantía de la apuesta: double

Reglas

A continuación se detallan las reglas de tiradas del simulador de carreras:

1- Una tirada normal supone lanzar un dado estándar, es decir generar un número entero aleatorio del 1 al 6.
2- Cuando un caballo va en cabeza, realiza una tirada ganadora lanzando un dado del 1 al 7.
3- Cuando un caballo va el último, realiza una tirada remontadora lanzando dos dados normales.

Ejemplo En la siguiente tabla se ve un ejemplo de la posición y el valor de las tiradas en una simulación cuya meta está en 40:

	Caballo1	**Caballo2**	**Caballo 3**	**Caballo 4**	**Caballo 5**
Posición	0	0	0	0	0
Tirada	4	5	4	6	6
Posición	4	5	4	6	6
Tirada	3	3	7	4	2
Posición	7	8	11	10	8
Tirada	9	1	7	6	5
Posición	16	9	18	16	13
Tirada	4	3	5	6	3
Posición	20	12	23	22	16
Tirada	3	8	5	6	5
Posición	23	20	28	28	21
Tirada	3	10	5	5	5

Posición	26	30	33	33	26
Tirada	6	2	3	7	6
Resultado	32	32	36	40	32

En rojo aparecen marcadas las tiradas *ganadoras*, en azul las tiradas *remontadoras* y en gris las tiradas *normales*. En este ejemplo, gana el caballo 4 en 7 tiradas.