Comunicación entre procesos - Informática II

Clase 1 – Primera parte

¡Hola! ¿Cómo están? Continuamos bajo la modalidad en línea de Informática II y esta semana comenzaremos a trabajar sobre nuestra primera unidad donde profundizaremos los conocimientos sobre comunicación entre procesos.

Importante

Si todavía no escuchaste los podcast de repaso ni leíste del apunte de IPC los temas de Informática I, te recomendamos que comiences por ahí para no arrastrar errores de concepto en los temas que abordaremos a continuación.

Este documento está pensado para que puedas leerlo en tu tiempo disponible pero contemplando la carga horaria que tenemos en la materia. Les proponemos que lean este material y puedan probar los programas que están subidos en el aula, luego podrán realizar consultas sobre el foro de consultas que se encuentran en el bloque de **comunicación**. Como material de apoyo también están disponibles las presentaciones de clase sobre estos temas.

En el encuentro de hoy nos proponemos abordar los siguientes temas: cola de mensajes, memoria compartida y semáforos.

Tal como hemos visto a lo largo de Informática I y en el repaso realizado este año, podemos comunicar dos procesos mediante señales y pipes, pero en particular necesitamos conocer su identificación (*pid*) o tener una copia de la misma en los casos de proceso padre-hijo. Si queremos comunicar dos procesos independientes, es decir de diferentes padres podemos hacer uso de un *named pipe* o *FIFO* y solo necesitamos conocer el nombre del recurso o archivo específico.

Pero que sucede cuando queremos que más de 2 procesos se comuniquen entre sí, en general, se produce lo que llamamos una colisión, es decir una carrera entre el recurso en cuestión y los procesos que quieren acceder al mismo. ¿Por qué se produce esto? Porque dos procesos no pueden acceder como escritura al mismo recurso, piensen en lo que sucedía en Informática I si queremos abrir como escritura el mismo archivo desde 2 programas. Para solucionar este problema debemos **arbitrar** entre el recurso y los procesos que quieren escribir de forma que lo haga cada uno en un determinado momento y **no colisionen.**

Cola de mensajes

Una cola de mensajes es una estrategia de comunicación entre procesos basada en la lógica **productor – consumidor.** ¿Qué significa esto? Aunque parezca obvio, tiene que ver con los roles que pueden tener los procesos al momento de producir o procesar/distribuir información. Es decir, hay procesos que recibirán datos los procesarán y los enviarán a un recurso compartido y otros procesos que consultarán de una forma específica dicha información y la guardarán en un archivo, la mostrarán por pantalla o se la enviarán a otro proceso (figura 1).

Figura 1: productores (senders) - consumidores (receivers)

En Informática I trabajamos con el concepto de **pila** y **cola**, trabajando con memoria dinámica, ampliando o reduciendo nuestro array de datos. Nosotros éramos los *responsables de pedir o liberar memoria* como así también realizar las funciones pertinentes para *la gestión de la información*.

En el caso de las ***colas de mensajes***, contaremos con un grupo de funciones que nos permitirán **crear** la cola, **agregar** datos, **retirar** el primer dato que entro como alguno especifico y **destruir** el recurso creado. Entonces, ¿cuál es la ventaja frente a lo que hacíamos el año pasado?

En primer lugar, la gestión de memoria que hacíamos era dentro de un contexto, es decir, dentro del contexto del proceso en ejecución. Nuestro contexto era independiente de otro proceso, el sistema operativo se ocupaba que un proceso no interfiera en el espacio de otro. En este caso, Linux nos proveerá de un conjunto de funciones que nos permitirá evitar la gestión de memoria, compartiendo un espacio en común donde dos o más procesos pueden acceder.

Primero lo primero

Avancemos entonces y comencemos por lo primero, es decir, que es lo que debería hacer un proceso para **crear** o **utilizar** una ***cola de mensajes***.

Un proceso puede **crear** una nueva ***cola de mensajes***, o se puede **conectar** a una cola ya existente. De esta forma, dos procesos pueden compartir información. Una vez que se crea una cola de mensajes, esta no desaparece hasta que se destruya.

¿Qué función utilizaremos?

Para crear o conectarnos a una cola utilizaremos la función **msgget**, dicha función posee el siguiente prototipo:

int **msgget** (key_t **key**, int msgflg);

El primer argumento, **key** es del tipo key_t, el cual es un identificador único que describe la cola a la cual uno se quiere conectar (o crear). Cualquier otro proceso que quiera conectarse a esta cola deberá usar el mismo **key** o clave.

El segundo argumento msgflg es un valor entero, que le indicará a la function **msgget** si queremos **crear** la cola o simplemente **conectarnos**.

¿Cómo se crea el identificador key?

key_t es un typedef de un dato tipo **long**, por ello es posible utilizar cualquier número.

Pero:

a. ¿con qué criterio elegimos ese número?
b. ¿qué pasaría si otro programa utiliza el mismo número pero quiere usar otra cola?

La solución que provee Linux es utilizar la función **ftok**, la cual posee el siguiente prototipo:

key_t **ftok** (const char *path, int id);

path debe ser un archivo al cual los procesos que participan deban poder leer (**OJO** permisos).

id es normalmente un char escogido arbitrariamente, como por ejemplo 'A'.

El valor devuelto por la función es la clave que necesitamos para crear o conectarnos a la cola de mensajes.

Veamos un poco como llevamos estas primeras acciones a código:

```
//Genero una clave (es un typedef)
key_t Clave1;

//Adquiero la clave que voy a utilizar
Clave1 = ftok ("/bin/miArchivo", 'A');
```

Importante

Comenzaremos con un ejemplo creciente que creará la clave y luego la cola, controlando errores en cada caso. En los ejemplos siguientes tomaremos de base cada uno de los pasos que vayamos haciendo. Al finalizar la lectura encontrarán este ejemplo desarrollado en el aula.

```
…
if ((key = ftok(nombreArchivo, 'A')) == -1)
{
       perror("ftok");
       exit(1);
}

/* Creo una cola utilizando la constante IPC_CREAT y los permisos
   asociados */

if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1)
{
       perror("msgget");
       exit(1);
}

…
```

Como vemos la función **ftok** hace uso de dos argumentos: una variable que contiene el nombre del archivo elegido y un valor entero que permitió generar nuestra clave. A continuación se utiliza la función **msget** para obtener el identificador de nuestra cola a partir de la clave y el valor entero que nos permite crear la cola.

Ya contamos con nuestra cola de mensajes, nuestro objetivo ahora es poder enviar o retirar mensajes de la misma.

¿Qué funciones utilizaremos? msgsnd y msgrcv , veamos sus prototipos:

int **msgsnd** (int msqid, struct msgbuf *msgp, int msgsz, int msgflg);

int **msgrcv** (int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg);

¿Cómo enviamos datos a la cola?

Invocaremos a la función **msgsnd**, la cual posee 3 argumentos. Lo más lógico es que uno de ellos sea el **identificador** que generamos previamente. En nuestro caso es el primer argumento de la función.

El prototipo de la función es:
int **msgsnd** (int msqid, struct msgbuf *msgp, int msgsz, int msgflg);

- El segundo parámetro es el mensaje en sí. Las colas de mensajes se caracterizan por utilizar un identificador o tipo de mensajes, los mensajes que enviemos pueden para todos igual o para un destinatario en particular.

- Lo más importante es que el mensaje debe ser una estructura cuyo primer campo sea un long. En dicho long se almacena el tipo de mensaje. Al pasar el mensaje como parámetro, se pasa un puntero al mensaje y es necesario realizar un “casteo” a struct msgbuf *.

- El tercer parámetro es el tamaño en bytes del mensaje exceptuando el long, es decir, el tamaño en bytes de los campos con la información.

- El cuarto parámetro son flags: aunque hay varias opciones, la más habitual es poner un 0 o bien IPC_NOWAIT:

    - En el primer caso la llamada a la función queda bloqueada hasta que se pueda enviar el mensaje.

    - En el segundo caso, si el mensaje no se puede enviar, se vuelve inmediatamente con un error. El motivo habitual para que el mensaje no se pueda enviar es que la cola de mensajes esté llena.

Veamos como llevamos estas acciones a código:

```
struct my_msgbuf
{
       long mtype;
       char mtext[200];
};

...
int len;
struct my_msgbuf buf;
...

buf.mtype = 1;
strcpy(buf.mtext, "hola");
len = strlen(buf.mtext);
...
if (msgsnd(msqid, &buf, len+1, 0) == -1)
{
       perror("msgsnd");
}
...
```

¿Cómo retiramos datos de la cola?

Invocaremos a la función **msgrcv**, la cual posee 5 argumentos. Lo más lógico es que uno de ellos sea el **identificador** que generamos previamente. En nuestro caso es el primer argumento de la función.

El prototipo de la función es:
int **msgrcv** (int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg);

- El primer argumento es el identificador de la cola obtenido con **msgget**.

- El segundo argumento msgp, es un puntero a la estructura donde se desea recoger el mensaje. Puede ser, como en la función anterior, cualquier estructura cuyo primer campo sea un **long** para el tipo de mensaje.

- El tercer argumento msgsz, es el tamaño de la estructura exceptuando el **long**.

- El cuarto argumento msgtyp es el tipo de mensaje que se quiere retirar. Se puede indicar un entero positivo para un tipo concreto o un 0 para cualquier tipo de mensaje.

Veamos algunos detalles sobre este parámetro:

- Si el valor de msgtyp es 0, se leerá el mensaje que se encuentra al principio de la cola.
- Si el valor de msgtyp es mayor que 0, se leerá el primer mensaje disponible en la cola cuyo **mtype** coincida con msgtyp.
- Si el valor de msgtyp es menor que 0, se leerá el primer mensaje disponible en la cola cuyo **mtype** sea menor o igual al valor absoluto del msgtyp.

- El quinto parámetro son flags, que habitualmente puede ser 0 o bien IPC_NOWAIT.
  - En el primer caso, la llamada a la función se queda bloqueada hasta que haya un mensaje del tipo indicado.
  - En el segundo caso, se vuelve inmediatamente con un error si no hay mensaje de dicho tipo en la cola.

Veamos como llevamos estas acciones a código:

```
if (msgrcv(msqid, &buf, sizeof buf.mtext, 0, 0) == -1)
{
        perror("msgrcv");
        exit(1);
}
```

Nota: es importante pensar en este momento que esta llamada a función se dará seguramente dentro de un ciclo de control o loop de forma de ir retirando mensajes de la cola, hasta que esté vacía o se cumpla una condición específica.

En función de lo planteado, veamos algunos casos:

En la cola se insertaron seis (6) mensajes distintos cuyos **mtype** son 5, 3, 2, 6, 5 y 1 agregados en ese orden.

- Si invocamos a **msgrcv** con msgtyp **=** 0 sacaría al primer mensaje (mtype=5)
- Si invocamos a **msgrcv** con msgtyp **=** 2 sacaría al tercer mensaje (mtype=2)
- Si invocamos a **msgrcv** con msgtyp **=** -4 sacaría al segundo mensaje (mtype=3)

¿Cómo eliminamos la cola?

Una vez terminada de usar la cola, se debe invocar a la función **msgctl** para liberar los recursos utilizados.

El prototipo de la función es el siguiente:
int **msgctl** (int, int, struct msqid_ds *) ;

La función posee 3 argumentos:

- El primer argumento es el identificador de la cola de mensajes, obtenido con **msgget**.
- El segundo argumento es el comando que se desea ejecutar sobre la cola, en este caso *IPC_RMID.*
- El tercer argumento son datos necesarios para el comando que se quiera ejecutar. En este caso no se necesitan datos y se pasará un NULL.

Veamos como llevamos estas acciones a código:

```
if (msgctl(msqid, IPC_RMID, NULL) == -1)
{
        perror("msgctl");
        exit(1);
}
```

Final de esta clase

Bueno, este es el final de la primera parte de nuestra clase sobre **cola de mensajes**, los y las invitamos a buscar en la sección de **IPC** algunos ejemplos en el siguiente link.

Si surgen dudas o consultas, pueden escribirnos en el foro correspondiente.

En la próxima parte de la clase trabajaremos sobre *memoria compartida y semáforos.*

A continuación, encontrarán un resumen de las funciones que trabajamos en esta parte de la clase.

Nos leemos!

Resumen de funciones

Función	*¿Qué hace?*
int **msgget** (key_t **key**, int msgflg);	Nos conecta a una cola o la crea si no existe y devuelve el identificador.
key_t **ftok** (const char *path, int id);	Obtiene la clave para poder crear la cola.
int **msgsnd** (int msqid, struct msgbuf *msgp, int msgsz, int msgflg) ;	Envía mensaje a la cola
int **msgrcv** (int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg);	Retira mensaje de la cola
int **msgctl** (int msqid, int cmd, struct msqid_ds *buf);	Elimina la cola