Comunicación entre procesos - Informática II

Clase 1 – Segunda parte

¡Hola! ¿Cómo están? En el documento anterior comenzamos a hablar de *comunicación entre procesos* bajo una lógica de **productor – consumidor.** Es decir, procesos que generan información y procesos que consumen dicha información para hacer otra tarea.

Como en toda **transacción** lo importante es **ponernos de acuerdo**, para lo cual esta clase sumaremos un concepto que trabajaron previamente en forma implícita y hoy lo pondremos de manifiesto, los **mecanismos de sincronización**.

Para no perder de vista

Es importante destacar que estas estrategias que estamos viendo deben ser utilizadas en un contexto, es por ello que de poco sirve saber pipes, señales, named pipes o **cola de mensajes** sino que tengo que pensar en *cómo utilizar los recursos en un contexto específico y ver cuál es la mejor estrategia que puedo utilizar para resolver un problema concreto*.

No olvidemos que también podemos utilizar sockets para un tipo de aplicación con una arquitectura **cliente-servidor**. Es decir, que seguimos sumando recursos a nuestra *"caja de herramientas"* y dependerá de nuestra experiencia distinguir cuando es mejor utilizar uno que otro mecanismo. Es por ello, que no alcanzará con leer las clases sino explorar los ejemplos, modificarlos y aprender de ellos, identificar cuando funcionan y cuando no funcionan.

Continuamos entonces esta segunda parte de la clase bajo la modalidad en línea de Informática II y comenzaremos a trabajar sobre dos mecanismos, uno de comunicación y otro de control o sincronización.

¿Qué veremos?

- ***Memoria compartida*:** es un área de memoria accesible por dos o más procesos (para nosotros funcionará como un buffer). Es un mecanismo de *comunicación*.

- ***Semáforos*:** sirven controlar el acceso a información compartida entre dos o más procesos. Es un mecanismo de control o *sincronización*.

¿Qué tienen en común todas estas estrategias?

Todos estos IPC funcionan de una manera similar:

- En primer lugar, se debe crear una llave (key) compartida, para identificar unívocamente el IPC en cuestión. *Recordemos lo que vimos en la cola de mensajes!*
- Dos o más procesos deben conectarse al IPC, utilizando la llave generada.
- Una vez conectados al IPC, los procesos intervinientes pueden utilizar los IPC o configurarlos.

Memoria compartida (shared memory)

Tal como vimos en la primera parte de esta clase, necesitaremos **arbitrar** el acceso a un recurso compartido, en este caso, un segmento de memoria. Las acciones que podemos hacer en principio sobre un recurso es leerlo o escribirlo, para lo cual deberemos ver "quien" o en todo caso, qué proceso realiza *cada acción en cada momento*.

Figura: lógica productor – consumidor sobre un recurso específico

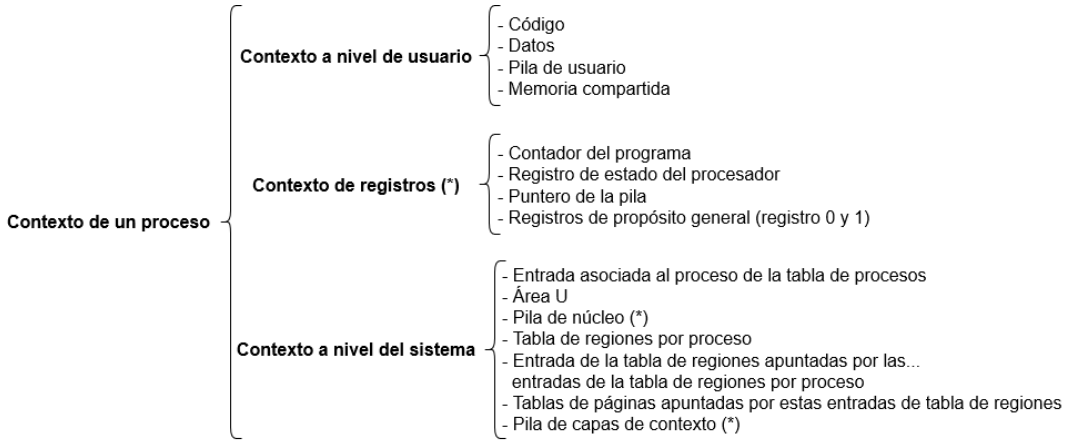
La **memoria compartida**, junto con los **semáforos** y las **colas de mensajes**, son los recursos que disponemos para que dos o más procesos puedan *intercambiar información evitando colisiones*. Esta estrategia plantea que dos o más procesos puedan compartir una zona de memoria en común y así intercambiar datos.

Para repasar y tener en cuenta:

"El contexto de un proceso en un cierto instante de tiempo está formado por su espacio de direcciones virtuales, los contenidos de los registros hardware de la máquina y las estructuras de datos del núcleo asociadas a dicho proceso.

Formalmente, el contexto de un proceso se puede considerar como la unión del contexto a nivel de usuario, contexto de registros y contexto a nivel del sistema.

El contexto a nivel de usuario de un proceso está formado por su código, datos, pila de usuario y memoria compartida que ocupan el espacio de direcciones virtuales del proceso".



Nota: Los elementos marcados con (*) constituyen la parte dinámica del contexto

Es decir que cada proceso tiene su propio contexto y el contenido de nuestros punteros tendrá lógico dentro de este contexto. Debemos entender que necesitamos de *nuevas herramientas* para compartir un segmento de memoria o buffer común. Estas herramientas nos las proveerá el sistema operativo.

Realizadas las aclaraciones, retomemos el tema en cuestión.

La **memoria compartida** es la forma más rápida de comunicación entre procesos ya que una vez que dispongamos de acceso a la memoria creada, el **kernel** no se involucrará en el intercambio de información como limitante deberemos disponer de un *mecanismo de sincronización*, en la clase de hoy, serán los **semáforos**.

Veamos el proceso de intercambio de información, suponiendo que contamos con una aplicación **cliente-servidor** (ver esquema) en la cual:

- El **servidor** genera el acceso a la **memoria compartida**
- Luego, el **servidor** lee el *archivo de entrada* y escribe su contenido en la **memoria compartida.**
- Cuando se finaliza el proceso de lectura, el **servidor** le da aviso al **cliente** mediante un mecanismo de sincronización.
- Al recibir el aviso, el **cliente** comienza a escribir el *archivo de salida* utilizando como origen de la información a la **memoria compartida**.

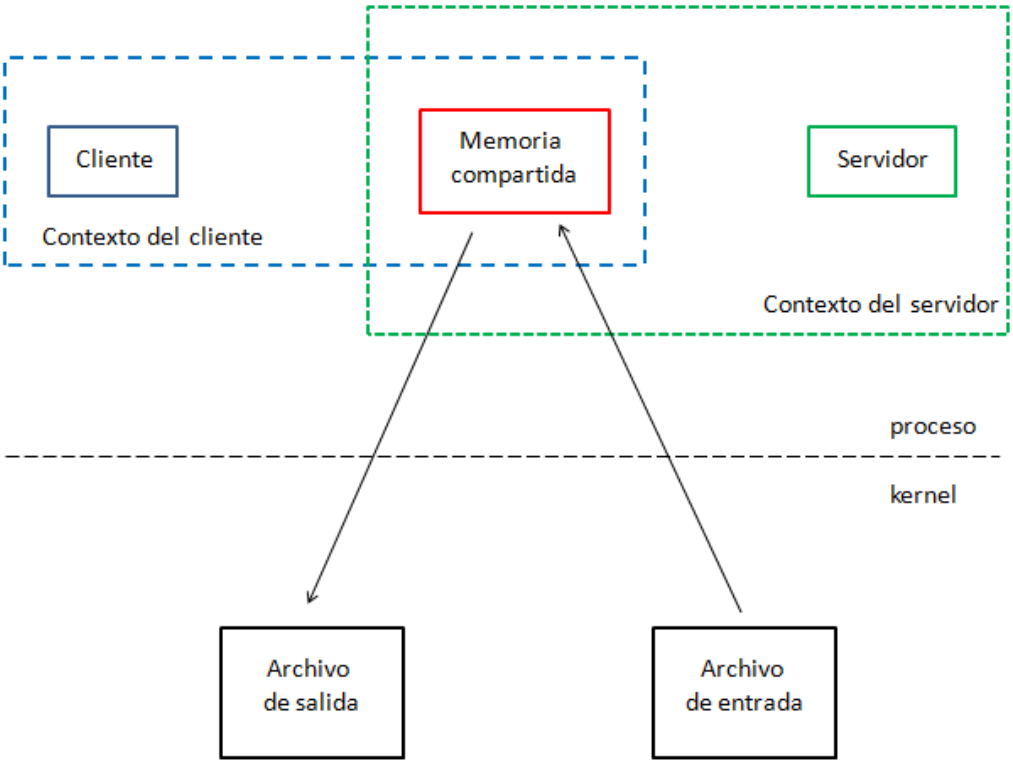


Figura: esquema cliente-servidor utilizando memoria compartida

Dentro de las líneas punteadas tanto del lado del **cliente** como del **servidor**, vemos que *cada proceso tiene su propio contexto* (recuerden lo que vimos como repaso en este mismo documento). Es decir, que la **memoria compartida** aparece en ambos contextos.

Sobre la memoria compartida podremos realizar 4 acciones: crearla o abrirla, acoplarnos, desacoplarnos y liberarla.

¿Cómo empezamos?

Al igual que en **cola de mensajes**, necesitamos conseguir una clave, de tipo **key_t**, que sea **común** para todos los programas que quieran compartir la memoria. En principio, repasando la primera parte de esta clase, ya sabemos cómo hacerlo a través de la invocación de la función **ftok** y sus parámetros.

Creación de la memoria compartida

Para crear la memoria compartida, haremos uso de la función:

int **shmget** (key_t key, int size, int shmflg);

Con la invocación de dicha función creamos el segmento de memoria y nos devolverá un identificador para dicha zona.

- El primer parámetro key, es la clave obtenida anteriormente a través de **ftok** y que debería ser la misma para todos los programas.

- El segundo parámetro size , es el tamaño en bytes que deseamos para la memoria.

- El tercer parámetro shmflg, son flags vinculados a permisos al igual que en el caso de cola de mensajes.

Veamos como lo llevamos a código, tal como hicimos anteriormente trabajaremos sobre un ejemplo creciente:

```
key_t Clave;
int Id_Memoria;
int *Memoria = NULL;
int i,j;

/* Conseguimos una clave para la memoria compartida.
Todos los procesos que quieran compartir la memoria, deben obtener la misma
clave. Esta se puede conseguir por medio de la función ftok.
*/

Clave = ftok ("/bin/ls", 'A');
if (Clave == -1)
{
      printf("No consigo clave para memoria compartida");
      exit(0);
}


/* Creamos la memoria con la clave recién conseguida. Para ello llamamos a la
función shmget pasándole la clave el tamaño de memoria que queremos reservar
(100 enteros en nuestro caso).

Los flags son  los permisos de lectura/escritura/ejecucion para propietario,
grupo y otros (es el 777 en octal) y el flag IPC_CREAT para indicar que cree
la memoria.

La función nos devuelve un identificador para la memoria recién creada.
*/

Id_Memoria = shmget (Clave, sizeof(int)*100, 0777 | IPC_CREAT);

if (Id_Memoria == -1)
{
      printf("No consigo Id para memoria compartida");
      exit (0);
}
```

Acoplarnos con la memoria compartida creada

El último paso antes de poder usar la memoria consiste en obtener un puntero que apunte la zona de memoria, para poder así escribir o leer sobre la misma. Para ello, declaramos en nuestro código un puntero al tipo de dato que sepamos que va a haber en la zona de memoria (una estructura, un array, tipos de datos simples) y utilizaremos la función **shmat** cuyo prototipo es el siguiente:

void* **shmat** (int shmid, const void *shmaddr, int shmflg) ;

Con la invocación de dicha función nos acoplaremos al segmento de memoria compartida creado:

- El primer parámetro shmid, es el identificador de la memoria obtenido en el paso anterior.
- Al momento, los otros dos parámetros bastará rellenarlos con ceros.
- El puntero devuelto es de tipo void *. Debemos hacerle un "casteo" al tipo que necesitemos o queramos utilizar.

Veamos como lo llevamos a código:

```
/* Una vez creada la memoria, hacemos que uno de nuestros punteros apunte a la
zona de memoria recién creada. Para ello llamamos a shmat, pasándole el
identificador obtenido anteriormente. */


Memoria = (int *)shmat(Id_Memoria, (char *)0, 0);

if (Memoria == NULL)
{
       printf("No consigo memoria compartida");
       exit (0);
}

/* Ya podemos utilizar la memoria. Escribimos algunos datos en la memoria. */

for (j=0; j<100; j++)
{
       Memoria[j] = i;
}
printf("Se escribio en la memoria");
```

Liberar la memoria compartida

Una vez terminada de usar la memoria, debemos liberarla. Para ello utilizamos las funciones:

int **shmdt** (const void *shmaddr) ;

int **shmctl** (int shmid, int cmd, struct shmid_ds *buf) ;

La primera función desacopla la **memoria compartida** de la zona de datos de nuestro programa. Alcanza con *pasarle el puntero* shmaddr *que tenemos* a la zona de memoria compartida y llamarla una vez por proceso.

La segunda función destruye realmente la zona de **memoria compartida**. Hay que pasarle el identificador de memoria shmid obtenido con **shmget**(), un flag cmd que indique que queremos destruirla IPC_RMID, y un tercer parámetro al que bastará con pasarle un NULL.

Para poder trabajar con **memoria compartida** *en forma ordenada y sin problemas de acceso*, vamos a necesitar utilizar un mecanismo de sincronización o control, en nuestro caso, utilizaremos **semáforos**. Es decir, podemos establecer un semáforo para acceder a la memoria, de forma que cuando un proceso lo está haciendo, el semáforo se pone "rojo" y ningún otro proceso puede acceder a ella. Cuando el proceso termina, el semáforo se pone "verde" y ya podría acceder otro proceso.

Veamos como lo llevamos a código:

```
/* Terminada de usar la memoria compartida, la liberamos. */

shmdt ((char *)Memoria);

shmctl(Id_Memoria, IPC_RMID, (struct shmid_ds *)NULL);
```

Resumen de funciones

Memoria compartida

Función	¿Qué hace?
int **shmget** (key_t key, int size, int shmflg);	Asigna un segmento de memoria compartida (crea o comprueba su existencia)
key_t **ftok** (const char *path, int id);	Obtiene la clave para poder crear la memoria compartida.
void* **shmat** (int shmid, const void *shmaddr, int shmflg) ;	Se acopla a un segmento de memoria compartida.
int **shmdt** (const void *shmaddr) ;	Se desacopla de un segmento de memoria compartida.
int **shmctl** (int shmid, int cmd, struct shmid_ds *buf) ; Valores posibles de cmd: IPC_SET (modificar) , IPC_STAT (consultar) , IPC_RMID (borrar) .	Operaciones de control sobre un segmento de memoria (borrar, consultar o modificar las propiedades).

Por último los invitamos a explorar los ejemplos que subimos en el aula virtual sobre **memoria compartida** a través del siguiente link.

¿Cómo venimos hasta ahora? Los y las invitamos a ir realizando consultas en el aula virtual de forma de ir resolviendo las dudas que vayan apareciendo.

Continuamos trabajando con estrategias para la comunicación y el intercambio de información entre procesos. El problema a resolver es como sincronizamos el trabajo que realiza cada proceso sobre un mismo recurso evitando las colisiones.

Veamos un ejemplo sencillo, si ejecutamos en la consola: **ps –ef** veremos en la consola la salida de los procesos que se están ejecutando en nuestra computadora. La salida de nuestro proceso es la pantalla (stdout).
Ahora si ejecutamos **ps –ef | grep hola** , el resultado del primer proceso se une mediante un pipe con la entrada del segundo, mostrándonos todas las apariciones de **"hola"** por la pantalla (stdout). En nuestro caso el comando **ps** es el **productor** y **grep** es el **consumidor.**
La sincronización entre productor y consumidor en manejada por el kernel. Este tipo de sincronización se la denomina *implícita.*

Muchas veces es necesario que dos o más procesos accedan a un recurso común, por ejemplo, leer una zona de memoria, mostrar información en pantalla, escribir un archivo, etc. Para ello debemos utilizar herramientas de sincronización que permitan que los procesos trabajen en forma ordenada, comúnmente a este problema lo denominamos como la relación entre productores y consumidores.

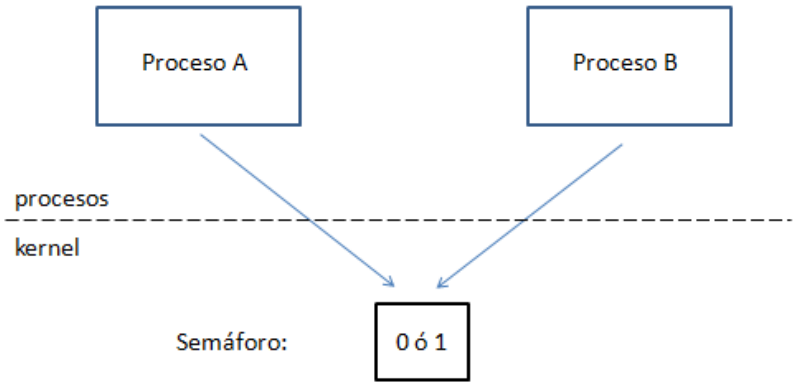
Para ello, tal como lo veníamos adelantando, introduciremos los *semáforos*:

Semáforos

Un semáforo da acceso al recurso a uno de los procesos y se lo niega a los demás mientras el primero no termine. Los semáforos, junto con la memoria compartida y las colas de mensajes, son los recursos compartidos que suministra Linux para comunicación entre procesos.

El funcionamiento del semáforo podemos asociarlo a un "flag" o "bandera", es decir, recursos que hemos utilizado previamente en otros programas de la carrera.

Para comenzar, consideraremos primero un **semáforo binario**, es decir, un semáforo que solo puede valer 0 ó 1.



Sobre un semáforo podemos realizar 3 operaciones:

- Crearlo y darle su valor inicial (create)
- Esperar al semáforo (wait)
- Controlar un semáforo (post)

Otra alternativa es utilizar un semáforo en modo **contador**. Es decir que su valor variará entre 0 y un valor máximo.

Utilizaremos un semáforo para controlar el acceso a un archivo por parte de dos procesos. Imaginemos un semáforo que inicialmente tiene el valor 1 (está en "verde"). Cuando un proceso quiere acceder al archivo, primero debe decrementar el semáforo. El contador queda en valor 0 y como no es negativo, deja que el proceso siga su ejecución y, por tanto, acceda al recurso, en este caso, el archivo.

Ahora un segundo proceso lo intenta acceder, y para ello también decrementa el contador. Ahora el contador toma el valor (-1) y como es negativo, el semáforo se encarga de que el proceso quede "bloqueado" en una cola de espera. Este segundo proceso no continuará por tanto su ejecución y no accederá al recurso.

Supongamos ahora que el primer proceso termina de escribir el archivo. Al finalizar debe incrementar el contador del semáforo. Al hacerlo, este contador se pone a 0. Como no es negativo, el semáforo se encarga de mirar en la cola de procesos pendientes y "desbloquear" al primer proceso de dicha cola. Con ello, el segundo proceso que quería acceder al archivo continua su ejecución y accede al recurso.

Cuando este proceso también termine con el archivo, incrementa el contador y el semáforo vuelve a ponerse a 1, es decir, a estar en "verde".

Es posible hacer que el valor inicial del semáforo sea, por ejemplo, 3, con lo que pasarán los tres primeros procesos que lo intenten. Pueden a su vez quedar muchos procesos encolados simultáneamente, con lo que el contador quedará con un valor negativo grande. Cada vez que un proceso incremente el contador (libere el recurso común), el primer proceso encolado despertará. Los demás seguirán dormidos.

¿Cómo comenzamos?

Para crear o acceder a un semáforo o conjunto de semáforos (*set*) utilizaremos la función **semget** cuyo prototipo es el siguiente:

int **semget** (key_t key, int nsems, int oflag);

- El parámetro key es el identificador que nos permitirá tener una misma clave desde varios procesos.
- El segundo argumento nsems nos permitirá fijar la cantidad de semáforos en el caso que lo estemos creando o si nos estamos sumando a uno creado, lo podemos colocar en 0.
- El tercer argumento oflag nos permitirá fijar los permisos y el modo , es decir si es creación o no.
- El valor devuelto por la función es el identificador del semáforo y será utilizado por las funciones **semop** y **semctl**. En caso de error, devuelve el valor (-1).

Ya tengo un semáforo, ¿y ahora qué hago?

Si ya tenemos un semáforo o conjunto de semáforos abiertos o creados con **semget**, las acciones que podemos realizar, las haremos a través de la invocación de la función **semop** cuyo prototipo es el siguiente:

int **semop** (int semid, struct sembuf *opsptr, size_t nops) ;

- El primer argumento de la función semid es el identificador del semáforo que obtuvimos del paso anterior.
- El segundo argumento opsptr apunta a un array del tipo *struct sembuf.* La información que contiene cada posición del array corresponde a sobre qué semáforo queremos trabajar y que acción llevaremos adelante.
- El tercer argumento nops es la cantidad de elementos del arryay *struct sembuf*.
- En caso de error la función devuelve (-1).

También disponemos de la función **semctl**, la cual tras su invocación, nos permite trabajar sobre un grupo o conjunto de semáforos para consultas o acciones:

int **semctl** (int semid, int semnum, int cmd , …);

- El primer argumento semid es la identificación del conjunto de semáforos
- Su segundo argumento semnum nos permite elegir uno semáforo en particular dentro del conjunto
- El tercer argumento cmd nos permite: obtener el valor del semáforo, establecer un nuevo valor, eliminar el semáforo, entre otras acciones

Final de esta clase

Bueno, este es el final de la segunda parte de la clase donde trabajamos con **memoria compartida** y **semáforos**, 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.

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

Nos leemos!

Resumen de funciones

Memoria compartida

Función	¿Qué hace?
int **shmget** (key_t key, int size, int shmflg);	Asigna un segmento de memoria compartida (crea o comprueba su existencia)
key_t **ftok** (const char *path, int id);	Obtiene la clave para poder crear la memoria compartida.
void* **shmat** (int shmid, const void *shmaddr, int shmflg) ;	Se acopla a un segmento de memoria compartida.
int **shmdt** (const void *shmaddr) ;	Se desacopla de un segmento de memoria compartida.
int **shmctl** (int shmid, int cmd, struct shmid_ds *buf) ; Valores posibles de cmd: IPC_SET (modificar) , IPC_STAT (consultar) , IPC_RMID (borrar) .	Operaciones de control sobre un segmento de memoria (borrar, consultar o modificar las propiedades).

Semáforos

Función	¿Qué hace?
int **semget** (key_t key, int nsems, int oflag);	Crea o accede a un conjunto de semáforos
key_t **ftok** (const char *path, int id);	Obtiene la clave para poder crear la memoria compartida.
int **semop** (int semid, struct sembuf *opsptr, size_t nops) ;	Operaciones de asignación a un conjunto de semáforos.
int **semctl** (int semid, int semnum, int cmd , …);	Operaciones de control o consulta sobre un conjunto de semáforos.