Teoría 5 - Comunicación y Sincronización

Comunicación

Los procesos que se ejecutan concurrentemente de forma **independiente** no afectan y no son afectados por otros del mismo sistema, como tampoco comparte datos. Al contrario, los procesos **cooperativos** pueden afectar como verse afectados por otros procesos, además de compartir datos.

Los procesos muchas veces necesitan cooperar entre sí para:

- Compartir información: brindar un entorno que permita acceder de forma concurrente a cierta información.
- Acelerar cálculos: dividr subtareas para ejecutarlas de forma paralela.
- Modularidad: dividir funciones.
- Conveniencia: trabajar en muchas tareas al mismo tiempo (editar, imprimir, compilar, etc.).

Memoria Compartida

Este es uno de los mecanismo de comunicación entre procesos. Permite establecer una zona común de memoria entre varias aplicaciones. A su vez, una herramienta para controlar el acceso de varios procesos a recursos comunes son los **semáforos**, para controlar que no se acceda con una operación de escritura al mismo lugar de memoria.

System Calls para memoria compartida

`int shmget(key_t key, size_t size, int shmflg)`: permite acceder a una zona de memoria compartida y, opcionalmente, crearla en caso de no existir. Es importante notar que solo reserva el espacio para alojar el segmento, pero no se crea hasta que algún proceso se asigne a él.

- `key`: clave.
- `size_t`: tamaño del segmento a crear.
- `shmflg` (share memory flag): bandera inicial de operación
  - `IPC_CREAT`: Para crear un nuevo segmento.
  - `IPC_EXCL`: se utiliza junto a `IPC_CREAT` para asegurar el fallo si el segmento existe
  - `mode_flags`: los 9 bits menos significativos del número están reservador para establecer permisos de acceso.
- `return`: devuelve un entero denominado `shmid` que se utiliza para hacer referencia al segmento.

`int shmctl(int shmid, int cmd, struct shmid_ds* buf)`: proporciona operaciones para el control de la memoria compartida a través de cmd.

`void* shmat(int shmid, const void* shmaddr, int option)` (Shared Memory Attach): se invoca para adjuntar una zona de memoria compartida dentro de un espacio de direcciones

- `shmid`: el identificador de segmento.
- `shmaddr`: dirección de memoria.
- `option`: banderas.
  - `SHM_RND`: el sistema intentará vincular la zona de memoria a una dirección múltiplo de SHMLBA (Shared Memory Low Boundary Address: constante que representa la unidad mínima de alineación para mapear segmentos de memoria compartida de forma virtual).
  - `SHM_RDONLY`: el segmento solo puede acceder al segmento de memoria en lectura.
- `return`: devuelve la dirección de comienzo del segmento asignado.

`int shmdt(void* schmaddr)` (Shared Memory Detach): esla opreción que realizará el proceso anterior para desvincularse de la memoria compartida. Recibe el puntero devuelto por `shmat`.

Mensajes

Los mensajes son otro mecanismo de comunicación entre procesos. En estos, la comunicación y sincronización es responsabilidad del SO, que proporciona un enlace lógico entre procesos. Los procesos solo deben invocar dos llamadas al sistema: enviar mensaje y recibir mensaje. Es importante notar que los procesos pueden estar tanto en la misma máquina como en máquinas distintas.

Para que los procesos se puedan comunicar, es necesario establecer un **enlace de comunicaciones** entre ellos, que puede implementarse de varias formas:

- Comunicación directa o indirecta.
- Comunicación síncrona o asíncrona.
- Almacenamiento en buffer explícito o automático.

Comunicación directa

En esta, cada proceso debe indicar explícitamente el nombre del receptor o emisor. En este método, los enlaces

- se establecen de forma automática y los procesos solamente deben conocer la identidad del otro,
- solo asocian dos procesos, y
- son únicos entre cada par de procesos.

En principio, el direccionamiento es **simétrico**, en el sentido de que tanto el transmisor debe referenciar al receptor, como el receptor debe referenciar al transmisor. Pero existe una variante **asimétrica** en la que el transmisor debe referenciar al receptor pero el receptor puede recibir mensajes de cualquier proceso, con una variable que captura el identificador del proceso desde el que se direcciona.

Comunicación indirecta

Los mensajes se reciben en **buzones** o puertos, que pueden ser propiedad del proceso que los crea (solo pueden ser usados por él) o del SO (pueden ser usados por varios procesos). En este último caso, el SO debe proporcionar mecanismos para crear y borrar buzones y enviar y recibr mensajes a través de un buzón.

En este tipo de comunicación, se establece un enlace entre procesos solo si comparten un buzón, es decir que puede asociarse un enlace a más de dos procesos.

Comunicación Síncrona

En este, el proceos que envía se bloquea hasta que el mensaje sea recibido (por el receptor o buzón). A su vez, el receptor se bloquea hasta que haya un mensaje disponible.

Existe una variante llamada **rendezvous** en donde el emisor queda bloqueado hasta que el receptor *esté listo* para recibir el mensaje. El receptor sigue estando bloqueado hasta haber un mensaje disponible.

Comunicación Asíncrona

En este caso, tanto el transmisor como el receptor siguen operando sin importar sin importar el "estado" del mensaje. En el caso del receptor, el mensaje es válido o nulo (no hay mensaje disponible).

Almacenamiento en Buffer

Tanto en comunicación directa como indirecta, los mensaje pueden almacenarse en una cola temporal. Esta cola puede implementarse de tres maneras:

- Capacidad limitada (n mensajes): si hay espacio, el emisor continúa la ejecución luego del envío, en caso contrario queda bloqueado hasta que se libere espacio para dejar un mensaje.
- Capacidad ilimitada: el emisor nunca se bloquea en el envío de mensajes
- Capacidad nula (rendezvous): la cola tiene capacidad 0, por lo que no almacena mensajes, por lo que el emisor debe bloquearse hasta que el receptor reciba el mensaje.

AVERIGUAR EN ALGUNO DE LOS LIBROS LAS VENTAJAS Y DESVENTAJAS DE CADA IMPLEMENTACIÓN

Cola de Mensajes

Son un mecanismo de comunicación entre procesos similar a las tuberías pero más vestatil.

Estas colas son gestionadas por el SO y pueden ser escritas y leídas por varios procesos bajo una política FIFO, aunque puede flexibilizarse agregando un "tipo" al mensaje. Para operar con colas, tenemos las llamadas al sistema de las librearias `sys/types.h`, `sys/ipc`, `sys/msg.h`:

- `msgget`: crea una cola o accede a una existente.

- `msgctl`: accede y modifica la información de control asociada a una cola.
- `msgsnd`: escribir/enviar un mensaje en la cola.
- `msgrcv`: leer/recibir un mensaje de la cola.

Además, en consola podemos acceder a las colas con los comandos:

- `ipcs`: muestra las colas con su key, id, bytes, N° de mensajes, etc.
- `ipcrm -q $msqid`: permite eliminar una cola por su id
- `ipcrm -Q $key`: permite eliminar una cola por su key

Pipes

Las tuberías (pipes) son canales de comunicación que conectan procesos. Un canal tiene un extremo de escritura y otro de lectura que funcionan bajo una política FIFO, es decir que lo primero en escribirse es lo primero en leerse. Generalmente un proceso escribe en el canal, mientras que otro proceso lee desde el mismo.

- Tuberías sin nombre: son unidireccionales y se utilizan con la instrucción POSIX **pipe**. Una vez creada, el sistema mantiene un búfer en memoria para transferir bytes del escritor al lector y, una vez que terminan, es recuperado y la canalización desaparace. En todos los sistemas modernos, la barrar vertical '|' representa una tubería sin nombre en linea de comandos.
- Tuberías con nombre: también llamadas FIFO, tienen un archivo de respaldo. Un proceso puede escribir en la tubería mientras que otro lee de la misma, eliminando el dato de la tubería en el proceso. Ambos programas deben ejecutarse en el mismo directorio de trabajo.

Concurrencia y Sincronización

La concurrencia de procesos se refiere a las situaciones en las que dos o más proceso *pueden* coincidir en el acceso a un recurso compartido. Si bien las rutinas funcionan por separado, pueden no funcionar correctamente al ejecutarse de forma concurrente.
Ejemplo: tenemos un proceso `productor` que ingresa productos en un contador (`cont++`) y otro procesos `consumidor` que retira productos del mismo (`cont--`). Las rutinas `cont++` y `cont--` pueden definirse en ensamblador como una seria de instrucciones

```
# cont++
reg1 = cont
reg1 = reg1 + 1
cont = reg1
return

# cont--
reg2 = cont
reg2 = reg2 - 1
```

```
cont = reg2
return
```

Al ejecutarse de forma concurrente `cont++` y `cont--`, el valor puede no ser el esperado. Si comenzamos con `cont = 5` podemos tener como resultado 4, 5 o 6.

Vemos que, al acceder dos procesos/hilos al mismo recurso compartido, se genera una **condición de carrera**, situación en la que el resultado depende del orden de ejecución de los procesos. Para **prevenir** las condiciones de carrera, los procesos concurrentes *cooperativos* deben ser **sincronizados**, y así lograr determinismo.

En esta sección nos vamos a encontrar con los siguientes términos:

- **Sección crítica** (critical section): sección dentro de un proceso que requiere acceso a recursos compartidos, por lo que no debe ser ejecutada mientras otro proceso esté accediendo al mismo recurso.
- **Interbloqueo** (deadlock): situación en la que dos o más procesos son incapaces de actuar porque cada uno está esperando que algunos de los otros haga algo.
- **Círculo vicioso** (livelock): situación en la que dos o más procesos cambian continuamente de estado en respuesta a cambios en los otros procesos, sin realizar ningún trabajo útil.
- **Condición de carrera** (race condition): situación en la qu dos o más procesos leen y escriben un dato compartido y el resultado final depende de la coordinación relativa de sus ejecuciones.
- **Exclusión mutua** (mutual exclusion): requisito para que, cuando un proceso esté en una sección crítica que accede a recursos compartidos, ningún otro proceso pueda estar en una sección crítica donde se acceda al mismo recurso.

Sincronizar

Para generar programas correctos en necesario controlar la cooperación entre threads. Los procesos multithreading deben producir los mismos resultados más allá del orden y forma de ejecución de los threads.

Denominamos **secciones críticas** a los segmentos de código en donde un proceso accede a recursos compartidos. El problema a resolver es el de asegurar que, cuando un proceso está ejecutando una sección crítica, otros procesos no ejecuten su respectiva sección crítica. Hay que tener en cuenta que este segmento debe ser lo más corto posible.
Para solucionar este problema se deben satisfacer las siguientes condiciones:

- Exclusión mutua: si el proceso P_j está ejecutando su sección crítica, ningún otro proceso puede ejecutar la sección crítica.
- Progreso o evolución: ningún proceso que *no* se encuentre en la sección crítica puede impedir que otro ejecute la misma.
- Espera limitada: ningún proceso puede esperar indefinidamente para ejecutar su sección crítica. Se debe limitar el número de veces que los procesos pueden ejecutar la sección crítica después que

otro proceso ha solicitado ejecutar la misma (y antes que la solicitud sea completada).

Soluciones por software

La estructura general de un proceso P_j es:

- Sección de entrada
- Sección crítica
- Sección de salida
- Sección restante (y otra vez a sección de entrada).

Turno compartido

Utiliza una variable turno para lograr la sincronización, donde un proceso solo puede ejecutar su sección crítica solamente cuando es su turno.

```
void Proceso_i(void* x) {
    while(true) {
        while(turno != i){};    // espera su turno
        seccionCritica();               // ejecuta la seccion critica
        turno = j;                              // cede su turno al proximo proceso
        //seccion restante
    }
}
```

Podemos que, aunque **satisface la exclusión mutua**, **no cumple con la condición de progreso**. Por ejemplo, si un proceso tiene una seccion restante muy extensa, el resto de procesos debe esperar que esta termine para que pase su turno y lo pase al siguiente proceso.

Operación atómica

Es una secuencia de operaciones que se ejecuta sin interrupciones, es decir que no puede ser detenida en un estado intermedio ni un estado intermedio puede ser modificado por otra operación.