Concurrencia de Procesos: Exclusión Mutua y Sincronización

En la concurrencia existen **términos clave**:

- Sincronización, los procesos coordinan sus actividades.
- Sección crítica, región de código que solo puede ser accedida por un proceso simultáneamente (variables compartidas).
- Exclusión mutua, solo un proceso puede estar en la sección crítica accediendo a recursos compartidos.
- Interbloqueo, varios procesos, todos tienen algo que otros esperan, y a su vez esperan algo de otros.
- Circulo vicioso, los procesos cambian continuamente de estado como respuesta a cambios en otros procesos sin que sea útil.
- Condición de carrera, varios hilos/procesos leen y escriben dato compartido.
- Inanición, proceso que está listo y se le deniega siempre el acceso a un recurso compartido.

En la concurrencia hay que tener en cuenta que la velocidad relativa de los procesos no puede predecirse.

Para la concurrencia de procesos se puede usar la **memoria compartida**.

- La función **`shmget(key, longitud, shmflag)`** crea un segmento de memoria compartida o solicita accesos a un segmento de memoria existente. La key identifica el segmento, la longitud determina el tamaño de la región compartida y la shmflag es un código que determina si se puede crear un segmento si no existe (IPC_CREAT), o que devuelva error si ya existe el segmento (IPC_EXCL | IPC_CREAT).
- La función **`shmat(shmid, shmaddr, shmflag)`** añade el segmento de memoria compartida a la memoria del proceso. shmaddr si es 0, el SO trata de encontrar una zona donde "mapear" el segmento compartido.

En la **sincronización**, el sistema operativo se encarga de:

1. Seguir la pista de los procesos activos.
2. Asignar y retirar recursos.
3. Proteger los datos y los recursos físicos.
4. Los resultados de un proceso deben ser independientes de la velocidad relativa a la que se realiza la ejecución de otros procesos concurrentes.

Los procesos **interaccionan** de tres formas:

1. Los procesos no tienen conocimiento de los demás. **Competencia**.
   Cuando varios procesos entran en competencia se pueden producir las siguientes situaciones:
   - Exclusión mutua. Solo un programa puede acceder a su sección crítica en un momento dado.
   - Interbloqueo.
   - Inanición.
2. Los procesos tienen un conocimiento indirecto de los otros. **Cooperación por comportamiento**.
   Para que los recursos puedan compartir recursos adecuadamente las operaciones de escritura deben ser mutuamente excluyentes.
3. Los procesos tienen un conocimiento directo de los otros (conocen el PID). **Cooperación por comunicación**.
   La cooperación puede ser por paso de mensajes. En esta situación no es necesario el control de la exclusión mutua. Puede producirse un interbloqueo, cada proceso puede estar esperando una comunicación del otro. Puede producirse inanición.

Los **requisitos para la exclusión mutua** son:

1. Solo un proceso debe tener permiso para entrar en la sección crítica por un recurso en un instante.
2. No debe permitirse el interbloque o la inanición.
3. Cuando ningún proceso está en su sección crítica, cualquier proceso que solicite entrar en la suya debe poder hacerlo sin dilación.
4. No se deben hacer suposiciones sobre la velocidad relativa de los procesos o el número de procesadores.
5. Un proceso permanece en su sección crítica solo un tiempo finito.

Para garantizar la exclusión mutua se puede usar la **espera activa**, **hardware**, **SO** o un lenguaje de programación (semáforos, monitores).

La espera activa no resulta eficaz con exclusión mediante el uso de turnos (variable compartida), aunque hay soluciones como el **algoritmo de Dekker**, en este se impone un orden de actividad de los procesos, si un proceso desea entrar en la sección crítica, debe activar su señal y puede que tenga que esperar a que llegue su turno.

Compartido

```
1  #define FALSE 0
2  #define TRUE 1
3  #define N 2 // Numero de procesos
4  int turno=1;  // con valores de 0 o 1
5  int interesado[N]; // inicializado a 0 para todos los elementos del array
```

Proceso i

```
7  while (1) {
8      interesado[i] =TRUE;
9      while (interesado [j] ==TRUE)
10         if (turno == j) {
11             interesado[i] =FALSE;
12             while (turno == j);
13             interesado[i] =TRUE;
14         }
15     ---SECCION CRITICA ---
16     turno = j; // cambia turno al otro proceso
17     interesado [i] =FALSE;
18     ---RESTO DEL PROCESO ---
19 }
```

Como se puede ver el algoritmo es valido para solo 2 procesos.

Existe otra solución que es la **solución de Peterson**.

Compartido

```
1  #define FALSE 0
2  #define TRUE 1
3  #define N 2 // Número de procesos
4  int turno; // con valores de 0 o 1
5  int interesado[N]; // inicializado a 0 para todos los elementos del array
```

Proceso i

```
7  while (1) {
8      interesado[i] =TRUE;
9      turno = j; // cambia turno al otro proceso
10     while ((turno==j) && (interesado [j] ==TRUE));
11     ---SECCION CRITICA ---
12     interesado[i] =FALSE;
13     ---RESTO DEL PROCESO ---
14 }
```

La última solución (y la más efectiva) es la del **algoritmo de la panadería de Lamport**. Es te algoritmo es válido para n procesos.

Compartido

```
1  #define FALSE 0
2  #define TRUE 1
3  #define N n // Número de procesos concurrentes
4  int eligiendo [N]; // con valores de 0 a n-1
5  int numero[N]={0}; // inicializado a 0 para todos los elementos del array
```

Proceso i

```
7  while (1) {
8      eligiendo[i] =TRUE; // Calcula el número de turno
9      numero[i]=max(numero[0],..., numero[n-1]) + 1;
10     eligiendo[i]=FALSE;
11     for(j=0;j<n;++j) { // Compara con todos los procesos
12         while (eligiendo[j] ==TRUE); // Si el proceso j está eligiendo ==> E. Activa
13         while ( (numero[j] !=0) && ((numero[j] < numero[i])
14                 || ((numero[j]== numero[i]) && (j<i)) ) );
15     }
16     ---SECCION CRITICA ---
17     numero[i] =0; //Libera la sección crítica
18     ---RESTO DEL PROCESO ---
19 }
```

Las soluciones hardware incluyen la **inhabilitación** de interrupciones. Para garantizar la exclusión mutua es suficiente con impedir que un proceso sea interrumpido. Un proceso continuará ejecutándose hasta que solicite un servicio del sistema operativo o hasta que sea interrumpido. Se limita la capacidad del procesador para intercalar programas. En un sistema multiprocesador inhabilitar las interrupciones de un procesador no garantiza la exclusión mutua.

Existen instrucciones especiales de máquina se realizan en un único ciclo y no están sujetas a injerencias por parte de otras instrucciones. **TEST** y **SET**. La función TEST comprueba un valor, mientras que la función SET lo modifica.

Instrucción TEST&SET

```
1  booleano TS (int i){
2      if (i == 0) {
3          i = 1;
4          return cierto;
5      } else return falso;
6  }
```

Instrucción intercambiar

```
1  void intercambiar(int registro,
2          int memoria) {
3      int temp;
4      temp = memoria;
5      memoria = registro;
6      registro = temp;
7  }
```

Instrucción TEST&SET

```
1  const int n = N;
2  int cerrojo; // Variable compartida
3  void P(int i ){ // Proceso i-esimo
4      while (TRUE){
5          while(!(TS(cerrojo)); // Espera Activa, TS ==> Fc. Atómica HW
6          ----> SECCION CRITICA
7          cerrojo =0;
8        ---->RESTO DEL PROCESO
9      }
10 }
11 void main( ){
12     cerrojo=0; // Inicializa cerrojo
13     parbegin(P(1), P(2),...,P(N));// Lanzamiento de N Procesos Concurrentes
14 }
```

Existe otra función **intercambiar** que cambia un valor con otro.

Instrucción intercambiar

```
1  const int n = N;
2  int cerrojo; // Variable compartida
3  void P(int i ){
4       int clavei; // Variable local
5       while (TRUE){
6            clavei=1; //Inicializa clavei
7            while(clavei) intercambiar(clavei,cerrojo); // Espera Activa
8            ----> SECCION CRITICA
9            intercambiar(clavei,cerrojo); //Fc. atómica HW
10           ---->RESTO DEL PROCESO
11      }
12 }
13 void main( ){
14      cerrojo=0; // Inicializa cerrojo
15      parbegin(P(1), P(2),...,P(N)); // Lanzamiento de N Procesos Concurrentes
16 }
```

Las **ventajas** de las soluciones hardware son:

- Es aplicable a cualquier número de procesos en sistemas con memoria compartida, tanto de monoprocesador como de multiprocesador.
- Es simple y fácil de verificar.
- Puede usarse para varias secciones críticas.

Las **desventajas** son:

- Interbloqueo: Si un proceso con baja prioridad entra en su sección crítica y existe otro proceso con mayor prioridad, entonces el proceso cuya prioridad es mayor obtendrá el procesador para esperar a poder entrar en la sección crítica.
- La espera activa consume tiempo de procesador.

Las soluciones software para acabar con los problemas de concurrencia son:

1. **Semáforos**. Los procesos se coordinan mediante el traspaso de señales, esta señalización se tramita mediante una variable especial llamada semáforo. Los semáforos tienen 2 operaciones atómicas, **up** y **down**. Los procesos en espera de recibir una señal son bloqueados en una cola hasta que tenga lugar dicha señal. Dicha cola puede ser:
   - **Robusta** (semáforos robustos), la cual garantiza la no inanición mediante el uso de una cola FIFO. Linux.
   - **Débil** (semáforos débiles), no garantizan la no inanición. MAC OS.

   Un semáforo se puede ver como un valor entero el cual con down se decrementa, y con up aumenta. Si antes de hacer down el valor es 0 y se hace, el proceso se queda bloqueado hasta que otro proceso haga up.

   En general, los problemas de semáforos contienen 2 tipos, **mutex** para garantizar la exclusión mutua y de **recursos** para garantizar el uso de un número exacto de recursos.

semget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(int key, const void nsems, size_t semflg);
```

Accede o crea los semáforos identificados por key

Significado de los parámetros

- **key**: Puede ser IPC_PRIVATE o un identificador.
- **nsems**: Número de semáforos que se definen en el array de semáforos.
- **semflg**: Es un código octal que indica los permisos de acceso a los semáforos y se puede construir con un OR de los siguientes elementos:
  - IPC_CREAT: crea el conjunto de semáforos si no existe.
  - IPC_EXCL: si se usa en combinación con IPC_CREAT, da un error si el conjunto de semáforos indicado por *key* ya existe.
- Devuelve un identificador

semop

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

Realiza distintas operaciones sobre el array de semáforos

Significado de los parámetros

- **semid**: Identificador del conjunto de semáforos.
- **sops**: Conjunto de operaciones.
- **nsops**: Número de operaciones.

2. **Monitores**. Son módulos de software que constan de uno o más procedimientos, una secuencia de inicialización y unos datos locales. Las características básicas son:
   - Las variables del monitor solo son accesibles para este.
   - Un proceso entra en el monitor invocando a uno de sus procedimientos.
   - Solo un proceso se puede estar ejecutando en el monitor en un instante dado.

   Para sincronizar procesos se usan variables de condición solo accesibles desde el interior del monitor. Las funciones básicas son: **cwait** bloquea la ejecución de un proceso invocante y libera al monitor. **csignal** reanuda la ejecución de algún proceso bloqueado con un cwait sobre la misma condición.
3. **Mensajes**. Se utilizan como refuerzo para la exclusión mutua mediante el intercambio de información. La implementación del paso de mensajes se implementa mediante dos primitivas: **send** y **receive** argumentos (destino/origen, mensaje).
   El emisor y el receptor pueden ser bloqueantes o no bloqueantes (esperando a leer o a recibir). Existen varias combinaciones posibles:
   - Envío y recepción bloqueantes: Tanto el emisor y el receptor se bloquean hasta que se entrega el mensaje. Esta técnica se conoce como *rendezvous*.
   - Envío no bloqueante, recepción bloqueante: Permite que un proceso envíe uno o más mensajes a varios destinos tan rápido como sea posible. El receptor se bloquea hasta que llega el mensaje solicitado.
   - Envío y recepción no bloqueantes. Nadie espera.

   Hay 2 tipos de direccionamientos:

   - **Directo**. En las primitivas send y receive se incluye una identificación del proceso de receptor y emisor respectivamente. A parte receive puede utilizar el parámetro origen para devolver un valor cuando se haya realizado la operación de recepción.

- **Indirecto**. Los mensajes se envían a una estructura de datos compartida formada por colas (buzones). Un proceso envía un mensaje a un buzón y este los coge de dicho buzón.

msgsnd

#include <sys/msg.h>
int **msgsnd**(int *msqid*, const void **msgp*, size_t *msgsz*, int *msgflg*);
Envía un mensaje a la cola asociada con el identificador msqid

Significado de los parámetros

- *msgp* apunta a un buffer definido por el usuario, encabezado por un valor de tipo `long int` que indica el tipo del mensaje, seguido de los datos a enviar. Se puede implementar como una estructura:

```
struct mymsg {
  long mtype; /* tipo de mensaje */
  char mtext[1]; /* texto del mensaje */
}
```

Significado de los parámetros (continuación)

- *msgsz* indica el tamaño del mensaje, que puede ser hasta el máximo permitido por el sistema.
- *msgflg* indica la acción que se debe llevar a cabo si ocurre alguno de las siguientes circunstancias:
  - El número de bytes en la cola es ya igual a *msg_qbytes*.
  - El número total de mensajes en en colas del sistema es igual al máximo permitido.
- Si la función se ejecuta correctamente, la función devuelve un 0. En caso contrario -1.

Significado de los parámetros (continuación)

- Las acciones posibles en función de *msgflg* son:
  - Si (msgflg&IPC_NOWAIT) es distinto de 0 el mensaje no se envía y el proceso no se bloquea en el envío.
  - Si (msgflg&IPC_NOWAIT) es 0, se bloquea la ejecución del proceso hasta que ocurre uno del estos eventos:
    - Se solventa la condición que ha provocado el bloqueo, en cuyo caso el mensaje se envía.
    - *msqid* se elimina del sistema. Esto provoca el retorno de la función con un `errno` igual a `EIDRM`.
    - El proceso que está intentando enviar el mensaje recive una señal que debe capturar, por lo que el programa continúa como se le indique en la rutina correspondiente.

msgrcv

#include <sys/msg.h>
int **msgrcv**(int *msqid*, void **msgp*, size_t *msgsz*, long int *msgtyp*, int *msgflg*);
Lee un mensaje de la cola asociada con el identificador msqid y lo guarda en el buffer apuntado por msgp

Significado de los parámetros

- *msgp* apunta a un buffer definido por el usuario, encabezado por un valor de tipo `long int` que indica el tipo del mensaje, seguido de los datos a enviar. Se puede implementar como una estructura:

```
struct mymsg {
  long mtype; /* tipo de mensaje */
  char mtext[1]; /* texto del mensaje */
}
```

Significado de los parámetros (continuación)

- *msgtyp* indica el tipo del mensaje a recibir. Si es 0 se recibe el primero de la cola; si es >0 se recibe el primer mensaje de tipo *msgtyp*; si es <0 se recibe el primer mensaje del tipo menor que el valor absoluto de *msgtyp*.
- *msgsz* indica el tamaño en bytes del mensaje a recibir. El mensaje se trunca a ese tamaño si es mayor de msgsz bytes y (msgflg&MSG_NOERROR) es distinto de 0. La parte truncada se pierde.
- Si la función se ejecuta correctamente, la función devuelve un 0. En caso contrario -1.

Significado de los parámetros (continuación)

- *msgflg* indica la acción que se debe llevar a cabo no se encuentra un mensaje del tipo esperado en la cola. Las acciones posibles son:
  - Si (msgflg&IPC_NOWAIT) es distinto de 0 el proceso vuelve inmediatamente y la función devuelve -1.
  - Si (msgflg&IPC_NOWAIT) es 0, se bloquea la ejecución del proceso hasta que ocurre uno del estos eventos:
    - Se coloca un mensaje del tipo esperado en la cola.
    - *msqid* se elimina del sistema. Esto provoca el retorno de la función con un `errno` igual a `EIDRM`.
    - El proceso que está intentando enviar el mensaje recibe una señal que debe capturar, por lo que el programa continúa como se le indique en la rutina correspondiente.