TEMA 2: PROCESOS E HILOS: CONCURRENCIA, SINCRONIZACIÓN Y COMUNICACIÓN

1. Introducción

Definición de proceso (tarea según los fabricantes):

- Programa en ejecución.
- Cálculo computacional que puede hacerse concurrentemente con otros cálculos.
- Secuencia de acciones llevadas a cabo a través de la ejecución de una serie de instrucciones cuyo resultado final consiste en proveer alguna función del sistema.
- Trabajo mínimo que es susceptible de ser planificado por un S.O.

Diferencia entre proceso y programa:

- Un programa lo forma sólo un código. Es un concepto estático.
- Un proceso lo forma el programa, la pila, los registros del procesador (como puede ser el Contador de Programa (CP), Acumuladores, ...). Es un concepto dinámico.

Un proceso puede involucrar la ejecución de más de un programa.
Por ejemplo:

- El sistema operativo es un programa que crea varios procesos en la computadora donde se ejecuta.
- Los programas multitarea.

Un determinado programa puede estar involucrado en más de un proceso.
Por ejemplo:

- El código reentrante de un compilador.

La concurrencia es la activación de varios procesos a la vez. La concurrencia se consigue haciendo que:

(1) Los procesadores se asignen a distintos procesos (paralelismo o concurrencia real).
(2) Se alternen varios procesos en un mismo procesador (concurrencia aparente o simulada).

Sólo tendremos concurrencia real en sistemas multiprocesador (en un sistema uniprocesador existe una concurrencia aparente).

El procesado concurrente o paralelo aparece cuando varios procesos se encuentran en un instante determinado en un estado intermedio entre sus estados inicial y final.

La programación concurrente es mucho más difícil que la secuencial. Los programas concurrentes son más difíciles de escribir, depurar y modificar, y es más difícil probar que son correctos. La programación concurrente ha despertado un gran interés porque permite expresar de una forma natural las soluciones a ciertos problemas de carácter inherentemente paralelo y también por el paralelismo de hardware que se logra con los multiprocesadores y sistemas distribuidos.

Estados básicos de un proceso:

- **Entregado**: Cuando un proceso pretende iniciar su ejecución.
- **En ejecución**: Ese proceso tiene el control de la CPU.
- **Listo o preparado**: El proceso se encuentra en memoria principal, sin operaciones de E/S pendientes y listo para entrar en ejecución cuando le corresponda.
- **Terminado**: El proceso ha terminado su ejecución y se ha liberado la memoria principal en la que se ubicaba.

Transiciones entre los estados básicos de un proceso:

- **create**: Cuando un proceso ha sido creado.
- **schedule**: Cuando un proceso ha sido planificado (se le asigna la CPU).
- **reschedule**: Cuando un proceso va a volver a planificarse.
- **end**: Cuando el proceso en ejecución ejecuta una instrucción de fin de programa o cuando recibe una señal de fin de ejecución.

Los procesos concurrentes funcionan de forma totalmente independiente unos de otros, lo que quiere decir que se ejecutan de forma asíncrona, pero en ocasiones requieren cierta sincronización y cooperación.

2. Conceptos básicos

Veamos los conceptos básicos relacionados con la sincronización y comunicación entre procesos concurrentes asíncronos.

2.1. Condiciones de carrera

Ocurre cuando dos o más procesos que están trabajando juntos comparten algún dato modificable (por ejemplo, una posición de memoria de lectura/escritura) que cada proceso puede leer o escribir y el resultado final depende del orden de ejecución. (**Nota:** Este problema no sucede en monoprogramación).

Ejemplo 1: Reserva de plazas en una compañía aérea.

Un asiento se representa por el contenido de una posición de memoria (se puede identificar con una variable de tipo array) que es accesible a más de un proceso.

Agente A	Agente B
Comprueba que la plaza está libre	Comprueba que la plaza está libre
Consulta a su cliente	Consulta a su cliente
.....	.....
Reserva el asiento	Reserva el asiento

Ejemplo 2: Spooler de impresora.

El directorio del Spooler tiene un número de ranuras, una para cada nombre del archivo a imprimir.

Se utilizan dos variables compartidas por los procesos para la gestión del directorio:

- OUT: Apunta al próximo archivo a imprimir.
- IN: Apunta a la próxima ranura libre en el directorio.

Supongamos que dos procesos deciden encolar un archivo para su impresión de forma casi simultanea:

A	**B**
Lee IN y almacena el valor 7 en una variable local.	
(Se produce una interrupción)	Lee IN, obtiene 7, almacena el nombre de su archivo de impresión en la ranura 7 y pone IN = 8.
(vuelve el control de la CPU a A)	
Escribe su nombre de archivo de impresión en la ranura 7 (machacando el de B) y pone IN = 8.	

El archivo de impresión creado por B nunca se imprimirá.

2.2. Exclusión mutua. Regiones (Secciones) críticas y regiones (secciones) críticas condicionales

Los procesos compiten por el uso de unos recursos limitados. Esos recursos pueden ser:

- Compartibles: Pueden ser empleados por varios procesos de forma concurrente. Ejemplos: CPU, archivos de lectura, áreas de memoria no modificables, ...
- No compartibles: Su uso se restringe a un único proceso por:
    - La naturaleza física del recurso. Ejemplos: Unidad de cinta magnética, lectora de tarjetas, impresora, ...
    - Si el recurso lo usan otros procesos de forma concurrente, la acción de uno de ellos puede interferir en la de otro. Ejemplos: Archivo de escritura, posición de memoria, ...

La exclusión mutua consiste en asegurar que los recursos no compartibles sean accedidos por un único proceso a la vez.

Para evitar las condiciones de carrera necesitamos exclusión mutua, es decir, si un proceso está usando una variable o archivo compartido, el otro proceso será excluido de hacer lo mismo.

Las secciones críticas o regiones críticas son fragmentos de programa que acceden a recursos no compartibles. Si dos procesos no están nunca en sus secciones críticas al mismo tiempo, se evitan las condiciones de carrera (esto no impide tener procesos paralelos que cooperen correctamente y eficientemente usando datos compartidos).

Las regiones críticas condicionales permiten que los procesos ejecuten sus secciones críticas sólo cuando se cumpla una determinada condición.

Cuando a un proceso no se le satisface la condición, se quedará suspendido en una cola especial en espera del cumplimiento de la condición. Así se consigue que, aunque un proceso espere en una condición, no evite que otros utilicen el recurso. Con ello conseguimos la sincronización además de la exclusión mutua a la sección crítica.

Condiciones que se requieren para evitar las condiciones de carrera:

- Dos procesos no pueden estar simultáneamente dentro de sus secciones críticas.
- No se hacen suposiciones sobre las velocidades de los procesos o el número de CPUs.
- Ningún proceso que se pare fuera de su sección crítica bloqueará a otros procesos.
- Ningún proceso esperará mucho tiempo para entrar en su sección crítica. Por ello, las secciones críticas deben ejecutarse tan rápido como sea posible. Además, un proceso no debe bloquearse dentro de su propia sección crítica (interbloqueo).

2.3. Sincronización

Cada proceso se ejecuta asíncronamente con respecto a otro (son impredecibles las frecuencias de reloj asociadas).

En algunos instantes, los procesos deben sincronizar sus actividades; por ejemplo, en el caso de que un proceso no pueda progresar hasta que otro haya terminado algún tipo de actividad.

2.4. Comunicación

La comunicación permite que dos o más procesos puedan intercambiar información. Las dos formas de realizar esto son mediante la memoria compartida y mediante el paso de mensajes. Todo lo demás será sincronización. En este tema se verá sólo el paso de mensajes.

2.5. Interbloqueos

A veces se dan situaciones en las que los procesos no pueden progresar debido a que los recursos que cada uno de ellos necesita están ocupados por los otros.

3. Primitivas de sincronización y comunicación

Las primitivas son capacidades adicionales al hardware proporcionadas por las funciones del núcleo del sistema operativo.

Vamos a buscar primitivas de comunicación entre procesos que bloqueen en lugar de gastar tiempo de CPU.

3.1. Semáforos

Fueron introducidos por Dijkstra en 1965.

Un semáforo es un entero no negativo sobre el que puede actuarse sólo a través de las operaciones WAIT y SIGNAL tras su inicialización.

```
WAIT (s) :  SI (s?0) ENTONCES s = s - 1
                DE LO CONTRARIO  añadir proceso a la cola del semáforo y
                                 hacerlo no ejecutable (bloqueado)
```

SIGNAL (s) : SI (cola del semáforo vacía) ENTONCES s = s + 1
DE LO CONTRARIO sacar proceso de cola del semáforo y ponerlo en estado preparado.

El proceso que ejecute un WAIT sobre un semáforo a 0 quedará bloqueado hasta que otro proceso ejecute una operación SIGNAL sobre el mismo semáforo.

Si hay varios procesos bloqueados en el mismo semáforo, sólo uno de ellos podrá desbloquearse (normalmente, en orden FIFO).

Las operaciones WAIT y SIGNAL son indivisibles, por lo que no existe posibilidad alguna de que dos procesos actúen al mismo tiempo sobre el mismo semáforo.

Los semáforos se utilizan para la sincronización entre procesos.

Valor del semáforo:

val (sem) = C (sem) + ns (sem) - nw (sem)

donde C = valor inicial

ns = número de operaciones SIGNAL llevadas a cabo

nw = nº de operaciones WAIT llevadas a cabo con éxito

Con val (sem) >= 0 tenemos que:

nw (sem) <= C (sem) + ns (sem)

La igualdad sólo se cumple cuando **val (sem) = 0**.

Esta relación es invariante frente al número de operaciones WAIT y SIGNAL efectuadas.

Mediante el uso de semáforos puede ocurrir que un proceso quede bloqueado temporalmente. Por tanto, debemos reflejar este nuevo estado de los procesos:

- **Bloqueado**: Cuando un proceso se encuentra esperando en la cola de un semáforo a que se realice una operación SIGNAL sobre dicho semáforo.

Transiciones:

- **wait**: Cuando el proceso en ejecución ha ejecutado una operación WAIT sobre un semáforo cuyo valor es 0.
- **signal**: Cuando un proceso en ejecución ejecuta una operación SIGNAL sobre un semáforo cuya cola de procesos no está vacía.

Ejemplo 1: Implementación de la exclusión mutua con semáforos

Al principio y al final de cada sección crítica se colocan, respectivamente, una operación WAIT y otra SIGNAL sobre el mismo semáforo, cuyo valor inicial debe ser 1.

```
.............
WAIT (exmut);
S.C.;                                nw (exmut) <= ns (exmut) + 1
SIGNAL (exmut);
.............
```

Sólo un proceso como máximo podrá ejecutar WAIT (exmut) con éxito antes de que otro lleve a cabo una operación SIGNAL (exmut). Se desautoriza, por tanto, la entrada de un proceso a su S.C. sólo cuando otro proceso esté dentro de la suya.

Ejemplo 2: Sincronización asimétrica de procesos

Un proceso A no puede ir más allá de un determinado punto hasta que otro proceso B no haya alcanzado otro punto (A necesita información que debe suministrarle B y, por tanto, A está regulado por B).

seguir: semáforo inicializado a cero.

Proceso A	Proceso B
.....	.....
WAIT (seguir);	SIGNAL (seguir);
.....	.....

nw (seguir) <= ns (seguir)

Ejemplo 3: Interbloqueo ("deadlock")

Sean dos semáforos **x** e **y** inicializados a 1.

Proceso A	Proceso B
.....	.....
WAIT (x);	WAIT (y);
.....	.....
WAIT (y);	WAIT (x);
.....	.....

- Asociamos cada semáforo con un recurso. A y B retienen el recurso que solicita el otro.
- La compartibilidad de un recurso (semáforo) viene dada por su valor inicial. Si se inicializa a un valor N mayor que 1, entonces el recurso podrá ser compartido por N procesos.
- El interbloqueo puede aparecer como consecuencia de una secuencia incorrecta de operaciones WAIT y SIGNAL lo que introducirá una dificultad adicional en el diseño de programas.

Ejemplo 4: Problema de los procesos productores y consumidores que se comunican a través de un búfer que es llenado por los productores y vaciado por los consumidores.

Sea un búfer acotado (finito) compartible por un productor y un consumidor. El productor coloca información en el búfer y el consumidor la extrae. Si el búfer está lleno, el productor se va a dormir y, si está vacío, el consumidor se va a dormir.

El búfer sólo tiene capacidad para N elementos del mismo tamaño.

Semáforos:

- Manipulación del búfer (**MB**) = 1. Realiza la exclusión mutua para proteger al búfer de la interferencia entre procesos.
- Espacio libre (**EL**) = N.
- Elementos disponibles (**ED**) = 0.

  Estos dos últimos semáforos se utilizan para la sincronización ya que:
  - Los productores no pueden colocar elementos en el búfer si está lleno.
  - Los consumidores no pueden extraer elementos del búfer si está vacío.

PRODUCTOR (i)

```
REPETIR
  COMIENZO
    producir un elemento;
    WAIT (EL);
    WAIT (MB);
      depositar elemento en el búfer;
    SIGNAL (MB);
    SIGNAL (ED);
  FIN;
```

CONSUMIDOR (j)

```
REPETIR
  COMIENZO
    WAIT (ED);
    WAIT (MB);
      sacar elemento del búfer;
    SIGNAL (MB);
    SIGNAL (EL);
    consumir elemento;
  FIN;
```

Hay que asegurar la relación: 0 <= el. almacenados - el. extraídos <= N

Caso práctico: *Cons.* : Spooler *Prod.* : Proceso *Elemento* : Archivo

Ejemplo 5: La cena de los filósofos

Fue propuesto y resuelto por Dijkstra en 1965 y desde entonces se considera un problema clásico de sincronización, no por su utilidad práctica, sino porque es un ejemplo para un gran conjunto de problemas de sincronización.

Definición del problema: Cinco filósofos pasan su vida pensando y comiendo. Comparten una mesa circular rodeada de 5 sillas, perteneciente cada una de ellas a un filósofo. En el centro de la mesa hay un recipiente con arroz (puede sustituirse por espaguetis, migas, ...) y la mesa dispone de 5 tenedores. Cuando un filósofo piensa, no interactúa con sus colegas. De vez en cuando, un filósofo tiene hambre e intenta coger los dos tenedores que tiene a cada lado (izquierdo y derecho). Un filósofo tan solo puede coger un tenedor en cada momento, y no puede coger un tenedor que ya ha cogido su "vecino" (con el que lo comparte). Si un filósofo consigue coger los dos tenedores, comienza a comer sin dejarlos. Cuando termina de comer, los deja en su sitio y vuelve a pensar de nuevo.

Una posible solución es asociar un semáforo a cada filósofo y utilizar un semáforo binario para conseguir la exclusión mutua cuando se accede a los tenedores de un filósofo.

Semáforos:

- **exmut**, inicializado a 1 (para la exclusión mutua)
- **s[i]**, i = 0, 1, ..., 4, inicializados a 0 (uno por filósofo)

Variables compartidas:

- **estado[i]**, i = 0, 1, ..., 4, inicializadas a 0 (una por filósofo). Puede tomar los valores 0 (pensando), 1 (hambriento) y 2 (comiendo).
- **izquierdo[i]** y **derecho[i]**, i = 0, 1, ..., 4, vectores constantes que permiten localizar los filósofos a izquierda y derecha de cada uno.

filósofo (i)

```
REPETIR
    pensar;
    WAIT (exmut);
    estado[i] = 1;
    SI (estado[i] = 1  Y  estado[izquierdo[i]] ? 2  Y  estado[derecho[i]] ? 2) ENTONCES
        COMIENZO
            estado[i] = 2;
            SIGNAL (s[i])
        FIN
    SIGNAL (exmut);
    WAIT (s[i]);                    // Bloqueo si no se cogen los dos tenedores.
    comer;
    WAIT (exmut);
    estado[i] = 0;
    SI (estado[izquierdo[i]] = 1  Y  estado[izquierdo[izquierdo[i]]] ? 2  Y  estado[i] ? 2) ENTONCES
        COMIENZO
            estado[izquierdo[i]] = 2;
            SIGNAL (s[izquierdo[i]])
        FIN
    SI (estado[derecho[i]] = 1  Y  estado[derecho[derecho[i]]] ? 2  Y  estado[i] ? 2) ENTONCES
        COMIENZO
            estado[derecho[i]] = 2;
            SIGNAL (s[derecho[i]])
        FIN
    SIGNAL (exmut);
```

Ejemplo 6: Problema de los lectores/escritores

Un objeto de datos (registro o archivo) es compartido por varios procesos concurrentes. Algunos tan solo pretenden leer mientras que otros pueden actualizar los datos (leer y escribir); nos referimos a estos procesos como lectores y escritores. Obviamente, si dos lectores intentan acceder a los datos simultáneamente, no se presentará ningún problema. Sin embargo, si un escritor y algún otro lector o escritor quieren acceder simultáneamente a

los datos, entonces puede producirse un caos. Con objeto de evitar estos problemas, se ha de exigir que los escritores accedan de forma exclusiva a los datos compartidos. Es un problema típico de acceso a bases de datos.

Existen diversas variantes de este problema, la más simple de las cuales se conoce como Primer problema de lectores y escritores, en el que se postula que ningún lector esperará a menos que un escritor haya obtenido permiso para acceder a los datos compartidos (prioridad para los lectores). El Segundo problema de lectores y escritores postula que si un escritor está esperando para acceder a los datos, ningún nuevo lector puede iniciar la lectura (prioridad para los escritores). Ambas variantes pueden provocar que un proceso permanezca indefinidamente bloqueado.

Consideremos la solución al primer problema (prioridad para los lectores):

Semáforos:

- **exmut**, inicializado a 1 (para la exclusión mutua)
- **bd**, inicializado a 1 (controla el acceso a la base de datos)

Variables compartidas (sólo para los lectores):

- **re**, inicializada a 0 (número de procesos leyendo la base de datos)

lector (i)

```
REPETIR
   WAIT (exmut);
   re = re + 1;
   SI (re = 1) ENTONCES WAIT (bd);
   SIGNAL (exmut);
   Leer base de datos;
   WAIT (exmut);
   re = re - 1;
   SI (re = 0) ENTONCES SIGNAL (bd);
   SIGNAL (exmut);
   usar datos leídos;
```

escritor (j)

```
REPETIR
   Pensar datos;
   WAIT (bd);
   Escribir en la base de datos;
   SIGNAL (bd);
```

El semáforo **exmut** se utiliza para asegurar la exclusión mutua cuando se actualiza la variable **re**. El semáforo **bd** se utiliza para la exclusión mutua de los escritores, y también lo utilizan el primer lector que entra en su sección crítica y el último que sale de ella.

Para evitar el problema de la espera indefinida, Hoare planteó la siguiente solución: a) Un nuevo lector no comienza si un escritor está esperando, y b) todos los lectores esperando tras un escritor tendrán prioridad sobre el siguiente escritor.

Semáforos:

- EM = 1 (Exclusión Mutua)
- L = 0 (Lectores)
- E = 0 (Escritores)

Variables compartidas:

- NL = 0 (Nº de procesos leyendo)
- NLE = 0 (Nº de lectores esperando)
- NE = 0 (Nº de procesos escribiendo)
- NEE = 0 (Nº de escritores esperando)

Lector (i)

```
WAIT (EM);
SI ((NE=0) AND (NEE=0) ENTONCES
   NL = NL + 1;
   SIGNAL (L);
DE LO CONTRARIO
   NLE = NLE + 1;
SIGNAL (EM);
WAIT (L);
Leer la base de datos;
WAIT (EM);
NL = NL - 1;
SI (NL = 0) ENTONCES
   SI (NEE ? 0) ENTONCES
      NE = 1;
      NEE = NEE - 1;
      SIGNAL (E);
SIGNAL (EM);
usar los datos leídos;
```

Escritor (j)

```
WAIT (EM);
SI ((NL=0) AND (NE=0) ENTONCES
   NE = 1;
   SIGNAL (E);
DE LO CONTRARIO
   NEE = NEE + 1;
SIGNAL (EM);
WAIT (E);
Modificar la base de datos;
WAIT (EM);
NE = 0;
SI (NLE ? 0) ENTONCES
   j = 0;
   MIENTRAS (j < NLE) HACER
      SIGNAL (L);
      j = j + 1;
   NL = NLE;
   NLE = 0;
DE LO CONTRARIO
   SI (NEE ? 0) ENTONCES
      NE = 1;
      NEE = NEE - 1;
      SIGNAL (E);
SIGNAL (EM);
```

3.2. Paso de mensajes

Dividir una actividad lógica en una serie de procesos que se puedan ejecutar concurrentemente puede producir un mejor rendimiento. Para realizar funciones colectivas, los procesos cooperantes deben intercambiar datos (comunicación) y sincronizarse entre ellos (sincronización).

Los semáforos resuelven el problema de una o más CPUs que acceden a una memoria común, pero se vuelven inaplicables en un sistema distribuido compuesto de varias CPUs cada una con su memoria privada y conectadas por una red local.

Los mensajes son mecanismos sencillos para la sincronización y comunicación entre procesos en entornos centralizados y distribuidos (el envío y la recepción de mensajes es una forma estándar de comunicación en sistemas distribuidos).

Un mensaje se puede definir como una colección de información que se puede intercambiar entre un proceso emisor y otro receptor. Un mensaje puede contener datos, órdenes e incluso código.

El paso de mensajes utiliza dos primitivas que son llamadas al sistema:

- SEND (destino, mensaje) : Envía un mensaje a otro proceso.
- RECEIVE (origen, mensaje) : Bloquea un proceso hasta que le llegue un mensaje.

Decisiones que se toman en la implementación de mensajes (características de los mensajes):

(1) **Designación** del mensaje. Puede ser *directa* (se debe indicar explícitamente el proceso que debe recibir o transmitir el mensaje) o *indirecta* (mediante el uso de buzones, indicando así a qué buzón se envía el mensaje o desde qué buzón se recibe).

(2) **Intercambio** de mensajes. Decidir si el mensaje se intercambia mediante una "*copia*" del contenido del mismo desde el espacio de direcciones del transmisor al receptor, o bien pasando un puntero al mensaje entre los dos procesos (paso de mensaje por "*referencia*").

(3) **Almacenamiento intermedio**. Decidir si los mensajes enviados por un proceso pero no recibidos todavía deben ser *almacenados* o *no.*
Los mensajes se pueden perder en la red. El receptor devolverá un mensaje de reconocimiento especial que, si no es recibido por el emisor dentro de un cierto intervalo de tiempo, debe retransmitirlo.
Si se pierde el mensaje de reconocimiento, el emisor retransmite el mensaje y el receptor lo obtiene dos veces, por lo que habrá que distinguir entre un mensaje nuevo y uno retransmitido (se utilizan números de secuencia consecutivos en cada mensaje original).

(4) **Longitud** del mensaje. Puede ser *fija* o *variable*.

Consideraciones de diseño para sistemas que utilizan paso de mensajes:

- **Nombres de los procesos**: proceso@máquina.dominio
  Las máquinas se agrupan en dominios, evitándose así el problema de que dos máquinas tengan el mismo nombre en distintos sitios.
- **Autentificación**: Se cifran (criptografía) los mensajes con claves conocidas sólo por usuarios autorizados.

Debido a que un proceso pueden encontrarse en un momento determinado en espera de recibir un mensaje, aparecerá un nuevo estado posible para un proceso:

- **Recibiendo**: Cuando un proceso espera la recepción de un mensaje.

Transiciones:

- **receive**: Cuando el proceso está esperando un mensaje.
- **send**: Cuando el proceso recibe el mensaje.

Ejemplo: Problema del productor - consumidor

Productor

```
REPETIR
    Producir elemento;
    RECEIVE (Consumidor, mensaje);
    Construir mensaje;
    SEND (Consumidor, mensaje);
```

Consumidor

```
FOR i = 1 TO N
    SEND (Productor, mensaje);
REPETIR
    RECEIVE (Productor, mensaje);
    Extraer mensaje;
    Consumir elemento;
    SEND (Productor, mensaje);
```

- Inicialmente, el consumidor envía N mensajes (simulando un búfer de N posiciones libres).

- La línea del proceso consumidor

  SEND (Productor, mensaje);

  se utiliza para enviar un mensaje vacío una vez que se ha leído un mensaje lleno del productor.

- El consumidor envía mensajes vacíos, y el productor toma un mensaje vacío y lo envía lleno.

- En la implementación anterior se ha considerado una designación directa de los mensajes, es decir, en cada operación SEND o RECEIVE se especifica el proceso hacia el que va dirigido o desde el que se debe recibir el mensaje. De ahí que esta implementación sólo sea válida para un proceso productor y un proceso consumidor.

- Para implementar una solución correcta para N procesos productores y M procesos consumidores (con N y M mayores que 1), se debe utilizar una designación indirecta de los mensajes (mediante buzones).