Concurrencia

Programación Concurrente

Espera ocupada.

Primitivas IPC con bloqueo

Programación concurrente

Los lenguajes concurrentes tienen elementos para:

- Crear procesos
- Sincronizar procesos
- Comunicar procesos

Comportamiento de procesos

- Independientes: no se sincronizan ni comunican (son muy raros).

•Cooperativos: se comunican y sincronizan sus actividades.

•Competitivos: compiten por recursos del sistema.

Sincronizar y Comunicar

- Sincronizar: Satisfacer las restricciones en el enlazado de las acciones de los distintos procesos.
- Comunicar: pasar información de un proceso a otro.

Sincronizar y Comunicar

- Variables compartidas: objetos a los que puede acceder más de un proceso

- Paso de mensajes: intercambio explícito de datos entre dos procesos mediante el paso de un mensaje mediante alguna forma que brinda el SO o el propio lenguaje.

Modelo de concurrencia

- Estructura: nro de procesos fijo o variable.
- Nivel: paralelismo soportado.
- Granularidad: muchos o pocos procesos.
- Inicialización: paso de parámetros, o comunicación explícita después de su ejecución
- Finalización: término, error, aborto, nunca, suicidio, no son necesarios
- Representación: proceso responsable de la creación (padre/hijo) y proceso afectado por su finalización (guardián/dependiente).

Variables compartidas

Considere dos procesos que actualizan una variable compartida, X, mediante la sentencia: X:= X+1

1. Carga el valor de X en algún registro.
2. Incrementa el valor en el registro en 1.
3. Almacena el valor del registro de nuevo en X.

Como ninguna de las tres operaciones es indivisible, dos procesos que actualicen la variable simultáneamente generarían un entrelazamiento que podría producir un resultado incorrecto.

Variables compartidas

- Las situaciones donde los resultados dependen del orden en que se ejecutan los procesos se llaman **Condiciones de Competencia**.
- Las partes de un proceso que tienen acceso a las variables compartidas han de ejecutarse indivisiblemente unas respecto a las otras.
- Estas partes se denominan **Secciones Críticas**.
- La protección requerida se conoce como **Exclusión Mutua.**

Secciones o regiones críticas

- Para obtener una solución se deben cumplir:
  1. Ningún par de procesos pueden estar simultáneamente dentro de sus regiones críticas.
  2. No debe hacerse ninguna suposición sobre la velocidad o el número de CPUs.
  3. Ningún proceso fuera de su región crítica puede bloquear a otros procesos.
  4. Ningún proceso deberá tener que esperar infinitamente para entrar en su región crítica.

Exclusión Mutua con Espera Ocupada

- Posibles soluciones:
  - Inhabilitación de interrupciones
  - Variables de cerradura
  - Alternancia estricta
  - Solución de Peterson

Inhabilitación de Interrupciones

- Se inhabilitan las interrupciones antes de entrar a la sección crítica. Se rehabilitan al salir.
- La CPU no podrá interrumpir la ejecución por timeout.
- Ventaja: muy simple
- Desventajas:
  - Los procesos de usuario no pueden deshabilitar las interrupciones
  - Si hay más de un procesador no funciona, pues la inhabilitación afecta a un solo CPU.

Variables de Cerradura

- Variable compartida inicialmente en 0.
- Antes de entrar a su sección crítica un proceso chequea la bandera:
  - Si está 0, el proceso la setea en 1 y entra a su SC.
  - Si está en 1, espera hasta que se ponga en 0.
- Desventajas:
  - Espera ocupada.
  - No funciona por condiciones de competencia.

Alternancia Estricta

```
while (TRUE) {
   while (turno != 0) /* nada */ ;
   region_critica() ;
   turno = 1 ;
   region_no_critica() ;
}
```

```
while (TRUE) {
   while (turno != 1) /* nada */ ;
   region_critica() ;
   turno = 0 ;
   region_no_critica() ;
}
```

- Desventajas:
  - Ineficiente si un proceso es más lento que el otro.
  - Viola la condición 3.
  - Espera ocupada.

Solución de Peterson

```
int turno ;
int interesado[2] ;

void entrar_en_region ( int proceso )
{
     int otro ;
     otro = 1 – proceso ;
     interesado[proceso] = TRUE ;
     turno = proceso ;
     while (turno == proceso
          && interesado[otro] == TRUE) ;
}

void abandonar_region ( int proceso )
{
     interesado[proceso] = FALSE ;
}
```

Primitivas de IPC con bloqueo

- La solución de Peterson es correcta, pero tiene el defecto de requerir espera ocupada:
  - Cuando un proceso quiere entrar en su región crítica, comprueba si se le permite la entrada
  - Si no es así, el proceso se mete en un bucle vacío esperando hasta que sí se le permita entrar.
- Este método no sólo gasta tiempo de CPU, sino que también puede tener efectos inesperados.

Inversión de Prioridades

- Consideremos un ordenador con dos procesos, *H* con alta prioridad y *L* con baja prioridad.
- Las reglas de planificación son tales que *H* pasa a ejecución inmediatamente siempre que se encuentre en estado listo.
- En un cierto momento, estando *L* en su región crítica, *H* pasa al estado listo (por ejemplo, debido a que se completa una operación de E/S que lo mantenía bloqueado).

Inversión de Prioridades

- De inmediato *H* comienza la espera activa, pero ya que *L* nunca se planifica mientras *H* esté ejecutándose, *L* nunca tendrá la oportunidad de abandonar su región crítica, con lo que *H* quedará para siempre dando vueltas al bucle de espera activa.
- Este es el problema de la Inversión de Prioridades.

Primitivas de IPC con bloqueo

- Son primitivas que bloquean al proceso que las invoca.
- Permite a la CPU continuar sin desperdiciar tiempo como en la espera ocupada.

Dormir y despertar.

Semáforos.

Monitores.

Métodos sincronizados.

Dormir y Despertar

- **Sleep** es una llamada al sistema que provoca que el proceso que la invoca se bloquee, esto es, se suspenda hasta que otro proceso lo despierte.
- **Wakeup** tiene un parámetro, que es el proceso a ser despertado.

El problema del Productor-Consumidor

- Son aquellos problemas en los que existe un conjunto de procesos que *producen* información que otros procesos *consumen,* siendo diferentes las velocidades de producción y consumo de la información.
- Este desajuste en las velocidades, hace necesario que se establezca una sincronización entre los procesos de manera que la información no se pierda ni se duplique, consumiéndose en el orden en que es producida.

El problema del Productor-Consumidor

```
#define N 100
int contador = 0 ;

void productor (void)
{
   int elemento ;

   while (TRUE) {
      elemento = producir_elemento( ) ;
      if (contador == N) sleep( ) ;
      meter_elemento(elemento) ;
      contador = contador + 1 ;
      if (contador == 1) wakeup(consumidor) ;
   }
}
```

```
void consumidor ( void )
{
   int elemento ;

   while (TRUE) {
      if (contador == 0) sleep( ) ;
      elemento = sacar_elemento( ) ;
      contador = contador – 1 ;
      if (contador == N – 1) wakeup(productor) ;
      consumir_elemento(elemento) ;
   }
}
```

código

El problema del Productor-Consumidor

- Existen condiciones de competencia.
- La esencia del problema es la pérdida de una señal enviada para despertar a un proceso que no estaba (todavía) dormido.

El problema del Productor-Consumidor

- Posible solución: añadir un bit de espera por la señal que despierta al proceso (***wakeup waiting bit***).
- Este bit se activa cuando se envía una señal para despertar a un proceso que todavía está despierto. Posteriormente, cuando el proceso intente dormirse, si encuentra ese bit activo, simplemente lo desactiva permaneciendo despierto.
- Existen ejemplos con tres o más procesos en los cuales un único bit de este tipo es insuficiente.

Semáforos

- Los introdujo Dijkstra en 1968.
- Permiten resolver la mayoría de los problemas de sincronización entre procesos y forman parte del diseño de muchos sistemas operativos y de lenguajes de programación concurrentes.
- La idea es utilizar una variable entera para contar el número de señales enviadas para despertar un proceso guardadas para su uso futuro.

Semáforos

- Introdujo un nuevo tipo de variable, denominado **semáforo**.
- El valor de un semáforo puede ser 0, indicando que no se ha guardado ninguna señal, o algún valor positivo de acuerdo con el número de señales pendientes para despertar al proceso.
- Operaciones sobre los semáforos: **down** y **up** (las cuales generalizan las operaciones sleep y wakeup, respectivamente).

Semáforos

- Está garantizado que una vez que comienza una operación sobre un semáforo, ningún otro proceso puede acceder al semáforo hasta que la operación se completa o hasta que el proceso se bloquea.
- La comprobación del valor del semáforo, su modificación y la posible acción de dormirse, se realizan como una única **acción atómica** indivisible.Esta atomicidad es absolutamente esencial para resolver los problemas de sincronización y evitar que se produzcan condiciones de competencia.

Semáforos

- La operación **up** incrementa el valor del semáforo al cual se aplica.
- Si uno o más procesos estuviesen dormidos sobre ese semáforo, el sistema elige a uno de ellos (por ejemplo de forma aleatoria) y se le permite continuar.
- El proceso que usa un despertar acumulado, decrementa el semáforo.

Semáforos

```
public final class Semaforo {
        String nombre;
        int s;

        public synchronized void down() {
                if (s == 0)
                        wait();
                s--;
        }

        public synchronized void up() {
                s++;
                notify();
        }
}
```

Semáforos

- La operación de **up** nunca bloquea al proceso que la ejecuta, de la misma forma que en el modelo anterior nunca se bloquea un proceso ejecutando una operación wakeup.

P()
down(S)
sleep(S)
wait(S)
Espera(S)

V()
up(S)
wakeup(S)
notify()
Signal(S)

Solución al problema del Productor-Consumidor con semáforos

```
tuberia.full.down();
tuberia.mutex.down();
c = tuberia.consumir();
tuberia.mutex.up();
tuberia.empty.up();
```

```
tuberia.empty.down();
tuberia.mutex.down();
tuberia.producir(c);
tuberia.mutex.up();
tuberia.full.up();
```

código

Solución al problema del Productor-Consumidor con semáforos

- Hemos utilizado los semáforos de dos formas muy distintas:
  - El semáforo *mutex* se utiliza para conseguir la exclusión mutua. Está diseñado para garantizar que solamente un proceso esté en cada momento leyendo o escribiendo en el búfer y sus variables asociadas. Cada proceso hace un **down** justo antes de entrar en su región crítica, y un **up** justo después de abandonarla
  - El otro uso de los semáforos es la **sincronización**.

Características de los Semáforos

- La manera normal es implementar **down** y **up** como llamadas al sistema, encargándose el sistema operativo de inhibir brevemente todas las interrupciones mientras comprueba el valor del semáforo, lo actualiza y bloquea el proceso, si es necesario.
- Como todas estas acciones requieren pocas instrucciones, no se provoca ningún daño al sistema inhibiendo las interrupciones durante ese corto lapso de tiempo.

Críticas a los semáforos

- Es difícil programar con semáforos.
- Si el semáforo se ubicó en un lugar erróneo falla (deadlock).

No se garantiza la exclusión mutua

Posible solución : monitores

Monitores

- Para hacer más fácil escribir programas correctos, Hoare (1974) y Brinch Hansen (1975) propusieron una primitiva de sincronización de alto nivel denominada **monitor**.
- Sus propuestas difieren ligeramente.

Monitores

- Un monitor es una colección de procedimientos, variables y estructuras de datos que están todos agrupados juntos en un tipo especial de módulo o paquete.
- Los procesos pueden llamar a los procedimientos de un monitor siempre que quieran, pero no se les permite acceder directamente a las estructuras de datos internas del monitor desde procedimientos declarados fuera del monitor.

Monitores

- En cualquier instante solamente un proceso puede estar activo dentro del monitor.
- Los monitores son construcciones del lenguaje, por lo que el compilador sabe que son especiales, de manera que puede tratar las llamadas a los procedimientos del monitor de forma diferente que a otras llamadas a procedimientos normales.

Monitores

- Si otro proceso está actualmente activo dentro del monitor, el proceso que hizo la llamada debe suspenderse hasta que el otro proceso abandone el monitor.
- Si ningún otro proceso está utilizando el monitor, el proceso que hizo la llamada puede entrar inmediatamente.

Críticas a monitores

- Solución elegante a problemas de exclusión mutua.
- Sólo lo implementan muy pocos lenguajes: Pascal Concurrente y Java (métodos sincronizados).
- No proporciona intercambio de información entre diferentes máquinas.

Métodos sincronizados

- Es el concepto de monitor implementado en el paradigma de Orientación a Objetos
- Se utilizan en lenguajes como Java, que tiene la concurrencia totalmente integrada.
- Los métodos se califican con el modificador synchronized.
- Puede existir también synchronized a nivel de bloque.

Métodos sincronizados

- **notify()** despierta un único thread entre los que están esperando en el monitor del objeto.
- **wait()** provoca que el thread invocante espere hasta que otro thread invoque el método **notify()** para este objeto.
- Productor-Consumidor con Monitores en Java