II
Programación Concurrente
Clases 3 a 6: Memoria compartida

Ramiro Martínez D'Elía

2021

Índice general

Capítulo 1

Locks y Barriers

1.1. Introducción

Los programas concurrentes emplean dos tipos básicos de sincronización: exclusión mutua y sincronización por condición. Este capítulo examina 2 (dos) problemas importantes (secciones críticas y barreras).

El problema de la sección crítica se preocupa en implementar acciones atómicas por software. Este problema surge en la mayoría de los programas concurrentes, donde, la exclusión mutua es implementada mediante locks que protegen las secciones críticas.

Una barrera (barrier), es un punto de sincronización al que todos los procesos deben llegar, antes de que cualquier proceso se le permita proceder. Es un problema muy común en los programas paralelos.

1.2. El problema de la sección crítica

En este problema, ***n*** procesos repetidamente ejecutan secciones críticas y no críticas de código. La sección crítica está precedida por un protocolo de entrada y seguida por un protocolo de salida. Los procesos que contengan secciones críticas, deberían ser de la siguiente forma.

```
1  Process SeccionCritica[i = 1 to n]
2    while (true)
3       # Entry protocol
4       # critical section
5       # Exit protocol
6       # Noncritical section
7    end;
8   end;
```

Cada sección crítica, es un conjunto de sentencias que acceden a algún recurso compartido. Mientras que, cada sección no crítica es otra secuencia de instrucciones. Para resolver este problema, es necesario implementar protocolos de entrada y salida que cumplan las siguiente 4 (cuatro) propiedades:

1. **Exclusión mutua:** A lo sumo un proceso podrá estar ejecutando su sección crítica. Esta es una ***propiedad de seguridad***; donde lo malo que puede ocurrir es que 2 (dos), o más, procesos accedan a su sección crítica en el mismo momento.

2. **Ausencia de deadlock:** Si 2 (dos) o más procesos intentan entrar a sus secciones críticas, al menos uno tendrá éxito. Esta es una propiedad ***propiedad de seguridad***; donde lo malo que puede ocurrir es que todos los procesos estén esperando ingresar pero, ninguno sea capaz de lograrlo.

3. **Ausencia de demoras inecesarias:** Si un proceso intenta ingresar a su sección crítica y los demás procesos se encuentran ejecutando sus secciones no críticas o finalizaron, el primer proceso no debe estar impedido de ingresar a su sección crítica. Esta es una ***propiedad de seguridad***, donde lo malo que puede ocurrir es que; un proceso no pueda ingresar a su sección crítica aunque no haya procesos en sus secciones críticas.

4. **Eventual entrada:** Todo procesos que intente ingresar a su sección crítica, eventualmente lo logrará. Esta es una ***propiedad de vida*** y es afectada directamente por la política de scheduling.

Cualquier solución al problema de la sección crítica, también puede ser utilizada para implementar sentencias ***await arbitrarias***.

1.3. Señalizando con barreras

Varios problemas pueden ser resueltos utilizando algoritmos iterativos que sucesivamente computen aproximaciones a la respuesta. Terminando cuando la respuesta final haya sido procesada, o bien, haya convergido.

La idea es utilizar múltiples procesos, para procesar partes disjuntas de una solución en paralelo. La clave principal en la mayoría de los algoritmos paralelos, es que cada iteración depende del resultado de una iteración previa. Así, podemos llegar a la siguiente forma general para todo algoritmo que implemente barreras.

```
1 Process Worker[i = 1 .. n]
2   while (true)
3      # Realiza la tarea i
4      # Espera por los demas procesos
5   end;
6  end;
```

Esto es llamado sincronización por barrera, porque la demora al final de cada iteración representa una barrera a la cual todos los procesos deben llegar, antes de que a cualquier otro se le permita continuar.

1.3.1. Barreras simétricas

Si todos los procesos ejecutan el mismo algoritmo y cada proceso está ejecutando en un procesador distinto. Entonces, todos los procesos deberían llegar a la barrera casi al mismo tiempo.

Esta es la opción más adecuada para programas que ejecuten en máquinas con memoria compartida. Mas adelante, se abordarán algoritmos para este tipo de barreras; el ***butterfly*** y ***dissemination barrier*** más precisamente.

1.3.2. Principio de sincronización por banderas

Algunas soluciones, como ***flags and coordinators***, implementan este principio de sincronización. El cual, se basa en las siguientes premisas:

1. El proceso que espera por un flag de condición, es el único que puede limpiar dicho flag.
2. Un flag no puede ser activado, nuevamente, hasta no ser "limpiado".

Capítulo 2

Soluciones con variables compartidas

Introducción

La sincronización, en esta sección, será implementada mediante la técnica de ***busy waiting***. Donde un proceso evalúa, repetidas veces, una condición hasta que esta se vuelva verdadera.

Ventajas

1. Puede ser implementada utilizando instrucciones, de máquina, disponibles es cualquier procesador moderno.
2. Adecuada si cada proceso se ejecuta en su propio procesador.

Desventajas

1. Ineficiente en arquitecturas monoprocesador.

2.1. Para secciones críticas

2.1.1. Spin locks

Solución de grano fino que utiliza instrucciones atómicas especiales, existentes en la mayoría de los procesadores. Por ejemplo, ***Test and Set (TS)***. Se dice que los procesos dan "vueltas" (spinning) hasta que se libere lock.

```
1  bool lock = false;
2
3  Process CS[i = 1 to n]
4    while (true)
5      while (TS(lock)) skip;    # Entry
6      # Critical Section
7      lock = false;             # Exit
8      # Noncritical Section
9    end;
10 end;
```

Ventajas

1. Cumple con 3 (tres) de los requisitos, para secciones críticas: *garantiza exclusión mutua*, *ausencia de deadlock* y *ausencia de demoras inecesarias*.

Desventajas

1. La *eventual entrada* se garantiza solo con *schedulers fuertemente fairs*. Ya que lock se vuelve verdadera, con infinita frecuencia.
2. No atiende prioridades, es decir; no controla el orden en que los procesos, demorados, entran a su sección crítica.

2.1.2. Implementaciones Fair

Spin locks, no termina siendo del todo adecuada. Sería deseable, contar con algoritmos que:

1. Cumplan las 4 (cuatro) propiedades, de una sección crítica.
2. Solo dependan de *schedulers débilmente fair*.
3. Sean más justos. Es decir, manejen prioridades.

Los algoritmos ***Tie breaker***, ***Ticket*** y ***Bakery*** parecen más adecuados ya que, cumplen con todos los requisitos mencionados anteriormente.

2.1.3. Tie breaker

Este algoritmo asegura la exclusión mutua mediante dos variables, una por proceso, `in1` e `in2`. En caso de que ambas valgan verdadero (empate) emplea una variable adicional, `last`, para determinar cuál fue el último en ingresar a su sección crítica.

```
1  bool in1, in2 = false;
2  int last = 1;

3  Process CS1
4    while (true)
5      last = 1; in1 = true; # Entry
6      while (in2 and last == 1) skip;
7      # Critical Section
8      in1 = false; #Exit
9      # Noncritical Section
10   end;
11 end;
12
```

```
13 Process CS2
14   while (true)
15     last = 2; in2 = true; # Entry
16     while (in1 and last == 2) skip;
17     # Critical Section
18     in2 = false; # Exit
19     # Noncritical section
20   end;
21 end;
```

Ventajas

1. No requiere instrucciones especiales.
2. Prioriza al primer proceso, en iniciar el protocolo de entrada.

Desventajas

1. Dificil generalizarlo a n procesos.

2.1.4. Ticket

El algoritmo ticket, es una solución al problema de la sección crítica generalizada para n procesos, fácil de entender e implementar. El algoritmo se basa en la entrega de tickets (números) a procesos y posteriormente atenderlos en orden de llegada.

Para esto, obligatoriamente, se requiere de alguna instrucción especial que entregue e incremente los números a cada proceso de forma atómica, para evitar duplicados. Esta instrucción puede ser ***Fetch and Add***.

De no existir una instrucción máquina, de estilo ***Fetch and Add***, podemos reemplazarla con otra sección crítica.

```
1  int number = 1;
2  int next = 1;
3  int[] turn[n] = ([n] 0);
```

```
# Con instruccion FA

Process Worker[i = 1..n]
  # Entry protocol
  turn[i] = FA(number, 1);
  while (turns[i] != next) skip;
  # Critical Section
  next = next + 1;        # Exit
  # Noncritical Section
end;
```

```
# Sin instruccion FA

Process Worker[i = 1..n]
  turn[i] = number;
  <number = number + 1>
  while (turns[i] != next) skip;
  # Critical Section
  next = next + 1;
  # Noncritical Section
end;
```

Ventajas

1. Sencillo de implemetar.
2. General para n procesos.

Desventajas

1. La implementación sin instrucciones especiales, puede entregar números repetidos. Esto, decrementa el grado de justicia del algoritmo.

2.1.5. Bakery

Al igual que en el algoritmo ticket, los procesos obtienen un número y esperan a ser atendidos. La diferencia radica en que en el algoritmo bakery, cada proceso debe revisar el número de los demás para obtener uno mayor a todos los que se encuentran demorados.

```
int[] turn[n] = ([n] 0);

Process SC [i = 1 .. n]
  while (true)
    turn[i] = 1;
    turn[i] = max(turn) + 1;
    for (j = 1 to n st j != i)                          # Entry
      while (turn[j] != 0 and turn[i] > turn[j]) skip;
    # Critical Section
    turn[i] = 0;                                        # Exit
    # Noncritical Section
  end;
end;
```

Ventajas

1. No requiere instrucciones especiales.
2. General para n procesos.

Desventajas

1. Resulta complejo, y costoso, calcular el máximo entre n valores.

2.2. Para barreras

2.2.1. Shared counter

La manera más sencilla de especificar una barrera, es la de utilizar un contador compartido iniciado en 0 (cero).

Asumiendo que existan n procesos que necesitan reunirse en un barrera. Cuando un proceso llega a la barrera, incrementa el contador; cuando el contador valga n, todos los procesos podrán continuar.

```
int count = 0;

Process Worker[i = 1 .. n]
  while (true)
    # Realizar tarea ...
    FA(count, 1);
    while (count != n) skip;
  end;
end;
```

Ventajas

1. Solución adecuada para un n pequeño.

Desventajas

1. La variable *count* debe ser reiniciada cuando todos crucen la barrera y, por sobre todo, antes de que cualquier proceso intente incrementarla.
2. Requiere instrucciones especiales.
3. Requiere una política de administración eficiente. La variable *count* es referenciada varias veces, esto puede provocar ***Memory Contention***.

2.2.2. Flags and coordinators

El algoritmo utiliza **dos arreglos como flags**; el primero para marcar la llegada de cada proceso a la barrera. Y el segundo, para indicar a cada proceso que puede continuar. Este último arreglo es actualizado por un proceso coordinador.

```
int[] arrive[n] = ([n] 0);
int[] continue[n] = ([n] 0);

Process Worker[i = 1..n]
  while (true)
    # Realizar tarea
    arrive[i] = 1;
    while (continue[i] == 0) skip;
    continue[i] = 0;
  end;
end;
```

```
Process Coordinator
  while (true)
    for (i = 1 to n)
      while (arrive[i] == 0) skip;
      arrive[i] = 0;
    end;
    for (i = 1 to n)
      continue[i] = 1
  end;
end;
```

Ventajas

1. Resetea correctamente los contadores.
2. Evita ***memory contention***. Ya que, cada elemento del arreglo utiliza una línea de caché distinta.

Desventajas

1. El tiempo de ejecución del coordinador, es proporcional a n. Por su uso, no es recomendable para n muy grandes.

2.2.3. Combined tree barrier

Este algoritmo combina el rol de los workers y el del coordinador, de forma tal que cada worker es, también, un coordinador.

Los procesos son organizados en forma de árbol y se procede con la siguiente lógica: cada nodo worker primero espera a que sus hijos le den la señal de llegada, luego avisa a su padre que él también llegó.

Cuando el nodo raíz, recibe la señal de llegada de sus hijos se sobreentiende que todos los demás workers también lo hicieron. Así, la raíz envía la señal de continuar a sus hijos y así sucesivamente.

```
1  int[n] arrive = 0;
2  int[n] continue = 0;

3  Process Leaf[1..L]
4    # Hacer algo ...
5    arrive[L] = 1;
6    < await (continue[L] == 1) >
7    continue[L] = 0;
8  End.
9
10 Process Root
11   < await(arrive[left] == 1) >
12   arrive[left] = 0;
13   < await(arrive[right] == 1) >
14   arrive[right] = 0;
15   # Hacer algo ...
16   arrive[R] = 1;
17   < continue[left] = 1;
18   continue[right] = 1; >
19 End.
20
21 Process Internal[1..I]
22   < await(arrive[left] == 1) >
23   arrive[left] = 0;
24   < await(arrive[right] == 1) >
25   arrive[right] = 0;
26   # Hacer algo ...
27   arrive[I] = 1;
28   < await(continue[I] == 1) >
29   continue[I] = 0;
30   < continue[left] = 1;
31   continue[right] = 1; >
32 End.
```

Ventajas

1. Útil para n muy grandes ya que, el tiempo de ejecución es proporcional al alto del árbol: $\log_2 n$.
2. Adecuado para máquinas con memoria distribuida.

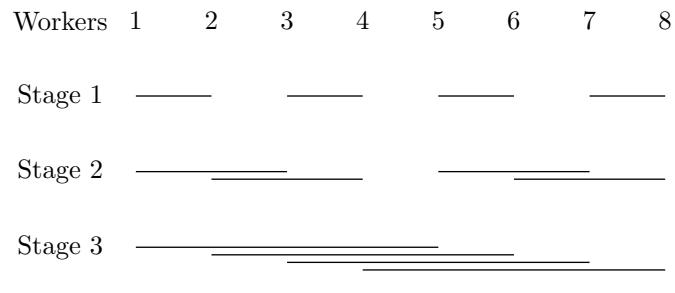
2.2.4. Butterfly

La idea es conectar barreras de pares de procesos, para construir una barrera de n procesos. Asumiendo que *Worker[1:n]* es un arreglo de procesos y que n es potencia de 2, podríamos combinarlos de la siguiente manera.

Por la forma de conexión, es conocida como butterfly barrier. Como se aprecia en la figura, cada proceso se conecta con otro distinto en cada una de sus $\log_2 n$ pasadas. Más precisamente, en cada pasada, cada proceso se sincroniza con otro a una distancia 2^{S-1}.

Cuando un proceso finalizó todas sus pasadas, todos los procesos arribaron a la barrera y pueden proceder. Esto es, porque los procesos están directa o indirectamente sincronizados los unos con los otros.

```
1  int[n] arrive = 0;
2  int stages = log2(n);
3
4  Process Worker[i=1..n]
5    for (s = 1 to stages)
6      arrive[i] = arrive[i] + 1;
7      j = neighbord_for(i, s);
8      while (arrive[j] < arrive[i])
9        skip;
10   end;
11 End.
```



Si n no fuese potencia de 2, podría utilizarse el siguiente n potencia de 2. Generando workers substitutos para cada iteración. Este workaround, decrementa la eficiencia del algoritmo.

2.3. Defectos

La mayoría de los protocolos implementados por ***busy waiting*** son ***complejos*** y la ***separación entre variables***, utilizadas para la sicronización y para cómputo general, es ***poco clara***.

Otro defecto es la ***ineficiencia*** de los protocolos de busy waiting ***en la mayoria de los programas multihilos***. Excepto para el caso de los programas paralelos donde el número de procesos concuerde con el número de procesadores.

No obstante, usualmente, existen más procesos que procesadores y ***resulta menos productivo otorgar CPU a procesos para que hagan spinning en lugar de cómputo***.

La eficiencia es fundamental, en los programas concurrentes, por esto es deseable tener herramientas más eficientes, para el desarrollo de aplicaciones concurrentes, como: ***semáforos*** y ***monitores***.

Con este tipo de herramientas, los procesos demorados no ocupan tiempo CPU hasta que no tengan la posibilidad de ejecutarse. En cuyo caso, serán colocados en una cola de listos.

Capítulo 3

Soluciones con semáforos

Al igual que los semáforos viales sirven para proveer un mecanismo de señalización para prevenir accidentes. En los programas concurrentes, los semáforos sirven para proveer un mecanismo de señalización entre procesos e implementar exclusión mutua y sincronización por condición.

3.1. Sintáxis y semántica

Un semáforo es una variable compartida, que puede pensarse en términos de una instancia de la clase semáforo. Dicha clase, posee solo dos métodos y una variable interna contador.

El método ***v*** es utilizado para señalar la ocurrencia de un evento, y en consecuencia incrementa de forma atómica el contador interno.

El método ***p***, demora al proceso hasta que un evento haya ocurrido y decrementa de forma atómica el contador interno.

Por último, el ***contador*** es una variable que sólo toma valores enteros positivos.

Tipo de semáforo	Valores del contador
Binario	Entre 0 y 1
General	Entre 0 y ∞

```
1  sem s;
2
3  P(s): < await(s > 0) s = s - 1; >
4
5  V(s): < s = s + 1; >
```

3.1.1. Para secciones críticas

El problema de la sección crítica se puede resolver empleando una variable *lock*. La cual, valdrá 1 (*true*) si no hay procesos en su sección crítica ó 0 (*false*) en caso contrario.

Cuando un proceso desea entrar en su sección crítica; primero deberá esperar a que lock valga 1 (*true*) y luego colocar lock en 0 (*false*). Cuando un proceso sale, deberá colocar a lock nuevamente en (*true*).

```
1  sem mutex = 1;
2  Process SC[i = 1 .. n]
3    while (true)
4      P(mutex);     # Entry
5      # Seccion Critica
6      V(mutex);     # Exit
7      # Seccion no critica
8    end;
9  end;
```

3.2. Split Binary Semaphores

La técnica split binary semaphores consiste en combinar 2 (dos), o más, semáforos binarios como si fuesen un solo. Todo conjunto de semáforos, formaran un SBS si cumplen la siguiente regla: $0 \leq s_1 + s_2 + ... + s_n \leq 1$.

3.2.1. Productores y consumidores

Dado un programa donde los procesos se comunican, entre sí, mediante un ***búffer con capacidad para 1 (un) mensaje***. En dicho programa, existiran 2 (dos) clases de procesos: los productores y los consumidores. Los ***productores***; crean mensajes, esperan a que el búffer esté vacío, depositarán su mensaje y marcarán el búffer como lleno. Mientras que, los ***consumidores***; esperan a que el búffer esté lleno, retiran el mensaje y marcan el búffer como vacío.

La forma más sencilla de sincronizar a los procesos es; utilizar semáforos en terminos de los estados posibles del búffer; *empty* y *full*. Juntos, *empty* y *full*, conforman un split binary semaphore que; proveerá exclusión mutua sobre el acceso del búffer.

```
1  sem empty = 1, full = 0;
2  any buffer;
```

```
3  Process Consumer[1..n]
4    while (true)
5      P(full);
6      message = buffer;
7      V(empty);
8    end;
9  End.
10
```

```
11 Process Producer[1..n]
12   while (true)
13     P(empty);
14     buffer = message;
15     V(full);
16   end;
17 End.
```

3.3. Contadores de recursos

Usualmente los procesos compiten por el acceso a recursos limitados. En esos casos, ***semáforos generales***; pueden ser utilizados como contadores de recursos disponibles.

3.3.1. Buffers limitados

Dado un programa donde los procesos se comunican, entre sí, mediante un ***búffer con capacidad para n mensaje***. En dicho programa, existiran 2 (dos) clases de procesos: los productores y los consumidores. Los ***productores***; crean mensajes, esperan a que el búffer esté vacío, depositarán su mensaje y marcarán el búffer como lleno. Mientras que, los ***consumidores***; esperan a que el búffer esté lleno, retiran el mensaje y marcan el búffer como vacío.

En este caso, el recurso son los espacios libres del buffer. Como adición, se debe aplicar exclusión mutua para que distintos consumidores no recuperen el mismo mensaje (mantener consitente *front*) y para, que los productores no sobreescriban mensajes (mantener consistente *near*).

```
1  int front, rear = 0;
2  sem empty = n, full = 0;
3  sem mutexFetch = 1, mutexDeposit = 1;
4  array[n] buffer;
```

```
Process Producer[1..n]
  while (true)
    P(empty);
    P(mutexDeposit);
    buffer[rear] = data;
    rear = (rear + 1) % n;
    V(mutexDeposit);
    V(full);
  end;
End.
```

```
Process Consumer[1..n]
  while (true)
    P(full);
    P(mutexFetch)
    data = buffer[front];
    front = (front + 1) % n;
    V(mutexFetch)
  end;
End.
```

3.4. Exclusión mutua selectiva

La exlusión mutua selectiva, se presenta cuando cada proceso compite contra un subconjunto de procesos (por un recurso). En lugar de, competir contra todos.

3.4.1. Dining Philosophers

Por ejemplo: cinco filósofos se sientan a comer en una mesa redonda donde hay solo cinco tenedores. Cada filósofo, para comer, requiere de dos tenedores. Esto implica que; dos filósofos vecinos no pueden comer al mismo tiempo y que, a lo sumo, solo dos filósofos podrán comer al mismo tiempo.

El problema de exclusión mutua selectiva se da entre cada par de filósofos y un tenedor. Tener en cuenta, de que si fuesen 3 filósofos, no sería un problema de exclusión mutua.

```
sem forks[5] = {0,0,0,0,0}

Process Philosopher[i = 0 to 3]
  while(true)
    p(fork[i]);
    p(fork[i+1]);
    # come ...
    v(fork[i]);
    v(fork[i+1]);
  end;
end;
```

```

Process Philosopher[4]
  while(true)
    p(fork[0]);
    p(fork[4]);
    # come ...
    v(fork[0]);
    v(fork[4]);
  end;
end;
```

💎 Si en lugar de 5 filósofos fueran 3, ¿el problema seguiría siendo de exclusión mutua selectiva? ¿Por qué?
Si hay 3 filósofos, por consiguiente hay 2 tenedores. Lo que significa que todos los procesos son adyacentes, por lo cual la competencia se da entre todos y dejaría de ser un problema de exclusión mutua selectiva.

💎 El problema de los filósofos resuelto de forma centralizada y sin posiciones fijas ¿es de exclusión mutua selectiva? ¿Por qué?
Si los filósofos pueden utilizar cualquier cubierto (sin posiciones fijas) y, en caso de haber disponibilidad, son servidos por un mozo (coordinador). No estamos hablando de exclusión mutua selectiva ya que, los filosofos compiten contra todos no solo contra sus adyacentes.

3.4.2. Lectores y escritores

Otro ejemplo, de exclusión mutua selectiva, donde clases de procesos, compiten por el acceso a un recurso es el siguiente. Procesos escritores, que requieren acceso exclusivo para evitar interferencias. Y procesos, lectores, los cuales pueden acceder de forma concurrente entre sí (siempre y cuando no haya escritores haciendo uso de la base de datos).

El siguiente algoritmo, resuelve el problema. No obstante, esta solución no es fair. Ya que, prioriza lectores por sobre escritores.

```
1  int nr = 0; # lectores activos
2  sem rw = 1; # acceso a bbdd
3  sem mutexR = 1; # acceso a nr

4  Process Reader
5    while (true)
6      P(mutexR)
7      nr = nr + 1;
8      if (nr == 1) P(rw);
9      V(mutexR);
10     # lee en bbdd
11     P(mutexR)
12     nr = nr - 1;
13     if (nr == 0) V(rw);
14     V(mutexR);
15   end;
16 end;
```

```
17 Process Writer
18   while (true)
19     P(rw);
20     # escribe en bbdd
21     V(rw);
22   end;
23 end;
```

💎 Si solo se acepta sólo 1 escritor o 1 lector en la BD, ¿tenemos un problema de exclusión mutua selectiva? ¿Por qué?
En este caso, sería solo un problema de exclusión mutua. Debido a que, por definición, la competencia es entre todos los procesos (lectores vs escritores, lectores vs lectores, escritores vs escritores).

3.5. Passing the Baton

Técnica que utiliza ***split binary semaphores*** para proveer exclusión mutua y despertar procesos demorados (incluso respetando su orden). Empleando esta técnica, podremos ***especificar sentencias await arbitrarias***. Su implementación, respeta la siguiente forma:

1. Un semáforo $\boldsymbol{e}$. Inicialmente en 1 para, controlar los accesos a la sección crítica.
2. Un semáforo b_j para demorar procesos hasta que, su guarda, B_j sea verdadera.
3. Un contador d_j para contar los procesos demorados por b_j.

Cuando un proceso se encuentra en su sección crítica, retiene el permiso de ejecución (***baton***). Al finalizar, le pasa el permiso a otro proceso (si lo hubiera) o bien, lo libera.

3.5.1. Lectores y escritores

Resolveremos el mismo problema, de lectores y escritores, de la sección anterior introduciendo la técnica ***passing the baton***. Donde:

- $\boldsymbol{e} \rightarrow e$
- $b_j \rightarrow r$ y w
- $d_j \rightarrow dr$ y dw

Si bien este algoritmo, sigue priorizando a los lectores; podremos modificar a ***SIGNAL*** para darle la política que quisieramos.

```
int nr = 0,  # Lectores activos
int nw = 0;  # Escritores activos

int dr = 0, # Lectores demorados
int dw = 0; # Escritores demorados

# SBS: 0<=(e+r+w)<=1
sem e = 1,  # Baton
sem r = 0,  # Demora lectores
sem w = 0;  # Demora escritores
```

```
SIGNAL:
  if (nw == 0 & nr > 0)
    dr--;
    V(r);
  elseif (nr == 0 & nw == 0 & dw > 0)
    dw--
    V(w);
  else
    V(e);
```

```
Process Writer[1..n]
  while (true)
    P(e);
    if (nr > 0 or nw > 0)
      dw++;
      V(e);
      P(w);
    end;
    nw++;
    SIGNAL
    # Escribir en la bbbd ...
    P(e);
    nw--;
    SIGNAL
  end;
End.

```

```
Process Reader[1..n]
  while (true)
    P(e);
    if (nw > 0)
      dr++;
      V(e);
      P(r);
    end;
    nr++;
    SIGNAL
    # Leer en la bbbd ...
    P(e);
    nr--;
    SIGNAL
  end;
End.
```

💎 ¿Qué relación encuentra con la técnica de passing the condition?
Ambas técnicas poseen la misma finalidad. Solo que, Passing the condition es utilizada en Monitores.

3.5.2. Alocación de recursos - SJN

En los problemas de alocación de recursos, se deben implementar políticas de acceso generales a un recurso compartido por los procesos.

Dichas políticas deben contemplar: cuándo se le puede dar acceso a un proceso y cuál proceso toma el recurso si hubiese más de uno esperando.

Como su nombre lo indica, una política de ***Short Job Next*** le otorga el recurso al proceso en espera con menor valor de tiempo (el más corto). Su semántica, es la siguiente:

- `request(time, id)`: Un proceso solicita el acceso. Si el recurso está libre, se le otorga. Caso contrario, se coloca al proceso en una lista de espera.
- `release`: Un proceso libera el recurso. Si hay procesos en espera se otorga, el recurso, al proceso con valor de tiempo más chico. Caso contrario, el recurso queda libre.

Este problema puede ser implementado, utilizando la técnica Passing the baton de la siguiente manera:

```
request(time, id):
  P(e);   # mutex sobre free
  if (not free) DELAY;
  free = false;
  SIGNAL;

release():
  P(e);   # mutex sobre free
  libre = true;
  SIGNAL;
```

En **DELAY** un proceso:

- Se coloca en una lista de espera.
- Libera SC ejecutando `V(e)`.
- Demora en un semáforo `b[id]`.

En **SIGNAL** un proceso:

- Cuando se libera el recurso, se le asigna al próximo proceso en espera si lo hubiera.

Esta política minimiza el tiempo promedio de ejecución pero, **no es una solución fair**. Procesos con un tiempo mayor, pueden quedar relegados ante la aparición de procesos con tiempo menor.

Dicha desventaja se puede evitar, implementando políticas de **aging**. Es decir controlando, también, el tiempo en el que un proceso estuvo en lista de espera.

3.6. Instrucciones máquina utilizadas

3.6.1. Fetch and Add

De manera atómica incrementa *number* en *inc* veces y retorna su antiguo valor.

```
1 FA (int number, int inc)
2   < int temp = number;
3   number = number + inc;
4   return temp; >
5 end;
```

3.6.2. Test and Set

De manera atómica setea *lock* en true y retorna su antiguo valor.

```
1 bool TS(bool lock)
2   < bool initial = lock;
3   lock = true;
4   return initial; >
5 end;
```

3.7. Desventajas

Los semáforos resultan un mecanismo de sincronización fundamental. Con ellos, podemos resolver cualquier problema de sincronización.

No obstante, por naturaleza, son un mecanismo de bajo nivel. Un algoritmo que emplea semáforos posee las siguientes características que, pueden llevar a que el programador cometa errores.

- Variables compartidas globales a los procesos.
- Protocolos de entrada a secciones críticas, dispersas en el código.
- Al agregar procesos, se debe verificar el acceso correcto a las variables compartidas.
- La implementación de exclusión mutua y sincronización por condicion son similares, aunque los conceptos sean distintos.

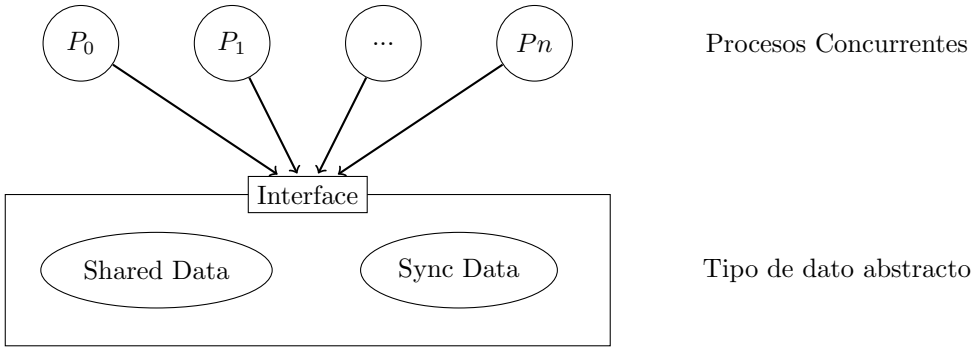
Capítulo 4

Soluciones con Monitores

4.1. Introducción

Los monitores son módulos, del programa, que ofrecen una estructura que permite desarrollar programas concurrentes más limpios con respecto al uso de semáforos.

Principalmente, los monitores, funcionan como una capa de abstracción de datos. Un monitor encapsula la representación de un objeto abstracto (o recurso) y ofrece un conjunto de operaciones para manipularlo.



4.2. Sintáxis y Semántica

Un monitor es utilizado para agrupar la representación e implementación de un recurso compartido (class). La especificación de un monitor consta de las siguientes partes:

- **Interface**: Especifica las operaciones provistas por el recurso.
- **Cuerpo**: Contiene las variables que representan al recurso y procedimientos que implemetan las operaciones de la interfaz.

```
1 Monitor Example
2
3 # Declracion de variables permanentes.
4 # Inicializacion de variables.
5
6 procedure op_test(formals)
7   # Body ...
8 end;
9 End.
```

Las ***variables permanentes*** son compartidas por todos los procedimientos dentro del monitor. Son llamadas permanentes, porque existen y consevan sus valores durante todo el ciclo de vida de un monitor. Ellas son inicializadas antes de cualquier llamado a procedimientos.

Los ***procedimientos***, tienen acceso a las variables permanentes, parámetros formales y sus variables locales. Estas últimas, son copiadas e inicializadas en cada llamado al procedimiento.

La principal ventaja, del uso de monitores, es que pueden ser desarrollados de manera aislada al programa que los vaya a utilizar.

4.3. Sincronización

En monitores, a lo sumo **un procedimiento** puede estar **activo a la vez**. Un procedimiento se considera activo, cuando un proceso externo se encuentra ejecutando alguna de sus sentencias.

De esta forma, la ***exclusión mutua*** se garantiza de forma **implícita**. Esto hace los algoritmos más fáciles de leer ya que, no hay protocolos de entrada y salida explícitos.

En contraste, la ***sincronización por condición*** debe ser especificada de forma **explícita**. Esto se debe a que, las condiciones son diferentes en cada programa.

4.3.1. Variables condición

Una variable condición es utilizada para demorar un proceso que no puede continuar ejecutandose, de forma segura, hasta que algún estado del minitor satisfaga una condición.

También son usadas para despertar procesos demorados, cuando la condición se vuelva verdadera.

Este tipo de variables, solo pueden ser utilizadas dentro de un monitor. Y su declaración, sigue la siguiente forma:

```
1 cond vc;
```

Internamente el valor de `vc` es el de una cola, inicialmente vacía. El programador no manipula directamente la cola, sino que lo hace mediante operaciones especiales.

- `empty`: Retorna `true` si la cola está vacía. Retornará `false`, en caso contrario.
- `wait`: Causa la demora del proceso en ejecución y lo agrega a la cola.
- `signal`: Despierta procesos demorados y los retira de la cola.

Si bien, es posible definir una política de prioridades a la cola de procesos demorados; el comportamiento por defecto es el de una política FIFO. De esta forma: `wait` colocará procesos al final de la cola y `signal` desperatá al proceso que este en el frente.

4.4. Disciplinas de señalización

Cuando un proceso ejecuta `signal`, se encuentra dentro de un monitor y por consiguiente tiene el control del lock implícito asociado al monitor.

Dicha situación plantea un dilema. Si `signal` despierta a otro proceso, ahora tendremos 2 (dos) procesos ejecutandose dentro del monitor y por definición, solo un proceso debe tener acceso exclusivo al monitor.

Ante la situación planteada, podemos adoptar alguna de las siguientes políticas:

1. ***Signal and Continue (SC)***: El proceso *señalizador* continua su ejecución. El proceso *señalizado*, es colocado en la cola de listos y espera a que se libere el monitor para continuar.
2. ***Signal and Wait (SW)***: El proceso *señalizado* continua su ejecución. El proceso *señalizador*, espera a que se libere el monitor para continuar.

4.5. Sincronización básica por condición

4.5.1. Buffers limitados

- `not_full` → señaliza `count < n`
- `empty` → señaliza `count > 0`

```
1  monitor BoundedBuffer
2
3  typeT buffer[n];
4  int front = 0, rear = 0, count = 0;
5  cond not_full, not_empty;

6  procedure deposit(typeT data)
7    while(count == n) wait(not_full);
8    buffer[rear] = data;
9    rear = (rear + 1) % 10;
10   count++;
11   signal(not_empty)
12 end;
```

```
13 procedure fetch(typeT &result)
14   while(count == 0) wait(not_empty);
15   result = buffer[front];
16   front = (front + 1) % n;
17   count--;
18   signal(not_full);
19 end;
```

4.6. Broadcast

4.6.1. Lectores y Escritores

- `oktoread` → señaliza `nw == 0`
- `oktowrite` → señaliza `nw == 0 && nr == 0`
- `signal_all` → despierta a todos (broadcast)

```
1  monitor RW_Controller
2
3  int nr = 0, nw = 0;
4  cond oktoread, oktowrite;

5  procedure request_read()
6    while (nw > 0) wait(oktoread);
7    nr = nr + 1;
8  end;
9
10 procedure release_read()
11   nr = nr - 1;
12   if (nr == 0) signal(oktowrite);
13 end;
```

```
14 procedure request_write()
15   while (nr > 0 || nw > 0)
16     wait(oktowrite);
17   nw = nw + 1;
18 end;
19
20 procedure release_write()
21   nw = nw - 1;
22   signal(oktowrite);
23   signal_all(oktoread);
24 end;
```

4.7. Priority Wait

4.7.1. Shortest Job Next Allocation (SJN)

- Se maneja la cola con una politica SJN, en lugar de FIFO.

```
1  Monitor SJN
2
3  bool free = true,
4  cond turn;

5  procedure request(int time)
6    if (free)
7      free = false;
8    else
9      wait(turn, time)
10 end;

11 procedure release()}
12   if (empty(turn))
13     free = true;
14   else
15     signal(turn);
16 end;
```