Sincronización
6

Sistemas operativos y distribuidos

Gustavo Distel
gd@cs.uns.edu.ar

DCIC - UNS

Sincronización

Contenido

- Generalidades.
- El problema de la sección crítica.
- La solución de Peterson.
- Soporte de *hardware* para sincronización.
- *Mutex locks*.
- Semáforos.
- Monitores.
- *Liveness*.

Sincronización

Generalidades

- Los procesos pueden ejecutarse concurrentemente o en paralelo, y a su vez ser cooperativos.
  - Un proceso podría completar parcialmente su ejecución antes de que se planifique otro.
  - Esto acarrea problemas en la integridad de los datos compartidos entre los procesos.
- Se elimina en el problema del **`productor-consumidor`** la restricción de **`BUFFER SIZE - 1`** item en el búfer al mismo tiempo
  - Se agrega la variable entera ***`count`*** inicializada en **`0`**.

<u>Productor</u>

```
while (true) {
    /* produce an item in next_produced */
    while (count == BUFFER SIZE)
      ; /* do nothing */

    buffer[in] = next produced;
    in = (in + 1) % BUFFER SIZE;
    count++;
}
```

<u>Consumidor</u>

```
while (true) {
    while (count == 0)
      ; /* do nothing */
    next consumed = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    count--;
/* consume the item in next_consumed */
}
```

Sincronización

Generalidades

- Aunque el productor y el consumidor definidos funcionan correctamente por separado, podrían no hacerlo al ejecutarse de manera concurrente.
- Supongamos que **`count = 5`** y se ejecuta concurrentemente **`count++`** y **`count--`**.
  - El valor correcto es **5**, pero podría ser "**4, 5** ó **6**", dado que en lenguaje máquina las sentencias posibles son:

count++	count--
$register_1 = count$	$register_2 = count$
$register_1 = register_1 + 1$	$register_2 = register_2 - 1$
$count = register_1$	$count = register_2$

- Donde **$register_1$** y **$register_2$** son registros locales de la *CPU*.

Sincronización

Generalidades

- La ejecución concurrente de `"count++"` y `"count--"` es equivalente a una ejecución secuencial en la que las sentencias de bajo nivel se entrelazan en algún orden arbitrario (el orden de las declaraciones de alto nivel se mantiene).
- Uno de las ejecuciones entrelazadas podría ser:

T_0 : producer execute $register_1$ = count		{$register_1$ = 5}
T_1 : producer execute $register_1$ = $register_1$ + 1		{$register_1$ = 6}
T_2 : consumer execute $register_2$ = count		{$register_2$ = 5}
T_3 : consumer execute $register_2$ = $register_2$ - 1		{$register_2$ = 4}
T_4 : producer execute count = $register_1$		{count = 6}
T_5 : consumer execute count = $register_2$		{count = 4}

- Llegamos al estado incorrecto `count == 4`.
- Y si cambiamos el orden de T_4 y T_5 llegamos al estado incorrecto `count == 6`.

Sincronización

Generalidades

- Se llega al estado incorrecto porque permitimos que ambos procesos manipulen la variable `count` concurrentemente.
- Condición de carrera (*race condition*): se produce cuando varios procesos acceden y manipulan los mismos datos concurrentemente y el resultado de la ejecución depende del orden particular de acceso.
- Para protegernos de la condición de carrera en el problema del `productor-consumidor`, debemos asegurarnos de que solo un proceso a la vez pueda manipular la variable `count`.
- Para lograr esto se requiere que los procesos se sincronicen de alguna manera.

El problema de la sección crítica

- Considerando un sistema que consta de ***n*** procesos $\{ P_0 , P_1 , \ldots , P_{n-1} \}$.
- Cada proceso tiene un segmento de código, denominado sección crítica, al cual accede y actualiza los datos que comparte con al menos otro proceso.
- Cuando un proceso se ejecuta en su sección crítica, ningún otro proceso puede ejecutarse en dicha sección.
- El problema de la sección crítica consiste en diseñar un protocolo de sincronización de la actividad de los procesos, de modo que se compartan los datos de manera cooperativa.
- Cada proceso debe solicitar permiso para ingresar a su sección crítica.
- La sección de código que implementa esta solicitud es la sección de entrada.
- La sección crítica puede estar seguida de una sección de salida.
- El código restante es la sección restante.

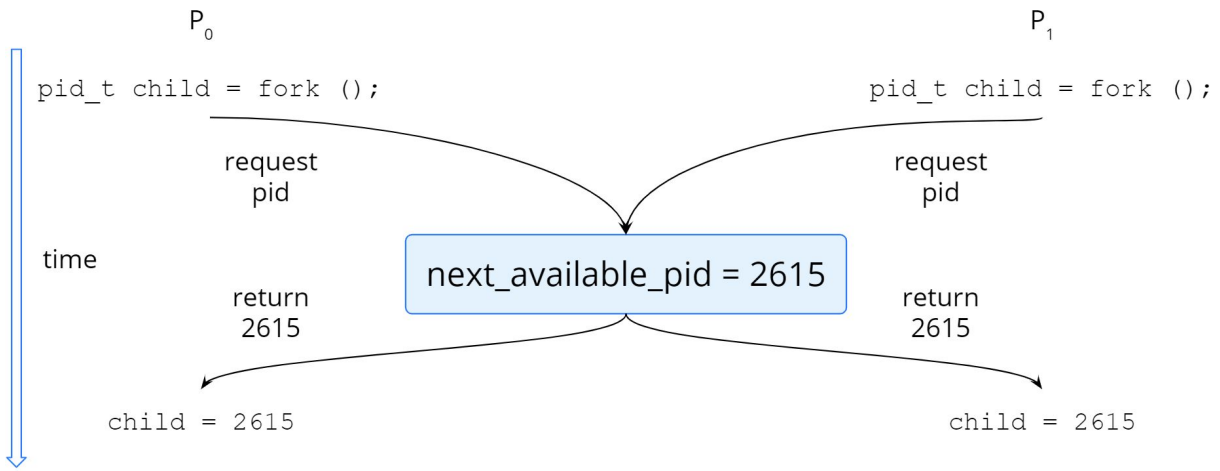
```
while (true) {
        entry section
        critical section
        exit section
        remainder section
}
```

El problema de la sección crítica

- Una solución al problema de la sección crítica debe satisfacer los siguientes tres requisitos:
  - **1. Exclusión mutua.** si el proceso P_i está ejecutándose en su sección crítica, los demás procesos no pueden estar ejecutando sus secciones críticas.
  - **2. Progreso.** si ningún proceso está ejecutando su sección crítica y algunos procesos desean ingresar a sus secciones críticas, sólo aquellos procesos que no estén ejecutando sus secciones restantes pueden participar en la decisión de cuál será el siguiente en entrar a su sección crítica, y esta selección no puede posponerse indefinidamente.
  - **3. Espera limitada.** existe un límite en el número de veces que se permite que otros procesos entren en sus secciones críticas después de que un proceso haya hecho una solicitud para ingresar en su sección crítica y antes de que la misma haya sido concedida.

El problema de la sección crítica

- En un momento dado, varios procesos pueden estar activos en modo *kernel*. Como resultado, el código *kernel* está sujeto a condiciones de carrera.
- Considerando dos procesos, P_0 y P_1, los cuales utilizan la llamada al sistema `fork()`.



- Se da una condición de carrera en la variable del *kernel* `next_available_pid` la cual representa el valor del siguiente *PID* disponible.
- A menos que se brinde exclusión mutua, es posible que se asigne el mismo *PID* a dos procesos.

El problema de la sección crítica

- Existen dos enfoques para el manejo de secciones críticas en los SOs: *kernel* apropiativos y *kernel* no apropiativos.
- Un *kernel* apropiativo permite que un proceso sea apropiado mientras está ejecutando en modo *kernel.*
  - No están libres de condiciones de carrera.
- Un *kernel* no apropiativo no permite que un proceso que se ejecuta en modo *kernel* sea apropiado; un proceso en modo *kernel* se ejecutará hasta que salga de modo *kernel*, se bloquee o ceda voluntariamente la *CPU*.
  - Libre de condiciones de carrera en estructuras de datos del *kernel.*
- Es preferible un *kernel* apropiativo dado que puede responder mejor, ya que hay menos riesgo de que un proceso en modo *kernel* se ejecute durante un período largo antes de ceder el procesador.
- Un *kernel* apropiativo es más adecuado en un entorno en tiempo real, ya que permitirá que un proceso en tiempo real apropie a un proceso ejecutándose en modo *kernel*.

La solución de Peterson

- Es una solución clásica basada en *software* para el problema de la sección crítica.
- La misma está restringida a dos procesos (P_0 y P_1 o P_i y P_j) que alternan la ejecución entre sus secciones críticas y secciones restantes.

Datos compartidos

```
int turn;

boolean flag[2];
```

P_i

```
while (true) {
     flag[i] = true;
     turn = j;
     while (flag[j] && turn == j)
          ;
          /* critical section */
     flag[i] = false;
          /*remainder section */
}
```

P_j

```
while (true) {
     flag[j] = true;
     turn = i;
     while (flag[i] && turn == i)
          ;
          /* critical section */
     flag[j] = false;
          /*remainder section */
}
```

- Se puede demostrar que esta solución es correcta, demostrando que, se preserva la exclusión mutua y se cumple el requisito de progreso y de espera limitada.

La solución de Peterson

- No se garantiza que la solución de Peterson funcione en arquitecturas de computadoras modernas, dado que para mejorar el rendimiento los procesadores y/o compiladores pueden reordenar operaciones de lectura y escritura que no tienen dependencias.
- Para una aplicación de un único hilo, este reordenamiento es irrelevante en lo que respecta a la corrección del programa, ya que los valores finales son consistentes con lo esperado.
- Pero para una aplicación multi-hilada con datos compartidos, el reordenamiento de las instrucciones puede volverse inconsistente o producir resultados inesperados.
- Ejemplo:

<u>Datos compartidos</u>

```
boolean flag = false;

int x = 0;
```

$Thread_1$

```
while (!flag)
;
print x;
```

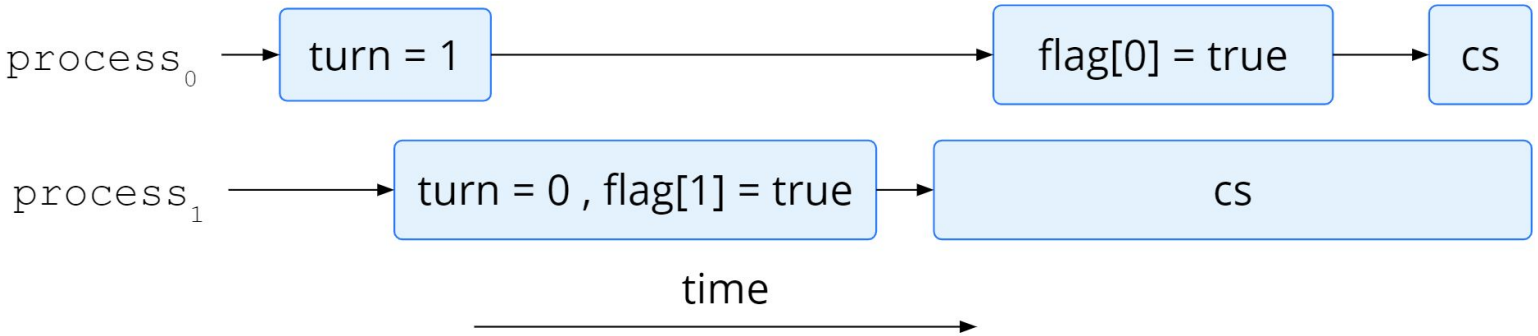
$Thread_2$

```
x = 100;
flag = true;
```

- Las variables `flag` y `x` podrian ser re-ordenadas → $Thread_1$ print `0`.

La solución de Peterson

- Los efectos del reordenamiento de instrucciones en la solución de Peterson:



- Esto permitiría que ambos procesos estén es su sección crítica al mismo tiempo.

Soporte de *hardware* para sincronización

Memory Barriers

- Consisten en instrucciones que fuerzan la propagación de cualquier cambio en memoria al resto de los procesadores, asegurando así que las modificaciones sean visibles para los hilos que se ejecutan en otros procesadores.
- Cuando se ejecuta una instrucción *memory barrier* el sistema garantiza que todos los *loads* y *stores* sean completados antes de que se realicen las siguientes operaciones de *loads* y *stores* (incluso en el reordenamiento de instrucciones).

$Thread_1$	$Thread_2$

`while (!flag)`
`  memory_barrier();`
`print x;`
	`x = 100;`
	`memory_barrier();`
	`flag = true;`

- Estas se consideran operaciones de muy bajo nivel y, por lo general, solo las usan los desarrolladores del *kernel* cuando escriben código especializado que garantice la exclusión mutua.

Soporte de *hardware* para sincronización

Instrucciones de *hardware*

- Muchos sistemas proporcionan instrucciones especiales de *HW* que permiten testear y modificar el contenido de una palabra o intercambiar atómicamente el contenido de dos palabras, es decir, como una unidad ininterrumpida.
- Esto permite resolver el problema de la sección crítica de una manera relativamente simple.

Instrucción `test_and_set()`

```
boolean test_and_set(boolean *target)
{
    boolean rv = *target;
    *target = true;

    return rv;
}
```

Exclusión mutua con `test_and_set()`

- lock inicializada en false.

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */
        /* critical section */
    lock = false;
        /* remainder section */
} while (true);
```

- `test_and_set()` se ejecuta atómicamente, esto es: si dos instrucciones **`test_and_set()`** se ejecutan simultáneamente (cada una en un *core* diferente), se ejecutarán secuencialmente en un orden arbitrario.

Soporte de *hardware* para sincronización

Instrucciones de *hardware*

- `compare_and_swap()`: opera atómicamente sobre dos palabras intercambiando el contenido.

Instrucción `compare_and_swap()`

```
int compare_and_swap(int *value, int
             expected, int new
value){

    int temp = *value;

    if (*value == expected)
        *value = new value;

    return temp;
}
```

Exclusión mutua con `compare_and_swap()`

P_i

- lock inicializada en `0`.

```
while (true) {

  while (compare_and_swap(&lock, 0, 1) != 0)
  ; /* do nothing */

  /* critical section */

lock = 0;

  /* remainder section */
}
```

- Este algoritmo satisface el requisito de exclusión mutua, aunque no satisface espera limitada.

Soporte de *hardware* para sincronización

Instrucciones de *hardware*

- Algoritmo que satisface todos los requisitos de la sección crítica usando **`compare_and_swap()`**:

Estructura de datos común

```
boolean waiting[n];
int lock;
```

- Los elementos del arreglo se inicializan en **`false`**.
- **`lock`** inicializado en **`0`**.

```
while (true) {
     waiting[i] = true;
     key = 1;
     while (waiting[i] && key == 1)
          key = compare_and_swap(&lock,0,1);
     waiting[i] = false;

          /* critical section */

     j = (i + 1) % n;
     while ((j != i) && !waiting[j])
          j = (j + 1) % n;

     if (j == i)
          lock = 0;
     else
          waiting[j] = false;

          /* remainder section */
}
```

Soporte de *hardware* para sincronización

Variables atómicas

- Proporcionan operaciones atómicas en tipos de datos básicos como enteros.
- Se usan para garantizar exclusión mutua en situaciones en las que puede haber una condición de carrera de datos en una variable mientras se actualiza, como al incrementar un contador.
- La mayoría de los sistemas que admiten variables atómicas proporcionan tipos especiales de datos atómicos, así como funciones para acceder y manipular dichas variables.
- Estas funciones a menudo se implementan mediante operaciones como `compare_and_swap()`.

- Aunque proporcionan actualizaciones atómicas, no resuelven por completo las condiciones de carrera en todas las circunstancias.
- Se usan en SOs así como en aplicaciones concurrentes, aunque su uso a menudo se limita a actualizaciones únicas de datos compartidos, como contadores y generadores de secuencias.

```
increment(&sequence)

void increment(atomic_int *v){
    int temp;
    do {
        temp = *v;
    }
    while(temp!=compare_and_swap
                (v,temp,temp+1));
}
```

Mutex Locks

- Las soluciones basadas en *hardware* son complicadas y generalmente inaccesibles para los programadores.
- Los diseñadores de SO crean herramientas de *software* de alto nivel para resolver el problema de la sección crítica, tales como los ***mutex lock***.
- Un proceso debe adquirir (**`acquire()`**) el *lock* antes de ingresar a una sección crítica y liberar (**`release()`**) el *lock* cuando sale de ella.
- Un *mutex lock* tiene una variable *booleana* ***available*** cuyo valor indica si el *lock* está disponible o no.
- Si está disponible, una llamada a **`acquire()`** resulta exitosa y el *lock* pasa a no disponible.
- Un proceso que intenta adquirir un *lock* no disponible se bloquea hasta que sea liberado.

```
while (true) {
        acquire lock
                critical section
        release lock
                remainder section
  }
```

Mutex Locks

```
acquire(){
    while (!available)
        ; /* busy wait */
    available = false;
}
```

```
release(){
    available = true;
}
```

- Las llamadas deben realizarse atómicamente y se pueden implementar utilizando las operaciones de *HW* descritas anteriormente.
- Una desventaja es el *busy waiting*: mientras un proceso está en su sección crítica, cualquier otro proceso que intente ingresar a su sección crítica debe ciclar en la llamada **`acquire()`**.
- Este bucle continuo es claramente un problema en un sistema multiprogramado, donde un solo *core* de *CPU* se comparte entre muchos procesos así como también se desperdician ciclos de la *CPU*.

Mutex Locks

- Este tipo de *mutex lock* también se denomina *spinlock* porque el proceso "*gira*" (spin) mientras espera que el *lock* esté disponible.
- Sin embargo, los *spinlock* tienen una ventaja ya que no se requiere un cambio de contexto cuando un proceso espera en un *lock*, el cual podría demorar un tiempo considerable.
- En ciertas circunstancias en sistemas *multicore* los *spinlocks* son la opción preferible para los *locks*.
- Si se va a mantener un *lock* por un período corto, un hilo puede "*girar*" en un *core* mientras otro hilo está en su sección crítica en otro *core*.
- En los sistemas informáticos *multicore* modernos, los *spinlocks* se utilizan en muchos SOs.

Semáforos

- Un semáforo `S` es una variable entera a la que, además de la inicialización, solo se accede a través de dos operaciones atómicas estándar: `wait()` y `signal()`.

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

```
signal(S) {
    S++;
}
```

- Todas las modificaciones al valor entero del semáforo en las operaciones `wait()` y `signal()` deben ejecutarse atómicamente.
- Es decir, cuando un proceso modifica el valor de un semáforo, ningún otro proceso puede modificar simultáneamente ese mismo valor.
- Además, en el caso de `wait(S)`, la condición de `S (S ≤ 0)`, así como su posible modificación `(S--)`, debe ejecutarse sin interrupción.

Semáforos

Usos

- Los SOs distinguen entre dos tipos de semáforos:
  - Semáforo contador: el valor puede variar en un dominio sin restricciones.
  - Semáforo binario: el valor varía entre `0` y `1`.
    - Se comportan de manera similar a los *mutex locks*.
    - Los sistemas que no proporcionan *mutex locks* pueden utilizar en su lugar semáforos binarios para brindar exclusión mutua.

- Los semáforos contadores se utilizan para controlar el acceso a un recurso que consiste en un número finito de instancias.
  - Inicializar en cantidad de recursos.
  - Proceso que necesita un recurso → `wait()`.
  - Proceso que libera un recurso → `signal()`.

Semáforos

Usos

- Los semáforos se pueden utilizar para resolver varios problemas de sincronización.
- Ejemplo: considerar dos procesos que se ejecutan simultáneamente: P_1 con una declaración D_1 y P_2 con una declaración D_2.
  - Supongamos que requerimos que D_2 se ejecute solo después de que D_1 se ejecute.
  - Podemos implementar este esquema dejando que P_1 y P_2 compartan un semáforo común `synch` inicializado en `0`.

P_1	P_2
D_1; `signal(synch);`	`wait(synch);` D_2;

Semáforos

Ejemplos

- Considerar dos procesos P_1 y P_2 que se ejecutan concurrentemente:

P_1	P_2

`printf("H");`
`printf("L");`
`printf("S");`
	`printf("O");`
	`printf("A");`

- Se solicita agregar semáforos para que la salida siempre sea `HOLAS` o `HOALS` y nunca otra secuencia. Especificar cómo se inicializan los semáforos.
- Solución:

```
Inicialización
    S1=1,
    S2=0,
    S3=0,
    S4=0;
```

P_1	P_2

`wait(S1);`
`printf("H");`
`signal(S2);`
`wait(S3);`
`printf("L");`
`wait(S4);`
`printf("S");`
	`wait(S2);`
	`printf("O");`
	`signal(S3);`
	`printf("A");`
	`signal(S4;);`

Semáforos

Ejemplos

- Considerar tres procesos P_1, P_2 y P_3 que se ejecutan concurrentemente:

P_1	P_2	P_3
`printf("A");` `printf("1");`	`printf("B");` `printf("2");`	`printf("C");` `printf("3");`

- Se solicita agregar semáforos de tal manera que las letras siempre se impriman primero y en orden y, que los números siempre se impriman después de todas las letras y en cualquier orden, por ejemplo:
  - `A B C 1 2 3: bien.`
  - `A B C 3 1 2: bien.`
  - `A C B 1 2 3: mal.`
  - `A B 1 C 2 3: mal.`

Semáforos

Ejemplos

- Solución:

Inicialización	P_1	P_2	P_3

`S1=1,`
`S2=0,`
`S3=0,`
`S4=0;`
	`wait(S1);`
	`printf("A");`
	`signal(S2);`
	`wait(S4);`
	`printf("1");`
		`wait(S2);`
		`printf("B");`
		`signal(S3);`
		`wait(S4)`
		`printf("2");`
			`wait(S3);`
			`printf("C");`
			`signal(S4);`
			`signal(S4);`
			`printf("3");`

Semáforos

Implementación

- Para evitar el *busy waiting* se modifica la definición de las operaciones `wait()` y `signal()` de la siguiente manera:
  - Cuando un proceso ejecuta `wait()`, si el valor del semáforo no es positivo debe esperar, por lo que el proceso se suspende.
  - La operación de suspensión coloca un proceso en una cola de espera asociada con el semáforo, y el estado del proceso cambia al estado de espera; luego el planificador selecciona otro proceso para ejecutar.
  - Un proceso suspendido en un semáforo `S` debe reiniciarse cuando otro proceso ejecute `signal()`.
    - El proceso se reinicia mediante una operación `wakeup()`, que cambia el estado del proceso de espera al estado de listo.

Semáforos

Implementación

```
typedef struct {
    int value;
    struct process *list;
} semaphore ;
```

- Cada semáforo tiene un valor entero **value** y una lista de procesos **list**.
- Si el valor del semáforo es negativo, su magnitud es el número de procesos que esperan en ese semáforo.

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list ;
        sleep();
    }
}
```

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
    }
}
```

Monitores

- Si bien los semáforos proporcionan un mecanismo conveniente y efectivo para la sincronización de procesos, un uso incorrecto puede generar errores difíciles de detectar.

Intercambio de `wait()` y `signal()`

```
signal(mutex);
…
critical section
…
wait(mutex);
```

- Varios procesos pueden estar en su sección crítica al mismo tiempo, lo cual es difícil de detectar.

Reemplazo de `signal()` con `wait()`

```
wait(mutex);
…
critical section
…
wait(mutex);
```

- El proceso se bloqueará permanentemente en la segunda llamada a **`wait()`**, ya que el semáforo no estará disponible.

- Supongamos que un proceso omite **`wait(mutex)`** o **`signal(mutex)`**, o ambas. En este caso, se viola la exclusión mutua o el proceso se bloqueará permanentemente.
- Estos ejemplos ilustran errores que pueden generarse cuando los programadores usan semáforos incorrectamente para resolver el problema de la sección crítica.

Monitores

Utilización

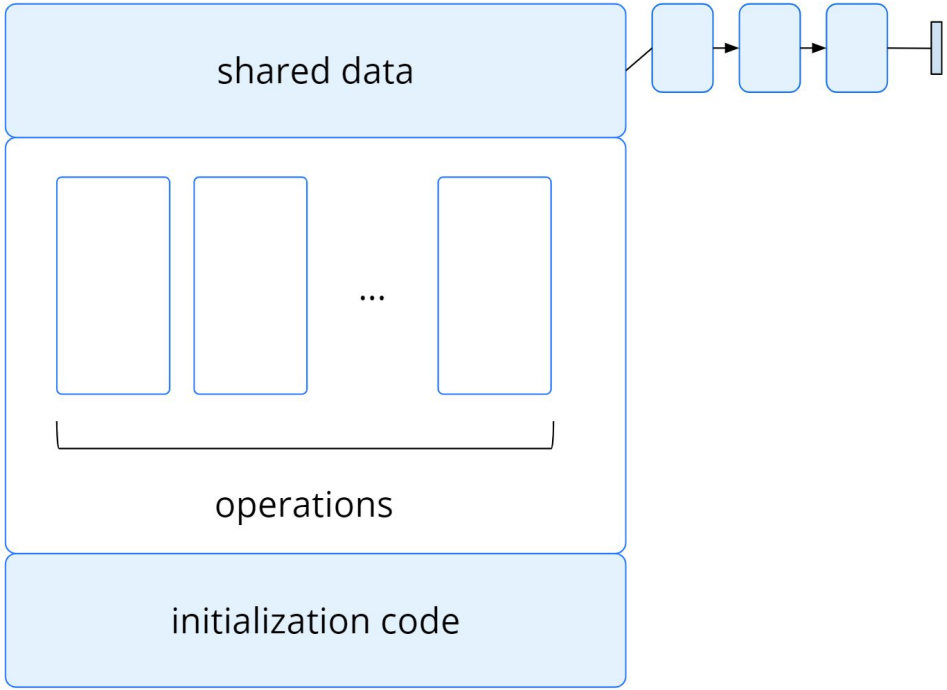
- Un monitor es un tipo de dato abstracto que incluye un conjunto de funciones definidas por el programador que son provistas con exclusión mutua dentro del monitor.
- También incluye variables, que definen el estado de una instancia de ese tipo, junto con las funciones que las operan.
- La representación de un tipo de monitor no puede ser utilizada directamente por los diversos procesos.
- Por lo tanto, una función definida dentro de un monitor puede acceder solo a aquellas variables declaradas localmente dentro del monitor y sus parámetros formales.
- Del mismo modo, solo se puede acceder a las variables locales de un monitor mediante las funciones locales.

```
monitor monitor name {
  /* shared variable declarations */
  function P1 ( . . . ) {
    . . .
  }
  function P2 ( . . . ) {
    . . .
  }
  ...
  function Pn ( . . . ) {
    . . .
  }
  initialization code ( . . . ) {
    . . .
  }
}
```

Monitores

Utilización

- La construcción del monitor asegura que solo un proceso esté activo a la vez dentro del monitor.
- Por lo tanto, el programador no necesita codificar la sincronización explícitamente.



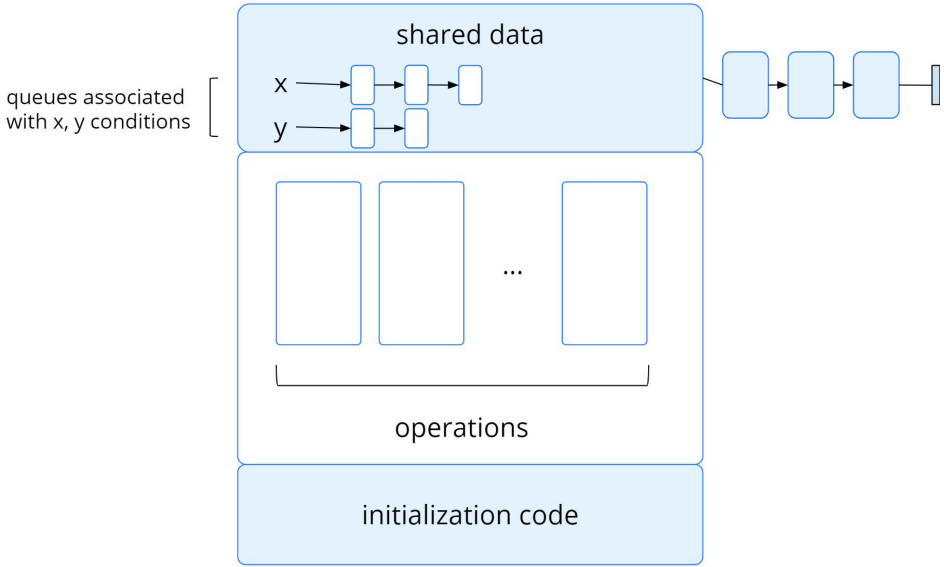
Monitores

Utilización

- Se necesitan definir mecanismos de sincronización adicionales, los cuales son proporcionados por la estructura **`condition`**.
- Un programador que necesita escribir un esquema de sincronización personalizado, puede definir una o más variables de tipo **`condition`**:

```
condition x, y;
```

- Las únicas variables que se pueden utilizar sobre estas variables son **`wait()`** y **`signal()`**:
    - **`x.wait()`**: suspende el proc.
    - **`x.signal()`**: reanuda el proc.
        - Si no hay ninguno suspendido, no tiene efecto.



Monitores

Utilización

- Supongamos que cuando la operación `x.signal()` es invocada por un proc. `P`, existe un proc. suspendido `Q` asociado con la condición `x`.
- Si `Q` reanuda su ejecución, `P` debe esperar. De lo contrario, `P` y `Q` estarían activos simultáneamente dentro del monitor. Existen dos posibilidades:
  - 1.***Signal and wait***: `P` espera hasta que `Q` abandone el monitor o espera otra condición.
  - 2.***Signal and continue***: `Q` espera hasta que `P` abandone el monitor o espera otra condición.
- Argumentos para adoptar alguna de las opciones:
  - Dado que `P` se estaba ejecutando en el monitor, el método de ***Signal and continue*** parece más razonable.
  - Pero si permitimos que el hilo `P` continúe, para el momento en que se reanude `Q`, la condición lógica por la que `Q` estaba esperando ya no se mantendrá.
  - Existe un compromiso entre estas opciones: cuando el hilo `P` ejecuta la operación de la señal, abandona inmediatamente el monitor. Por lo tanto, `Q` se reanuda inmediatamente.

Monitores

Utilización

- Un solución al problema **`productor-consumidor`** con monitores.
- Solo un procedimiento de monitor está activo a la vez.
- El búfer tiene **`N`** *slots*.

```
monitor ProducerConsumer
     condition full, empty;
     integer count;

     procedure insert(item: integer);
     begin
          if count = N then wait(full);
          insert item(item);
          count := count + 1;
          if count = 1 then signal(empty)
     end;

     function remove: integer;
     begin
          if count = 0 then wait(empty);
          remove = remove item;
          count := count - 1;
          if count = N - 1 then signal(full)
     end;

     count := 0;

end monitor;
```

Monitores

Utilización

```
procedure producer;

begin

    while true do

    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end

end;
```

```
procedure consumer;

begin
    while true do

    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end

end;
```

Liveness

- Una consecuencia del uso de herramientas de sincronización es la posibilidad de que un proceso que intenta ingresar a su sección crítica espere indefinidamente.
- La espera indefinida viola los criterios de progreso y de espera limitada.
- *Liveness* se refiere a un conjunto de propiedades que un sistema debe satisfacer para garantizar que los procesos progresen durante su ciclo de vida de ejecución.
  - Un proceso que espera indefinidamente en las circunstancias que se acaban de describir es un ejemplo de "*liveness failure*".
- Se caracterizan generalmente por bajo rendimiento y capacidad de respuesta.
- Un ejemplo muy simple de una *liveness failure* es un bucle infinito.
  - Un *loop busy wait* presenta la posibilidad de un *liveness failure*, especialmente si un proceso cicla por un período de tiempo arbitrariamente largo.

Liveness

Interbloqueo - *deadlock*

- La implementación de un semáforo con una cola de espera puede dar como resultado una situación en la que dos o más procesos esperan indefinidamente un evento que solo puede ser causado por uno de los procesos de espera.
  - El evento en cuestión es la ejecución de una operación de `signal()`.
- Cuando se alcanza ese estado, se dice que estos procesos están en punto muerto (*deadlock*).

P_0

```
wait(S);
wait(Q);
   .
   .
   .
signal(S);
signal(Q);
```

P_1

```
wait(Q);
wait(S);
   .
   .
   .
signal(Q);
signal(S);
```

- *Deadlock*: cada proceso en el conjunto está esperando un evento que solo puede ser causado por otro proceso del conjunto.

Liveness

Inversión de prioridades

- Cuando un proceso de mayor prioridad necesita leer o modificar los datos del *kernel* a los que está accediendo uno o varios procesos de menor prioridad surge un problema.
- Dado que los datos del *kernel* generalmente están protegidos con un *lock*, el proceso de mayor prioridad tendrá que esperar a que uno de menor prioridad termine de utilizar el recurso.
- La situación se vuelve más complicada si el proceso de menor prioridad es apropiado a favor de otro proceso con mayor prioridad.
- Como ejemplo, supongamos que tenemos tres procesos:
  - `L, M` y `H`, cuyas prioridades siguen el orden `L < M < H`.
- `H` requiere un semáforo `S`, al que actualmente se accede mediante el proceso `L`. Normalmente, el proceso `H` esperaría a que `L` termine de usar el recurso `S`.
- Sin embargo, supongamos que ahora el proceso `M` se vuelve ejecutable, apropiando así el proceso `L`.
- Indirectamente, un proceso de menor prioridad, `M`, ha afectado al proceso `H`, ya que debe esperar a que `L` renuncie al recurso `S`.

Liveness

Inversión de prioridades

- Esto se conoce como inversión de prioridad y solo puede ocurrir en sistemas con más de dos prioridades.
- Por lo general, la inversión de prioridad se evita mediante la implementación de un protocolo de herencia de prioridad.
- De acuerdo con este protocolo, todos los procesos que acceden a los recursos que necesita un proceso de mayor prioridad heredan la mayor prioridad hasta que terminen de utilizar los recursos en cuestión.
- Cuando terminan, sus prioridades vuelven a sus valores originales.
- En el ejemplo anterior, un protocolo de herencia de prioridad permitiría que el proceso `L` herede temporalmente la prioridad del proceso `H`, evitando así que el proceso `M` evite su ejecución.
- Cuando el proceso `L` finalice la utilización del recurso `S`, renuncia a su prioridad heredada de `H` y asume su prioridad original.
- Como el recurso `S` ahora estaría disponible, el proceso `H` es el que se ejecuta, y no el proceso `M`.