Sistemas Operativos

Sincronización

Armando Arce, Instituto Tecnológico, arce@itcr.ac.cr

Tecnológico de Costa Rica

Sincronización de procesos

Un proceso cooperativo es uno que puede afectar o ser afectado por los demás procesos que se ejecutan en el sistema.

- Los procesos cooperativos podrían compartir directamente un espacio de direcciones lógico (es decir, tanto código como datos), o bien compartir datos únicamente a través de archivos.
- El primer caso se lleva acabo mediante procesos ligeros o hilos, el segundo caso consiste en los procesos normales (pesados) del sistema.

Ejemplo - proceso productor

```
while (true) {
     /* produce an item in next_produced */

     while (counter == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

Figure 1:

Ejemplo - proceso consumidor

```
while (true) {
     while (counter == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;

     /* consume the item in next_consumed */
}
```

Figure 2:

Condiciones de competencia

El acceso concurrente a datos compartidos puede dar pie a inconsistencia en los datos.

- Una situación como esta, en la que varios procesos acceden a, y manipulan, los mismos datos de forma concurrente, y el resultado de la ejecución depende del orden en que haya ocurrido el acceso, se denomina *condición de competencia* (race condition).

Condiciones de competencia

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

Figure 3:

Condiciones de competencia

Para evitar la condición de competencia anterior, es necesario asegurarse que sólo un proceso a la vez pueda estar manipulando los datos compartidos.

- Tal garantía requiere alguna forma de sincronización de procesos.

Secciones críticas

La exclusión mutua consiste de un mecanismo que permite prohibir que más de un proceso lea o escriba en los datos compartidos a la vez.

- La sección crítica es la parte de un programa en la cual se tiene acceso a la memoria compartida.
- Si dos procesos nunca están en sus regiones críticas al mismo tiempo, se evitan las condiciones de competencia y con ello las inconsistencias de datos.
- El problema de la sección crítica consiste en diseñar un protocolo que los procesos puedan usar para cooperar.

Secciones críticas

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

Figure 5.1 General structure of a typical process P_i.

Figure 4:

Secciones críticas

Una solución al problema la sección crítica debe satisfacer los tres requisitos siguientes:

1. Exclusión mutua: Dos procesos no deben estar en sus secciones críticas al mismo tiempo.
2. Progreso: Ningún proceso, que se ejecute fuera de su sección crítica, puede bloquear a otros procesos.
3. Espera limitada: Ningún proceso debe esperar indefinidamente por entrar a su sección crítica.

En general, no se puede hacer ninguna suposición acerca de las velocidades relativas de los n procesos que se ejecutan en forma concurrente.

Alternancia estricta

Una primer estrategia, para el problema de la sección crítica, consiste en la *alternancia estricta*.

- Aquí, dos procesos compartan una variable entera común *turno* cuyo valor puede ser 0 o 1.
- Si la variable tiene valor de 0 se permite que ingrese el proceso 0 y cuando sale debe poner la variable en 1, si la variable tiene valor 1 se permite que ingrese el proceso 1 y cuando sale debe dejar la variable en 0.

Alternancia estricta

Esta estrategia, aún cuando asegura que sólo un proceso a la vez podrá usar la sección crítica, no satisface el requisito de progreso, ya que el proceso que espera depende de otro proceso que se encuentra fuera de su sección crítica.

Solución de Peterson

Una segunda estrategia (*solución de Peterson*) utiliza un arreglo adicional de indicadores.

- Cuando un proceso desea utilizar la sección crítica, activa su indicador y cede el turno al otro proceso.
- Luego dicho proceso debe verificar que sea su turno, o bien, que el otro proceso no desee ingresar.
- Al salir, desactiva su indicador. Esta solución resuelve el problema de la sección crítica pero únicamente para dos procesos.

Solución de Peterson

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Figure 5.2 The structure of process P_i in Peterson's solution.

Figure 5:

Algoritmo de la panadería

Una estrategia más general, para N procesos, es conocida como el *algoritmo de la panadería*.

- En este algoritmo cada proceso que desea entrar a su sección crítica obtiene un número consecutivo.
- Luego el proceso con el número menor es el que tiene derecho a utilizar su sección crítica.
- Sin embargo, no existe garantía que dos procesos no tengan asignado el mismo número, por lo que en dicho caso el proceso con el *PID* menor sería el primero en ingresar.

Uso de cerrojos (candados/bloqueos)

Una forma de evitar las condiciones de competencia es requerir que la regiones críticas se protejan mediante cerrojos.

- Es decir, un proceso debe adquirir un cerrojo antes de entrar en una sección crítica y liberarlo cuando salga de la misma.

Exclusión mutua mediante hardware

El soporte hardware puede facilitar cualquier tarea programación y mejorar la eficiencia del sistema.

- Muchos sistemas computacionales proporcionan instrucciones de hardware especiales que permiten consultar y modificar el contenido de una palabra o intercambiar los contenidos de dos palabras atómicamente, es decir, como una unidad de trabajo ininterrumpible.

Instrucción: Test And Set

La instrucción *TestAndSet* se utiliza para leer y modificar atómicamente una variable.

- La instrucción lee el contenido de una palabra de memoria, la coloca en un registro y luego almacena un valor distinto de cero en esa dirección de memoria.
- Si dos instrucciones de este tipo se ejecutan simultáneamente, cada una en un procesador diferente, se ejecutarán secuencialmente en un orden arbitrario.

Instrucción: Test And Set

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

   return rv;
}
```

Figure 5.3 The definition of the `test_and_set()` instruction.

Figure 6:

Instrucción: Test And Set

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Figure 5.4 Mutual-exclusion implementation with `test_and_set()`.

Figure 7:

Instrucción: Compare And Swap

La instrucción *CompareAndSwap* permite intercambiar el valor de dos variables, a diferencia de la instrucción que *TestAndSet* opera sobre los contenidos de dos palabras.

- Al igual que la instrucción *TestAndSet*, se ejecuta atómicamente.
- Si la máquina soporta esta instrucción, entonces la exclusión mutua se proporciona como sigue: se declara una variable global booleana y se inicializa como *false*.
- Además, cada proceso tiene variable local *key* con valor *true* que intercambia con la variable *lock* hasta obtener el valor *false*.

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;
}
```

Figure 5.5 The definition of the `compare_and_swap()` instruction.

Figure 8:

Instrucción: Compare And Swap

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

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

Figure 5.6 Mutual-exclusion implementation with the `compare_and_swap()` instruction.

Figure 9:

Problema de las instrucciones de hardware

Aunque estos algoritmos satisfacen el requisito de exclusión mutua, no satisfacen el requisito de espera limitada.

- Sin embargo, se pueden programar algoritmos, utilizando estas instrucciones, que garanticen dicho requisito.

Semáforos

Las diversas soluciones de hardware al problema de la sección crítica, son complicadas de utilizar por los programadores de aplicaciones.

- Para superar esta dificultad, se puede utilizar una herramienta de sincronización denominada semáforo.

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

The definition of `signal()` is as follows:

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

Figure 10:

Semáforos

Un semáforo S es una variable entera a la que, sólo se accede mediante dos operaciones atómicas indivisibles:

- wait(): Verifica si el valor es mayor que 0, y es cierto, decrementa el valor y continúa. Si el valor es 0 el proceso se pone a dormir antes de decrementar el semáforo.
- signal(): Incrementa el valor del semáforo asociado. Si uno o más procesos está durmiendo en ese semáforo, el sistema escoge uno, lo despierta y le permite terminar su operación *wait*.

Semáforos

El sistema garantiza que todas las modificaciones del valor entero del semáforo en las operaciones *wait* y *signal* se van a ejecutar de forma indivisible.

- Es decir, cuando un proceso modifica el valor del semáforo, ningún otro proceso puede modificarlo.

Semáforos

La mayor ventaja de los semáforos es que no provocan una espera activa, tal como las soluciones vistas hasta el momento, sino que los procesos dormidos se encuentran en una cola de espera y no están gastando ciclos del procesador.

Semáforos

Existen dos tipos de semáforos, los semáforos contadores y los semáforos binarios.

- El valor del semáforo contador puede variar en un dominio no restringido, mientras que el valor del semáforo binario sólo puede ser 0 ó 1.
- En algunos sistemas, los semáforos binarios se conocen como cerrojos *mutex*, ya que son cerrojos que proporcionan exclusión mutua, y se utilizan para abordar el problema de la sección crítica en el caso de múltiples procesos.

Semáforos

Los semáforos contadores se pueden usar para controlar el acceso a un determinado recurso formado por un número finito de instancias.

- El semáforo se iniciatiza con el número de recursos disponibles.
- Cada proceso que desee usar un recurso ejecuta una operación *wait()* en el semáforo, decrementando la cuenta.

Semáforos

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

Figure 11:

Semáforos

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

Figure 12:

Semáforos

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

Figure 13:

Semáforos

- Cuando un proceso libera un recurso, ejecuta la operación *signal()*, incrementando la cuenta.
- Cuando la cuenta del semáforo llega a cero, todos los recursos estarán en uso. Después, los procesos que deseen usar un recurso se bloquearán hasta que la cuenta sea mayor que 0.

Interbloqueos

La implementación de un semáforo con una cola espera pueda lugar a una situación en la que dos o más procesos estén esperando indefinidamente a que se produzca un suceso que sólo puede producirse como consecuencia de las operaciones efectuadas por otro de los procesos en espera.

- El suceso en cuestión es la ejecución de una operación *signal()*.
- Cuando se llega a un estado así, se dice que estos procesos se han interbloqueado.

Interbloqueos

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
.	.
.	.
.	.
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Figure 14:

Interbloqueos

Otro problema relacionado con los interbloqueos es el del bloqueo indefinido o muerte por inanición, una situación en la que algunos procesos esperan de forma indefinida dentro del semáforo.

- El bloqueo indefinido puede producirse si se añaden y eliminan los procesos de la lista asociada con el semáforo utilizando un esquema LIFO (último-en entrar, primero-en-salir).

Problemas clásicos de sincronización

Existen una serie de problemas clásicos que son utilizados como ejemplos para aplicar diferentes mecanismos de sincronización de procesos.

El problema del productor y el consumidor

Dos procesos comparten un buffer de tamaño fijo. Uno coloca información, mientras el otro la lee.

- El problema consiste en que el productor quiere escribir pero el buffer está lleno o bien, el consumidor quiere leer pero está vacío.
- La solución sería que el productor o el consumidor se duerman en un semáforo esperando la información.

El problema del productor y el consumidor

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

Figure 15:

El problema del productor y el consumidor

Un problema que se presenta es que suceden condiciones de competencia al utilizar la variable compartida que cuenta los elementos del buffer.

- Para evitar posibles inconsistencias, el acceso a dicha variable se asegura mediante un semáforo *mutex*.

El problema del productor y el consumidor

```
do {
      . . .
   /* produce an item in next_produced */
      . . .
   wait(empty);
   wait(mutex);
      . . .
   /* add next_produced to the buffer */
      . . .
   signal(mutex);
   signal(full);
} while (true);
```

Figure 5.9 The structure of the producer process.

Figure 16:

El problema del productor y el consumidor

```
do {
   wait(full);
   wait(mutex);
      . . .
   /* remove an item from buffer to next_consumed */
      . . .
   signal(mutex);
   signal(empty);
      . . .
   /* consume the item in next_consumed */
      . . .
} while (true);
```

Figure 5.10 The structure of the consumer process.

Figure 17:

El problema de los lectores-escritores

Este ejemplo consiste del acceso a una base de datos que debe ser compartidas por varios procesos concurrentes.

- Muchos procesos pueden leer al mismo tiempo la base de datos (lectores), solo un proceso (escritor) puede escribir en ella en un momento determinado (acceso exclusivo).

El problema de los lectores-escritores

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

Figure 18:

El problema de los lectores-escritores

Las soluciones a este problema utiliza un semáforo *mutex* y un semáforo *wrt* que se inicializan con valor de 1, mientras que una variable *readcount* se iniciativa con valor 0.

- El semáforo *wrt* es común a los pueden lectores y escritores.
- El semáforo mutex se usa para asegurar la exclusión mutua mientras actualiza la variable *readcount*.

El problema de los lectores-escritores

```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

Figure 5.11 The structure of a writer process.

Figure 19:

El problema de los lectores-escritores

La variable *readcount* almacena el número de procesos que están leyendo actualmente la base de datos.

- El semáforo *wrt* funciona como un semáforo exclusión mutua para los escritores.
- También lo utilizan el primer último lector que entran o salen de la sección crítica.
- Los lectores que entren o salgan mientras otros procesos lectores estén en sus secciones críticas no lo utilizan.

El problema de los lectores-escritores

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

Figure 5.12 The structure of a reader process.

El problema de la cena de los filósofos

Este problema ilustra una situación de competencia por un número limitado de recursos.

- En la situación clásica se cuenta con cinco filósofos, un plato por cada filósofo, y un tenedor entre cada plato.
- Los filósofos comen o piensan, y se necesita 2 tenedores para comer.
- El objetivo es maximizar la cantidad de filósofos comiendo en forma concurrente.

El problema de la cena de los filósofos

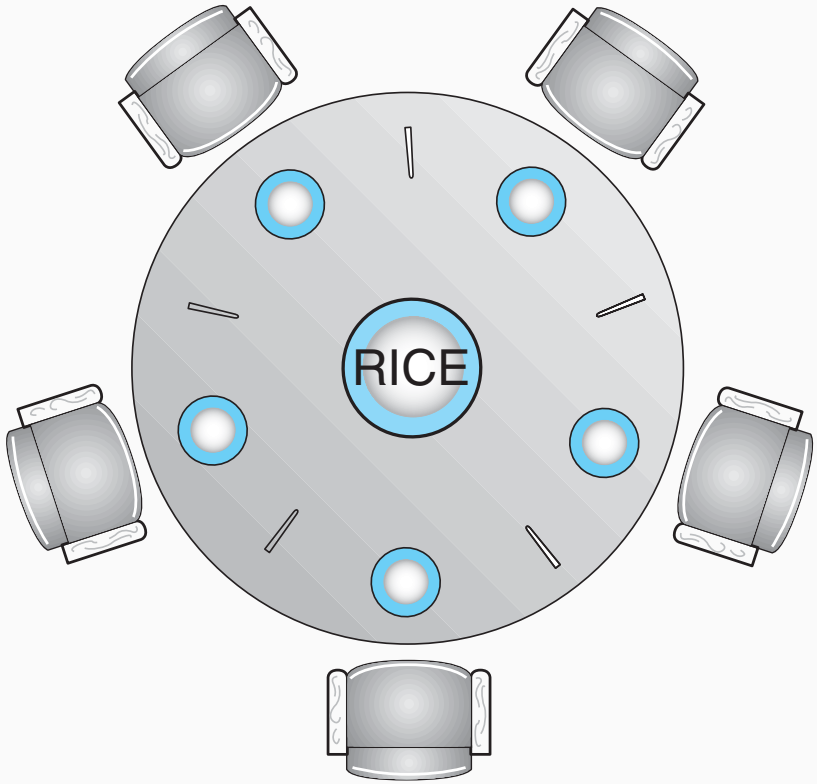

Figure 5.13 The situation of the dining philosophers.

Figure 21:

El problema de la cena de los filósofos

Una primer solución sería: tomar un tenedor y esperar hasta que el otro esté disponible.

- Sin embargo, esto puede producir un problema de interbloqueo.

El problema de la cena de los filósofos

Otra solución podría ser verificar el segundo tenedor y si no está disponible soltar el primero.

- Esto también conlleva un problema pues se podría dar un caso de inanición (en caso de varios de los procesos tomen y suelten los tenedores en forma simultánea).

El problema de la cena de los filósofos

5.8 N

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

Figure 5.14 The structure of philosopher i.

Figure 22:

El problema de la cena de los filósofos

Una nueva opción sería que cada filósofo espere un tiempo aleatorio antes de intentar tomar los tenedores de nuevo.

- Esto podría dar problemas algunas veces, por lo que es preferible una solución que siempre funcione.
- Se podría pensar en un solución que proteja las acción de tomar cualquier tenedor mediante un semáforo mutex. Sin embargo, esto acarrea un mal rendimiento, pues solo un filósofo puede comer en un momento dado.

El problema de la cena de los filósofos

Otra solución sería permitir que como máximo haya cuatro filósofos sentados a la mesa simultáneamente.

- O bien, permitir a cada filósofo coger sus tenedores sólo si ambos están disponibles.

Monitores

Aunque el uso de los semáforos proporciona un mecanismo adecuado y efectivo para el proceso de sincronización, un uso incorrecto de los mismos puede dar lugar a errores de temporización que son difíciles de detectar, dado que estos errores suceden si tienen lugar algunas secuencias de ejecución concretas y estas secuencias no siempre se producen.

Monitores

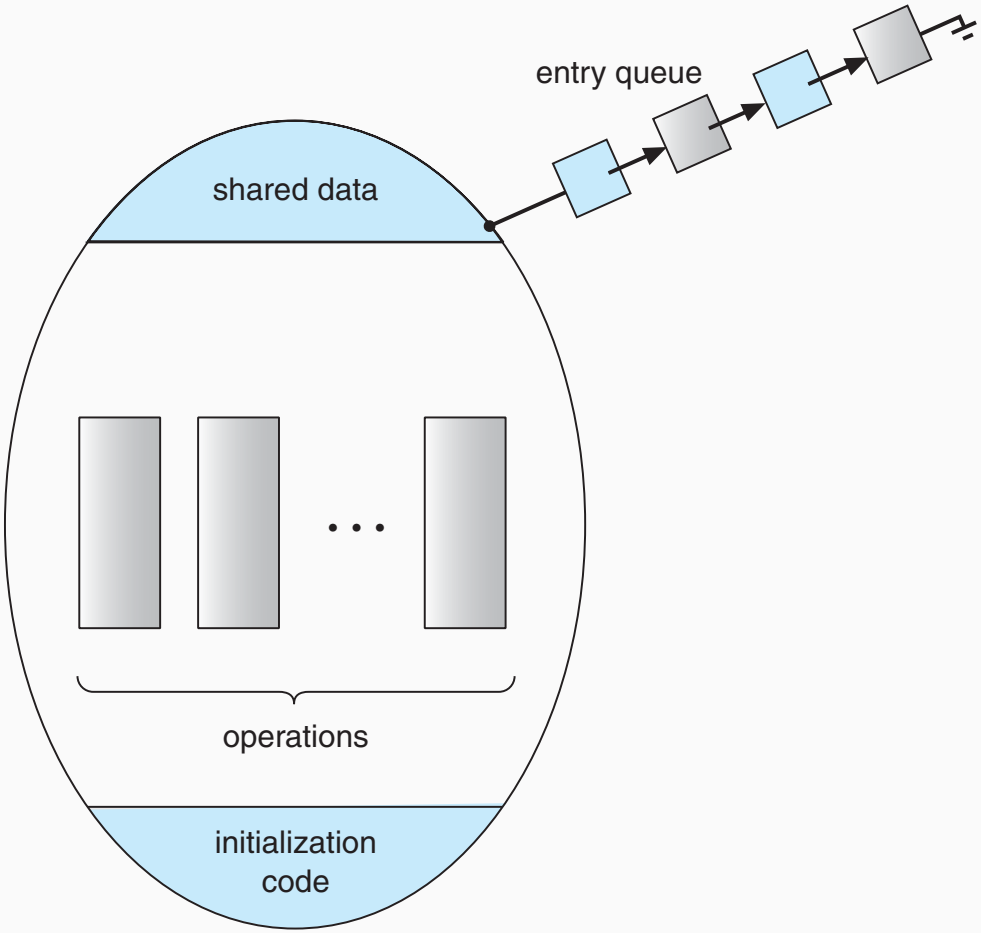

Figure 5.16 Schematic view of a monitor.

Figure 23:

Monitores

Otra construcción de sincronización de alto nivel es el tipo monitor.

- Un monitor se caracteriza por un conjunto de operadores definidos por el programador.
- La representación de un tipo monitor consiste en declaraciones de variables cuyos valores definen el estado de una instancia del tipo, así como los cuerpos de procedimientos o funciones que implementan operaciones con el tipo.

Monitores

Un monitor es un conjunto de procedimientos, variables y estructuras de datos que se agrupan en un tipo especial de módulo o paquete.

- Los procesos pueden invocar los procedimientos del monitor, pero no pueden acceder directamente a las estructuras de datos internas del monitor desde procedimientos declarados afuera del monitor.

Monitores

Los diversos programas no pueden usar directamente la representación de un tipo monitor.

- Así pues, un procedimiento definido dentro de un monitor sólo puede acceder a las variables declaradas localmente dentro del monitor, y a los parámetros formales.
- De forma similar, sólo los procedimientos locales pueden acceder a las variables locales de un monitor.

Monitores

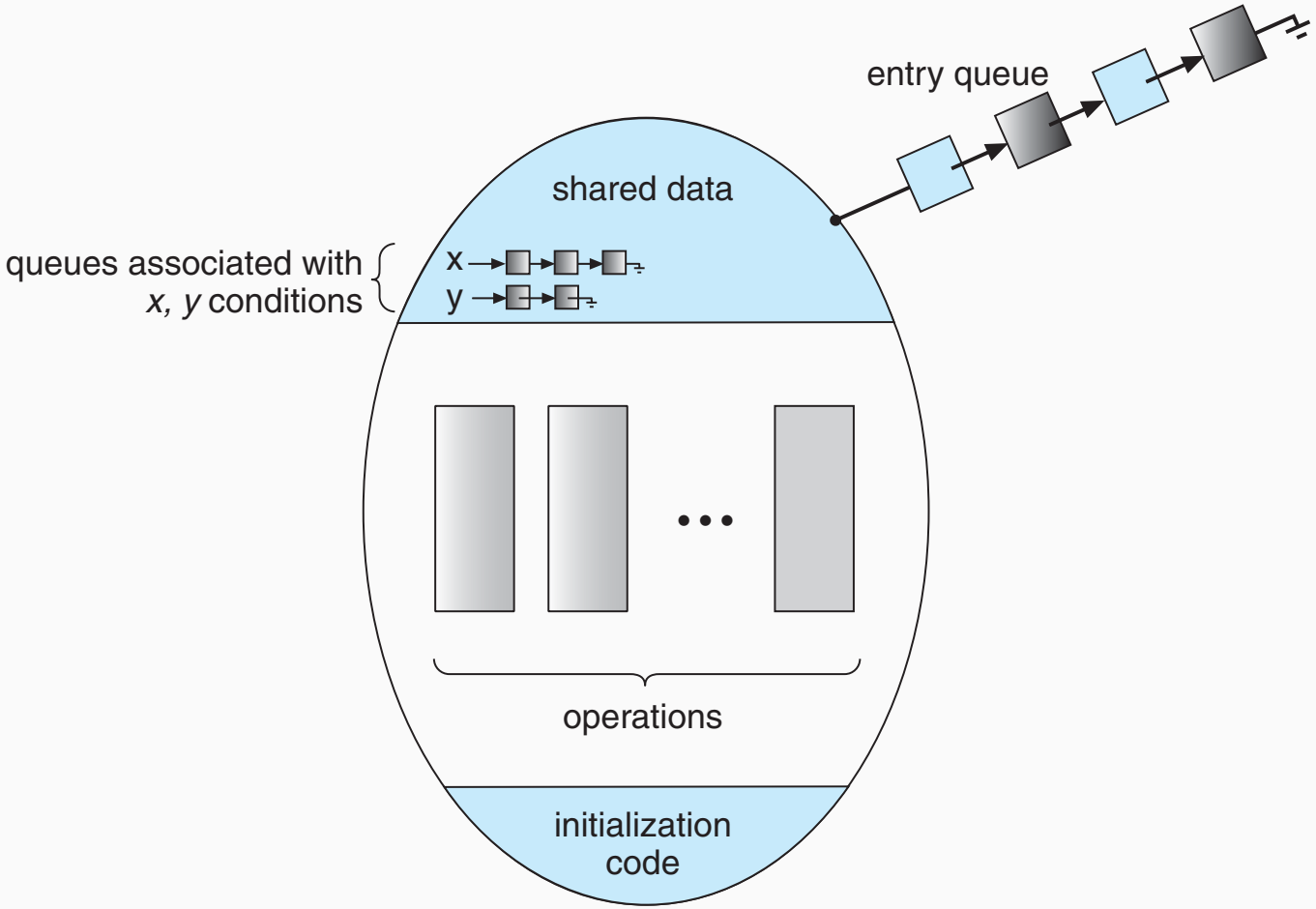

Figure 5.17 Monitor with condition variables.

Figure 24:

Monitores

La construcción de un monitor garantiza que sólo podrá estar activo un proceso a la vez dentro del monitor.

- Las secciones críticas se ejecutan como procedimientos del monitor.
- El compilador se encarga de controlar los ingresos, generalmente por medio de un semáforo.
- En consecuencia, el programador no necesita codificar esta restricción de sincronización explícitamente.

Referencia bibliográfica

A.SILBERSCHATZ, P. GALVIN, y G. GAGNE, Operating Systems Concepts, Cap. 5, 9a Edición, John Wiley, 2013.