Chapter 6: Process Synchronization

Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Synchronization Examples

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {

        /*  produce an item and put in nextProduced  */
         while (count == BUFFER_SIZE)
                 ; // do nothing
          buffer [in] = nextProduced;
          in = (in + 1) % BUFFER_SIZE;
          count++;
}
```

Consumer

```
while (true)  {
        while (count == 0)
                ; // do nothing
        nextConsumed =  buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;

        /*  consume the item in nextConsumed
}
```

Race Condition （竞态条件）

- count++ could be implemented as

  register1 = count
  register1 = register1 + 1
  count = register1

- count-- could be implemented as

  register2 = count
  register2 = register2 - 1
  count = register2

两个线程并发访问同一个变量，不同的调度顺序可能产生不同的结果

- Consider this execution interleaving with “count = 5” initially:

  S0: producer execute register1 = count {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = count {register2 = 5}
  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute count = register1 {count = 6 }
  S5: consumer execute count = register2 {count = 4}

A practical definition of Race Condition

- “A race condition is a situation in which a memory location is accessed concurrently, and at least one access is a write.”

- Additional Readings: Section 6.1 of “xv6: a simple, Unix-like teaching operating system”

Critical-section problem

并行/并发读写，多
线程或多进程

- To design a protocol that the processes can use to cooperate

```
Do {
        Entry section
        Critical section
        Exit section
        Remainder section
}while(TRUE);
```

Critical section可能包含读（也要保证读的顺序逻辑正确，写前读还是写后读）和写shared memory location

如果critical section的多条语句读写的变量不相关，可以拆分成非原子操作的话，则可以拆分成多个critical section，可以提高并行程度

General structure of a typical process Pj

- Q: critical section problems in OS kernel

Solution to Critical-Section Problem

互斥

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

空闲让进

2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

有限等待

3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the N processes

Algorithm 1

- int turn; turn = 0; // Pi can enter the critical section
- Process Pi:

只有turn=i时才能跳出while进入critical section

```
do{
    while(turn != i);
        critical section
    turn=j;
        remainder section
} while (1);
```

- Process Pj:

```
do{
    while(turn != j);
        critical section
    turn=i;
        remainder section
} while (1);
```

- Mutual Exclusion is satisfied. How about Progress?

不满足(如果pj后退出critical section且更早再次想进)

Algorithm 2

- boolean flag[2]; flag[0] = flag[1] = 0;
- flag[i] = true if Pi tries to enter CS
- Process Pi:

```
do{
    flag[i]=true;
    while( flag[j] );
        critical section
    flag[i]=false;
        remainder section
} while (1);
```

不满足，如果flag[i]=true后调度到flag[j]=true，则死循环

- Process Pj:

```
do{
    flag[j]=true;
    while( flag[i] );
        critical section
    flag[j]=false;
        remainder section
} while (1);
```

- Mutual Exclusion is satisfied. How about Progress?

Algorithm 3

- boolean flag[2]; flag[0] = flag[1] = 0;

- Process Pi:

```
do{
    while(flag[j]); // ①
    flag[i]=TRUE; // ③
        critical section;
    flag[i] = FALSE;
    remainder section;
}while(1);
```

- Process Pj:

```
do{
    while(flag[i]); // ②
    flag[j] =TRUE; // ④
      critical section;
    flag[j] = FALSE;
    remainder section;
} while (1);
```

- Is Mutual Exclusion satisfied?

不满足，两个进程可以同时进入critical section

Peterson's Solution

- Two-process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

The Algorithm for Process P_i

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

            CRITICAL SECTION

        flag[i] = FALSE;

             REMAINDER SECTION

 }
```

满足有限等待，一个进程最多连续进入critical section一次，因为再次进入while循环前将turn交给了等在while的另一个进程

★

The Respective Algorithm for Process P_j

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

              CRITICAL SECTION

        flag[j] = FALSE;

                REMAINDER SECTION

}
```

If two processes are running the statement simultaneously, only one will last.

★

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

TestAndSet Instruction

- Definition:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

Solution using TestAndSet

- Shared boolean variable lock., initialized to false.
- Solution:

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

                //    critical section

        lock = FALSE;

                //      remainder section

}
```

Swap Instruction

- Definition:

```
void Swap (boolean *a, boolean *b)
{
        boolean temp = *a;
        *a = *b;
        *b = temp:
}
```

Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key.
- Solution:

```
while (true)  {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );

                //    critical section

         lock = FALSE;

                //      remainder section

}
```

The compare_and_swap (CAS) Instruction

- Definition

```
int compare_and_swap(int *value, int expected, int new_value)
{
        int temp = *value;
        if (*value == expected)
           *value = new_value;
        return temp;
}
```

- Properties
  - Executed atomically
  - Returns the original value of passed parameter **`value`**
  - Set the variable **`value`** the value of the passed parameter **`new_value`** but only if **`*value == expected`** is true. That is, the swap takes place only under this condition.

Solution using compare_and_swap

- Shared integer `lock` initialized to 0;
- Solution:

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

        /* critical section */

        lock = 0;

         /* remainder section */
 }
```

- Does it solve the critical-section problem?

Bounded-waiting with compare-and-swap

```
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 */
}
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First `acquire()` a lock
  - Then `release()` the lock
- Calls to `acquire()` and `release()` must be **atomic**
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Solution to CS Problem Using Mutex Locks

```
while (true) {
        acquire lock

            critical section

        release lock

        remainder section
}
```

Definition of acquire() & release()

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

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

Q: How to implement the acquire() using atomic instructions such as compare-and-swap?

Semaphore

- Synchronization tool that is less complicated
- Semaphore *S* – integer variable
- Two indivisible operations modify S:
  - wait() and signal()
  - originally called P() and V()
- Can only be accessed via two indivisible (atomic) operations
  - ```
    wait (S) {
        while S <= 0
            ; // no-op
        S--;
    }
    ```
  - ```
    signal (S) {
        S++;
    }
    ```
- Can be implemented without busy waiting

Usage as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore S;    //  initialized to 1
wait (S);
        Critical Section
signal (S);
```

Usage as General Synchronization Tool(2)

- P1 has a statement S1, P2 has S2
- Statement S1 to be executed before S2

P1

```
S1;
Signal(S);
```

Question: What's the initial value of S?

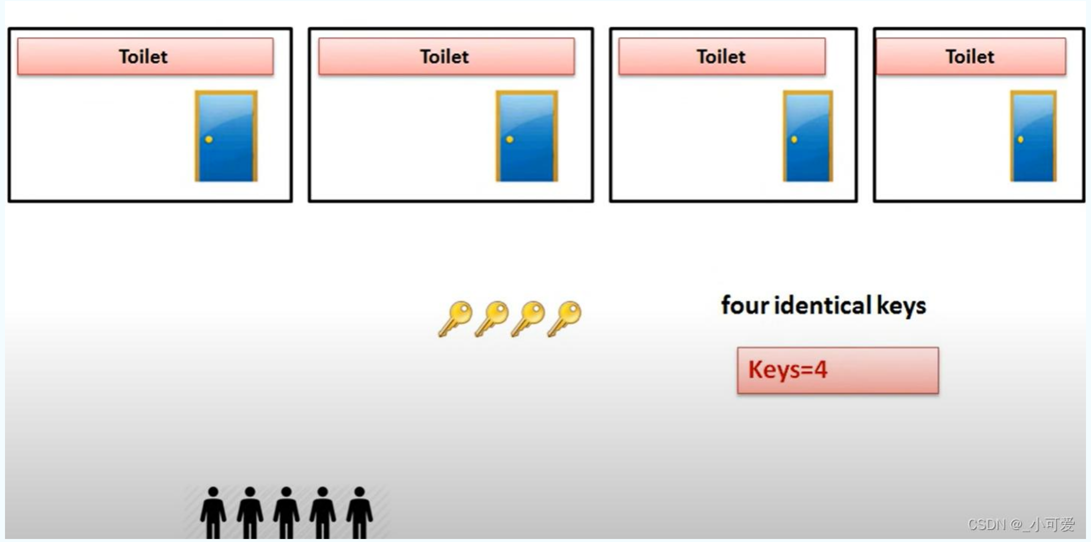
P2

```
Wait(S);
S2;
```

Usage as General Synchronization Tool (3)

Provide synchronized access to:

- Four rooms, four identical keys



- Four rooms, each with a unique key (four different keys)

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.
- See Example1

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each semaphore has two data items:
  - value (of type integer)
  - pointer to a linked-list of PCBs.
- Two operations (provided as basic system calls):
  - block (sleep) – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
wait (S){
        value--;
        if (value < 0) {
                // add this process to waiting queue
                block();  }
    }
```

- Implementation of signal:

```
Signal (S){
        value++;
        if (value <= 0) {
                // remove a process P from the waiting queue
                wakeup(P);  }
    }
```

- See Example 2

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

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

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0, counting full items
- Semaphore empty initialized to the value N, counting empty items.

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true)  {

        //   produce an item
    wait (empty);
    wait (mutex);

        //  add the item to the  buffer

    signal (mutex);
    signal (full);
}
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
        wait (full);
        wait (mutex);

                //  remove an item from  buffer

        signal (mutex);
        signal (empty);

                //  consume the removed item
}
```

Exercise:

- 三个进程 P1、P2、P3互斥使用一个包含 N（N>0）个单元的缓冲区。
- P1 每次用 produce（）生成一个正整数并用 put（）送入缓冲区某一空单元中；
- P2每次用 getodd（）从该缓冲区中取出一个奇数并用 countodd（）统计奇数个数；
- P3 每次用 geteven（）从该缓冲 区中取出一个偶数并用 counteven（）统计偶数个数。
- 请用信号量机制实现这三个进程的同步与互斥活动，并说明所定义的信号量的含义。要求用伪代码描述。

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write.
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1, to ensure mutual exclusion when readcount is updated.
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
while (true) {
        wait (wrt) ;

            //    writing is performed

        signal (wrt) ;
}
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
while (true) {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)  wait (wrt) ;
        signal (mutex)

                // reading is performed

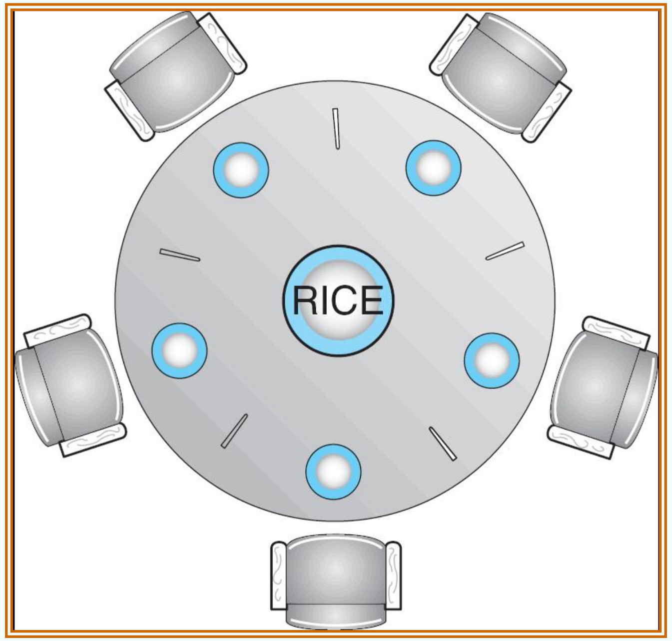
        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)  signal (wrt) ;
        signal (mutex) ;
}
```

"Locking" the wrt semaphore, rather than "waiting"

Reason is that wrt is initialized to "1"

"Unlocking" the wrt semaphore, rather than "signaling"

Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

```
While (true)  {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

                //  eat

        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

                //  think

}
```

Problems with Semaphores

- Correct use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

  (*hint*: the other processes may be sleeping within the monitor)

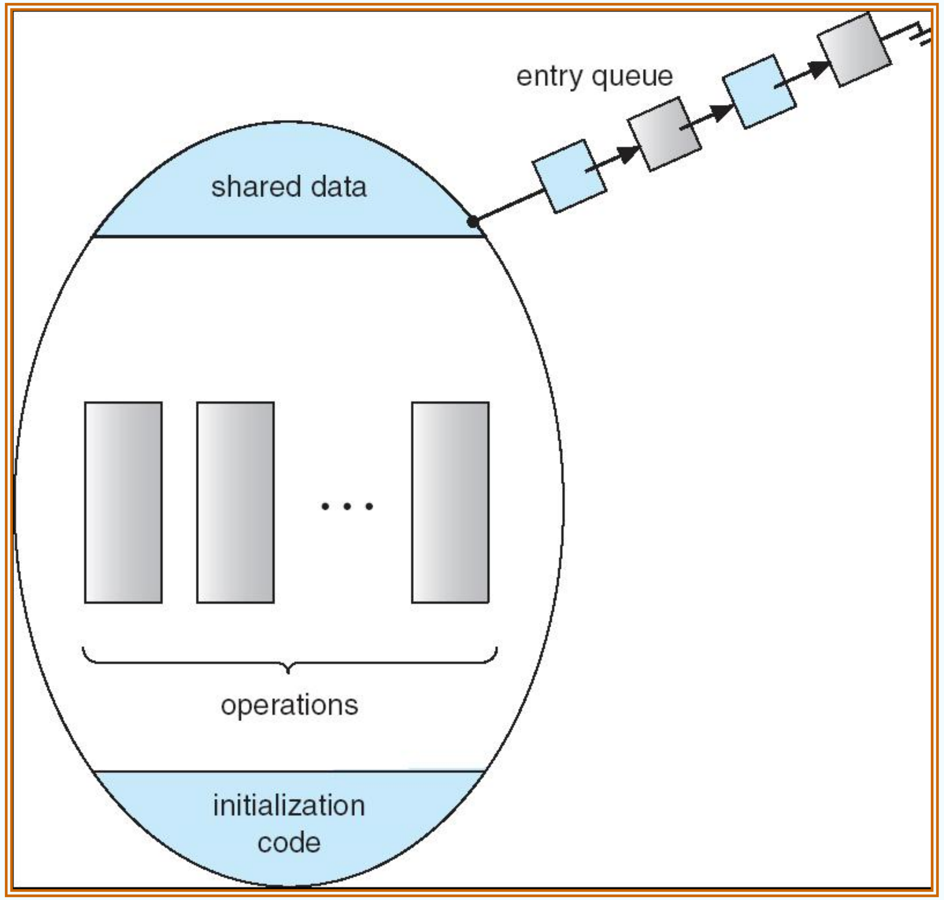
```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (…) { …. }
            …

    procedure Pn (…) {……}

     Initialization code ( ….) { … }
            …

}
```

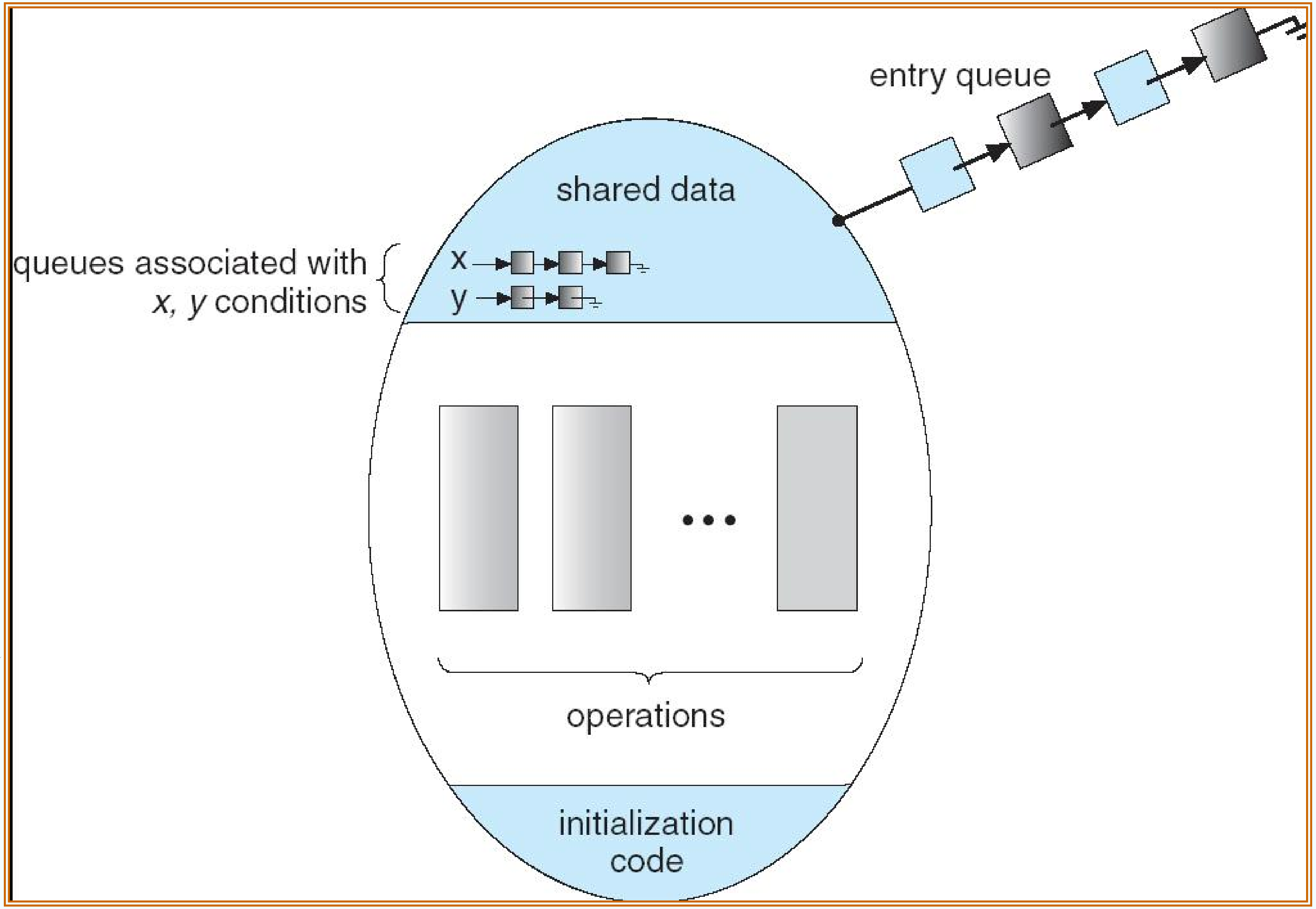
Schematic view of a Monitor



Condition Variables

- condition x, y;
- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

Monitor with Condition Variables



Solution to Dining Philosophers

```
monitor DP
  {
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];  //philosopher i can delay herself when unable to get
    chopsticks

    void pickup (int i) {
           state[i] = HUNGRY;
           test(i);
           if (state[i] != EATING) self [i].wait;
    }

     void putdown (int i) {
           state[i] = THINKING;
                // test left and right neighbors
            test((i + 4) % 5);
            test((i + 1) % 5);
     }
```

Solution to Dining Philosophers (cont)

```
void test (int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
             state[i] = EATING ;
              self[i].signal () ;
         }
 }

 initialization_code() {
       for (int i = 0; i < 5; i++)
       state[i] = THINKING;
}
}
```

Solution to Dining Philosophers (cont)

- Each philosopher *I* invokes the operations pickup() and putdown() in the following sequence:

  dp.pickup (i)

  EAT

  dp.putdown (i)

- When the left and right philosophers, **self[(i+4)%5]** and **self[(i+1)%5]** continue to eat, **self[i]** may starve.

Monitor Implementation Using Semaphores

- Variables

  semaphore mutex; // (initially = 1), entry protection

  semaphore next; // (initially = 0), signaling process may suspend themselves.

  int next-count = 0;

- Each procedure *F* will be replaced by

```
wait(mutex);
    …
            body of F;

      …
if (next-count > 0)
    signal(next)
else
    signal(mutex);
```

Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore "next" is introduced, on which the signaling process may suspend themselves.

- Mutual exclusion within a monitor is ensured.

Monitor Implementation

- For each condition variable ***x***, we have:

```
semaphore x-sem; // (initially  = 0)
int x-count = 0;
```

- The operation x.wait can be implemented as:

```
x-count++;
if (next-count > 0)
    signal(next);
else
    signal(mutex);
wait(x-sem);
x-count--;
```

If someone has been waiting, wake her up because I'll be entering the waiting state.

No one else waiting in the monitor. I'm going to block. Allow someone else to enter the monitor now.

Monitor Implementation

- The operation x.signal can be implemented as:

```
if (x-count > 0) {
    next-count++;
    signal(x-sem);
    wait(next);
    next-count--;
}
```

This is the signaling process. It will wait on the “next” semaphore.

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments (page 218)
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks (busy-waiting semaphore) on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

Linux Synchronization

- Linux:
  - disables interrupts to implement short critical sections
- Linux provides:
  - semaphores
  - spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables

- Non-portable extensions include:
  - read-write locks
  - spin locks

Pthread mutex example

- ```
  void reader_function ( void );
  void writer_function ( void );

  char buffer;
  int buffer_has_item=0;
  pthread_mutex_t mutex;
  struct timespec delay;
  void main ( void ){
     pthread_t reader;

     delay.tv_sec = 2;
     delay.tv_nec = 0;

     pthread_mutex_init (&mutex,NULL);
     pthread_create(&reader, pthread_attr_default, (void *)&reader_function), NULL);
     writer_function( );
  }
  ```
  ```

The writer thread

- ```
  void writer_function (void){
     while(1){

        pthread_mutex_lock (&mutex);
        if (buffer_has_item==0){
           buffer=make_new_item( );
           buffer_has_item=1;
        }

        pthread_mutex_unlock(&mutex);
        pthread_delay_np(&delay);
     }
  }
  ```
  ```

The reader thread

- ```
  void reader_function(void){
      while(1){
          pthread_mutex_lock(&mutex);
          if(buffer_has_item==1){
              consume_item(buffer);
              buffer_has_item=0;
          }
          pthread_mutex_unlock(&mutex);
          pthread_delay_np(&delay);
      }
  }
  ```
  ```

Using pthread_cond_wait() & pthread_cond_signal()

```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned count;
```

```
increment_count()
{
    pthread_mutex_lock(&count_lock);
    if (count == 0)

pthread_cond_signal(&count_nonzero);
    count = count + 1;
    pthread_mutex_unlock(&count_lock);
}
```

```
decrement_count()
{
    pthread_mutex_lock(&count_lock);
    while (count == 0)
        pthread_cond_wait(&count_nonzero, &count_lock);
    count = count - 1;
    pthread_mutex_unlock(&count_lock);
}
```

End of Chapter 6