Process Synchronization

Objectives

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors

- Background

- Introduction +
- Bounded buffer +
- Race Condition +

-- Introduction

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

-- Bounded-Buffer ...

Shared Variables

```
#define BUFFER-SIZE 10
Typedef struct {
. . .
} item;

Item buffer[BUFFER_SIZE];
Int in = 0;
Int out = 0;
```

Producer

```
while(1) {
    while (((in + 1) % BUFFER_SIZE) == out)
          ; /* do nothing */
    buffer[in] = nextProduced
     in = (in + 1) % BUFFER_SIZE;
}
```

Consumer

```
while(1)  {
        while (in == out)
             ;  /*  do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
}
```

... -- Bounded-Buffer ...

- Suppose that we modify the producer-consumer code of chapter 3 (inorder to use all the 10 buffers at the same time) by adding a variable *counter* initialized to 0 and incremented each time a new item is added to the buffer

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

... -- Bounded-Buffer ...

- Producer process

```
item nextProduced;

while (1) {
  while (counter == BUFFER_SIZE)
      ; /* do nothing */
   buffer[in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```

- Consumer process

```
     item nextConsumed;

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

... -- Bounded-Buffer ...

- The statement "count++" may be implemented in machine language as:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- The statement "count--" may be implemented as:

```
register2 = counter
register2 = register2 - 1
counter = register2
```

... -- Bounded-Buffer ...

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.
- Interleaving depends upon how the producer and consumer processes are scheduled.

... -- Bounded-Buffer

- Assume **counter** is initially 5. One interleaving of statements is:

```
producer: register1 = counter (register1 = 5)
producer: register1 = register1 + 1 (register1 = 6)
consumer: register2 = counter (register2 = 5)
consumer: register2 = register2 – 1 (register2 = 4)
producer: counter = register1 (counter = 6)
consumer: counter = register2 (counter = 4)
```

- The value of **counter** may be either 4 or 6, where the correct result should be 5.

... -- Bounded-Buffer ...

- The statements

  ```
  counter++;
  counter--;
  ```

  must be performed *atomically*.

- Atomic operation means an operation that completes in its entirety without interruption.

-- Race Condition

- **Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, concurrent processes must be **synchronized**.

- The Critical-Section Problem

- *n* processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

- Solution to Critical-Section (CS) Problem

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

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

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.

-- Initial Attempts to Solve CS problem

- General Structure of Processes with CS +
- Two-process Solution +
  - Algorithm 1 +
  - Algorithm 2 +
  - Algorithm 3 (Peterson’s solution)+

-- General Structure of a Process with CS.

- General structure of process

```
do {
    entry section
        critical section
    exit section
        reminder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.

--- Two-Process Solution - Algorithm 1

- For the next 3 algorithms assume two processes A and B.
- Shared variable: char turn;
  - initially turn = A
  - turn = A ⇒ Process A can enter its critical section

Process A

```
do {
   while (turn != A);
      critical section
  turn = B;
  reminder section
} while (1);
```

Process B

```
do {
   while (turn != B);
      critical section
  turn = A;
  reminder section
} while (1);
```

- Satisfies mutual exclusion, but not progress

--- Two-Process Solution - Algorithm 2

- Shared variables: boolean flag[A-B];
  - initially flag [A] = flag [B] = false.
  - flag [A] = true ⇒ Process A ready to enter its critical section

flag [A] = True; flag [B] = false;

Process A

```
do {
    flag[A] := true;
    while (flag[B]) ;
        critical section
    flag [A] = false;
    remainder section
} while (1);
```

Process B

```
do {
    flag[B] := true;
    while (flag[A]) ;
        critical section
    flag [B] = false;
    remainder section
} while (1);
```

- **Satisfies mutual exclusion, but not progress requirement.**

--- Two-Process Solution – Algorithm 3 (Peterson’s solution)

- Combined shared variables of algorithms 1 and 2.

Process A

```
do {
   flag [A]:= true;
   turn = B;
   while (flag [B] and turn == B) ;
        critical section
  flag [A] = false;
  remainder section
} while (1);
```

Process B

```
do {
   flag [B]:= true;
   turn = A;
   while (flag [A] and turn == A) ;
        critical section
  flag [B] = false;
  remainder section
} while (1);
```

- **Meets all three requirements; solves the critical-section problem for two processes.**

- Synchronization Hardware

- Many systems provide hardware support for critical section code.This make programming task easier and improve system efficiency.
- 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** (non-interruptable) hardware instructions
  - Either test memory word and set value (**TestAndSet**)
  - Or swap contents of two memory words (**Swap**)
- If two atomic instructions are executed simultaneously (each one on different CPU), they will be executed sequentially in some arbitrary order.
- Unfortunately for hardware designers, implementation of these atomic instructions in a multiprocessor environment is hard.

-- Test and Set

- Test and modify the content of a word atomically

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

--- Mutual Exclusion with Test-and-Set

- Shared data:

  boolean lock = false;

- Process P_i

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

              //    critical section

        lock = FALSE;

              //     remainder section
}
```

-- Swap

- Atomically swap two variables.

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

--- Mutual Exclusion with Swap

- Shared data (initialized to **false**): boolean lock;
- Each process has a local Boolean variable key.
- Process P_i

```
while(true)
    key = true;
    while (key == true)
                Swap(&lock,  &key);
        // critical section
    lock = false;
        // remainder section
}
```

--- Bounded-waiting mutual-exclusion with TestAndSet - (Pi)

```
While(true)
     waiting[i] = true
    key = true /* local */
     while (waiting[i] && key )
         key = TestAndSet(lock);
     waiting[i] = false;
        // critical section
      k = (i + 1) % n
      while ((k != i) && !waiting[k])
          k = (k + 1) % n
       if (k == i )
            lock = false;
       else
            waiting[k] = false;
       // remainder section
}
```

- Semaphores

- Definition +
- Semaphores Usage: CS of n Processes +
- Semaphore Usage: a General Synchronization Tool +
- Implementation +
- Deadlock and Starvation +
- Two Types of Semaphores +

-- Definition

- Synchronization tool that does not require busy waiting.
- Semaphore *S* – integer variable
- Apart from initialization it can only be accessed via two indivisible (atomic) operations

wait (*S*)
{
   while $S \le 0$;
   S--;
}
	signal (*S*)
	{
	   S++;
	}

-- Semaphore Usage: CS of n Processes

- Shared data:

```
semaphore mutex; // initially mutex = 1
```

- Process Pi:

```
do {
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
} while (1);
```

-- Semaphore Usage: a General Synchronization Tool

- Execute *B* in P_j only after *A* executed in P_i
- Use semaphore *flag* initialized to 0
- Code:

$\underline{P_i}$	$\underline{P_j}$
⋮	⋮
A	*wait(flag)*
signal(flag)	*B*

-- Semaphore Implementation ...

- To avoid spinlock define a semaphore as a record

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

- Assume two simple operations:
  - block suspends the process that invokes it.
  - wakeup(*P*) resumes the execution of a blocked process P.

... -- Semaphore Implementation

- Semaphore operations now defined as

```
wait(S)
{
    S.value--;
     if (S.value < 0) {
          add this process to S.L;
          block;
}
```

```
signal(S)
{
    S.value++;
     if (S.value <= 0) {
          remove a process P from S.L;
          wakeup(P);
 }
```

-- 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 ...

- Shared data

```
semaphore full, empty, mutex;
```

  Initially:

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

- Assume that the buffer consists of ***n*** buffers, each capable of holding one item.
- The ***mutex*** semaphore provides mutual exclusion to the buffer pool.
- The ***empty*** semaphores count the number of empty buffers.
- The ***full*** semaphore counts the number of full buffers.

… -- Bounded-Buffer Problem

Producer

```
do {
        …
     produce an item in nextp
        …
     wait(empty);
     wait(mutex);
        …
     add nextp to buffer
        …
     signal(mutex);
     signal(full);
} while (1);
```

Consumer

```
do {
     wait(full)
     wait(mutex);
        …
     remove an item from buffer to nextc
        …
     signal(mutex);
     signal(empty);
        …
     consume the item in nextc
        …
     } while (1);
```

-- Readers-Writers Problem ...

- Two types of processes:
  - **Writers**: modify a shared object
  - **Readers**: They just read. They do not modify shared object.

- Many readers can access a shared object simultaneously.

- A writer needs exclusive access to a shared object

- The Readers-Writers problem has several variations, all involving priorities.

... -- Readers-Writers Problem

- Shared data

  ```
  semaphore mutex, wrt;
  ```

  Initially

  ```
  mutex = 1; wrt = 1; readcount = 0
  ```

- The ***wrt*** semaphore is common to both readers and writers
- The ***mutex*** semaphore is to ensure mutual exclusion when the variable ***readcount*** is updated.

--- Readers-Writers Problem - Writer Process

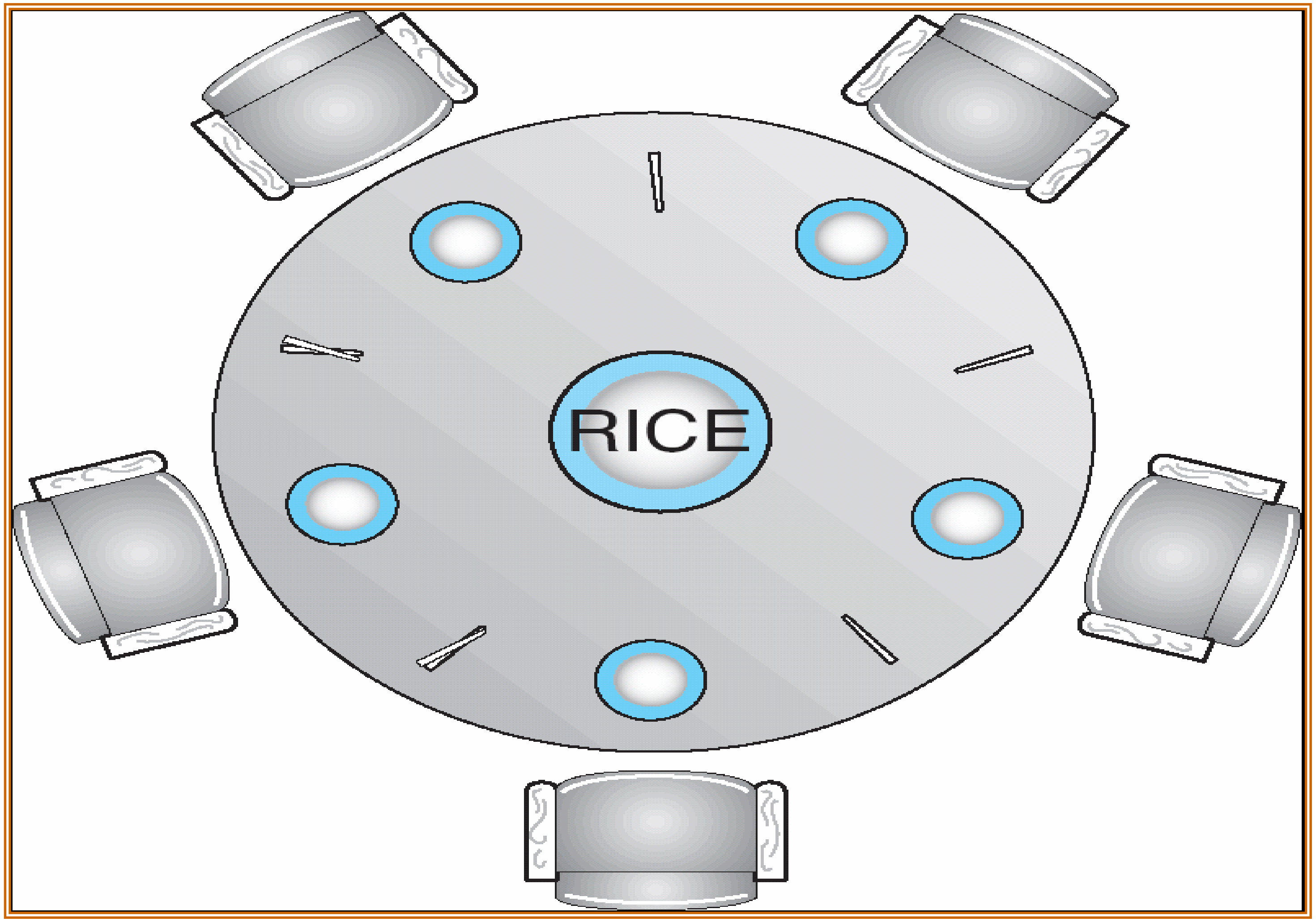
Writer Process

```
wait(wrt);
     …
  writing is performed
     …
signal(wrt);
```

Reader Process

```
wait(mutex);
readcount++;
if (readcount == 1)
     wait(wrt);
signal(mutex);
     …
     reading is performed
     …
wait(mutex);
readcount--;
if (readcount == 0)
     signal(wrt);
signal(mutex):
```

-- Dining-Philosophers Problem



-- Dining-Philosophers Problem ...

- Shared data

  ```
  semaphore chopstick[5];
  ```

  Initially all values are 1

- <u>Philosopher *i* :</u>

```
do {
    wait(chopstick[i])
    wait(chopstick[(i+1) % 5])
        …
        eat
        …
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
        …
        think
        …
} while (1);
```

- 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

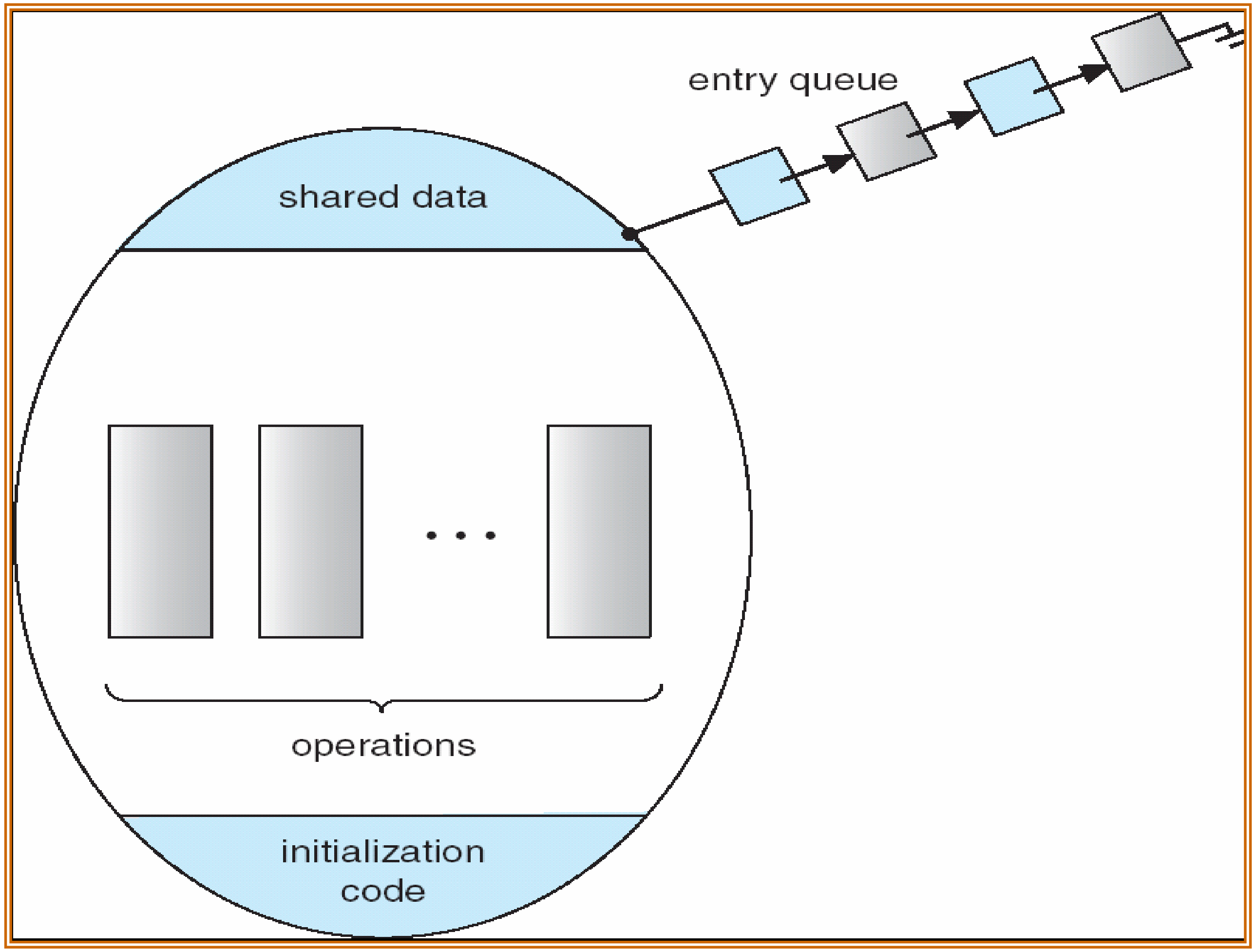
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

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

   procedure Pn (…) {……}

    Initialization code ( ….) { … }
            …
   }
}
```

-- Schematic view of a Monitor

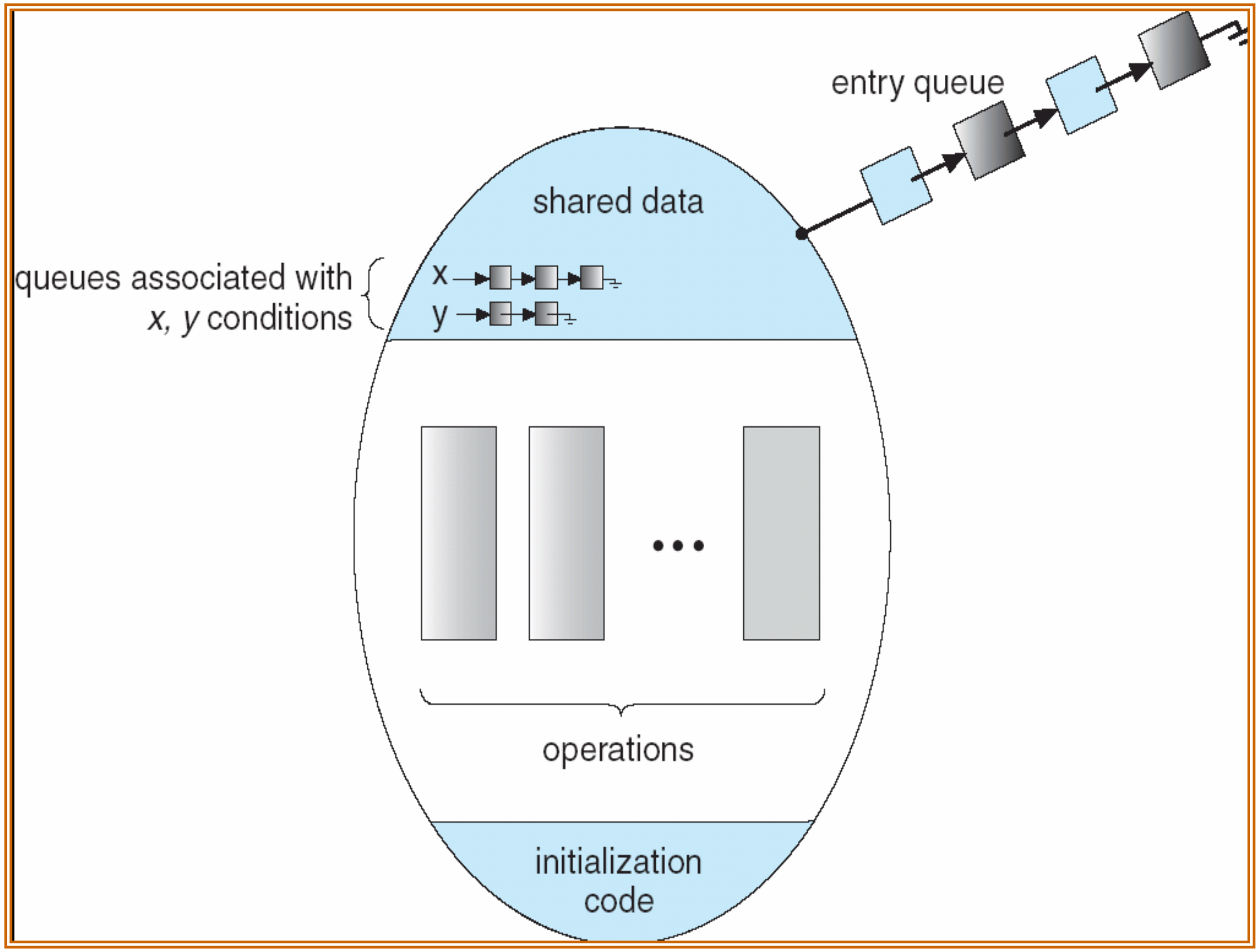


-- Condition Variables

- 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



Chapter summary

- Race condition, atomic operation
- Critical Section (CS) : Where shared data is modified
- Solution to CS:
  - Mutual exclusion
  - Progress
  - Bounded waiting
- Peterson’s Solution
- Synchronization HW:
  - Test and Set
  - Swap
- Semaphores: with spin lock and without spin lock
- Classic Problems of Synchronization
  - Bounded buffer
  - Readers writers,
  - Five philosophers
- Monitors

End of Chapter 6