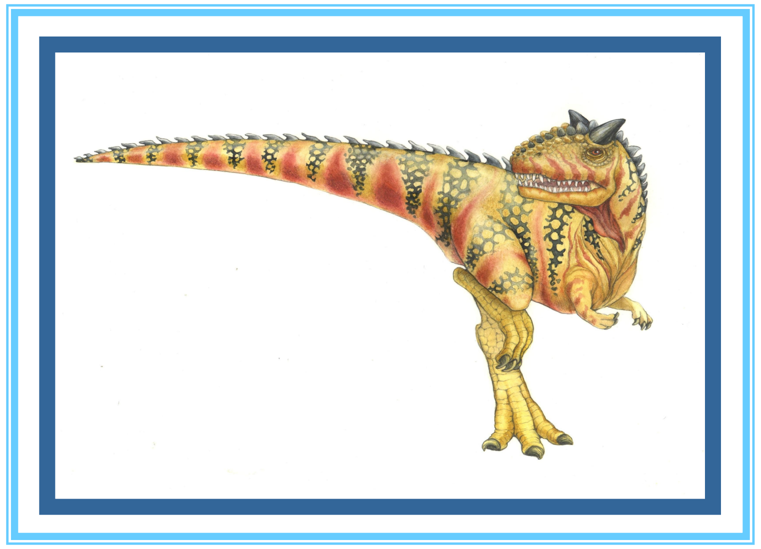
Chapter 6: Process Synchronization



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

Consumer

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

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

Race Condition

- counter++ could be implemented as

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

- counter-- could be implemented as

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

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

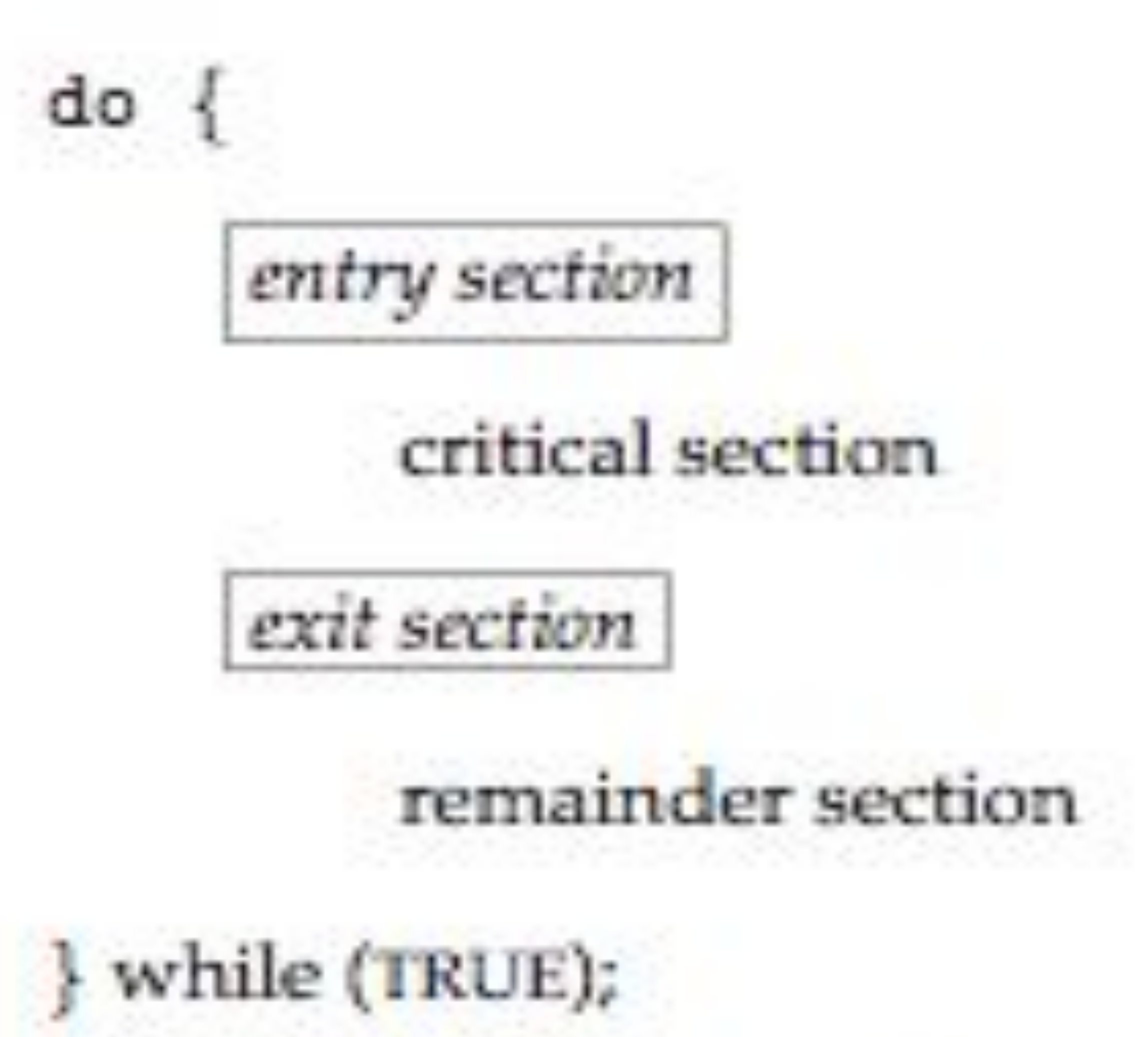
```
S0: producer execute register1 = counter   {register1 = 5}
 S1: producer execute register1 = register1 + 1   {register1 = 6}
 S2: consumer execute register2 = counter   {register2 = 5}
 S3: consumer execute register2 = register2 - 1   {register2 = 4}
 S4: producer execute counter = register1   {count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

Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process p_i is



```
do {

     entry section

          critical section

     exit section

          remainder section

} while (TRUE);
```

Figure 6.1 General structure of a typical process P_i.

Solution to Critical-Section 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

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 $\mathbf{P_i}$ is ready!

Algorithm for Process P_i

```
do {
     flag[i] = TRUE;
     turn = j;
     while (flag[j] && turn == j);
          critical section
     flag[i] = FALSE;
          remainder section
} while (TRUE);
```

- Provable that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

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
    - 4 Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - 4 Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

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:

```
do {
        while ( TestAndSet (&lock ))
                ;   // do nothing
              //    critical section
        lock = FALSE;
              //      remainder section
 } while (TRUE);
```

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:

```
do {
        key = TRUE;
        while ( key == TRUE)
               Swap (&lock, &key );
                      //    critical section
         lock = FALSE;
                 //      remainder section
 } while (TRUE);
```

Bounded-waiting Mutual Exclusion with TestandSet()

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;
        // critical section
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;
        // remainder section
} while (TRUE);
```

Semaphore

- Synchronization tool that does not require busy waiting (a process repeatedly checks to see if a condition is true and so consumes CPU time without doing useful work)
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

  -
    ```
    wait (S) {
        while S <= 0
            ; // no-op
          S--;
     }
    ```

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

Semaphore 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**
- Provides mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {
   wait (mutex);
         // Critical Section
     signal (mutex);
        // remainder section
} while (TRUE);
```

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
    - 4 But implementation code is short
    - 4 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

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – 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(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```

- Implementation of signal:

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

```
        P0                          P1
       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
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem (Producer Consumer Problem)

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

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do  {

            //   produce an item in nextp

        wait (empty);
        wait (mutex);
            //  add the item to the  buffer
         signal (mutex);
         signal (full);
    } while (TRUE);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
        wait (full);
        wait (mutex);
                // remove an item from  buffer to nextc
        signal (mutex);
        signal (empty);

                // consume the item in nextc

} while (TRUE);
```

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
- Several variations of how readers and writers are treated – all involve priorities
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
          wait (wrt) ;

               //    writing is performed

          signal (wrt) ;
} while (TRUE);
```

Readers-Writers Problem (Cont.)

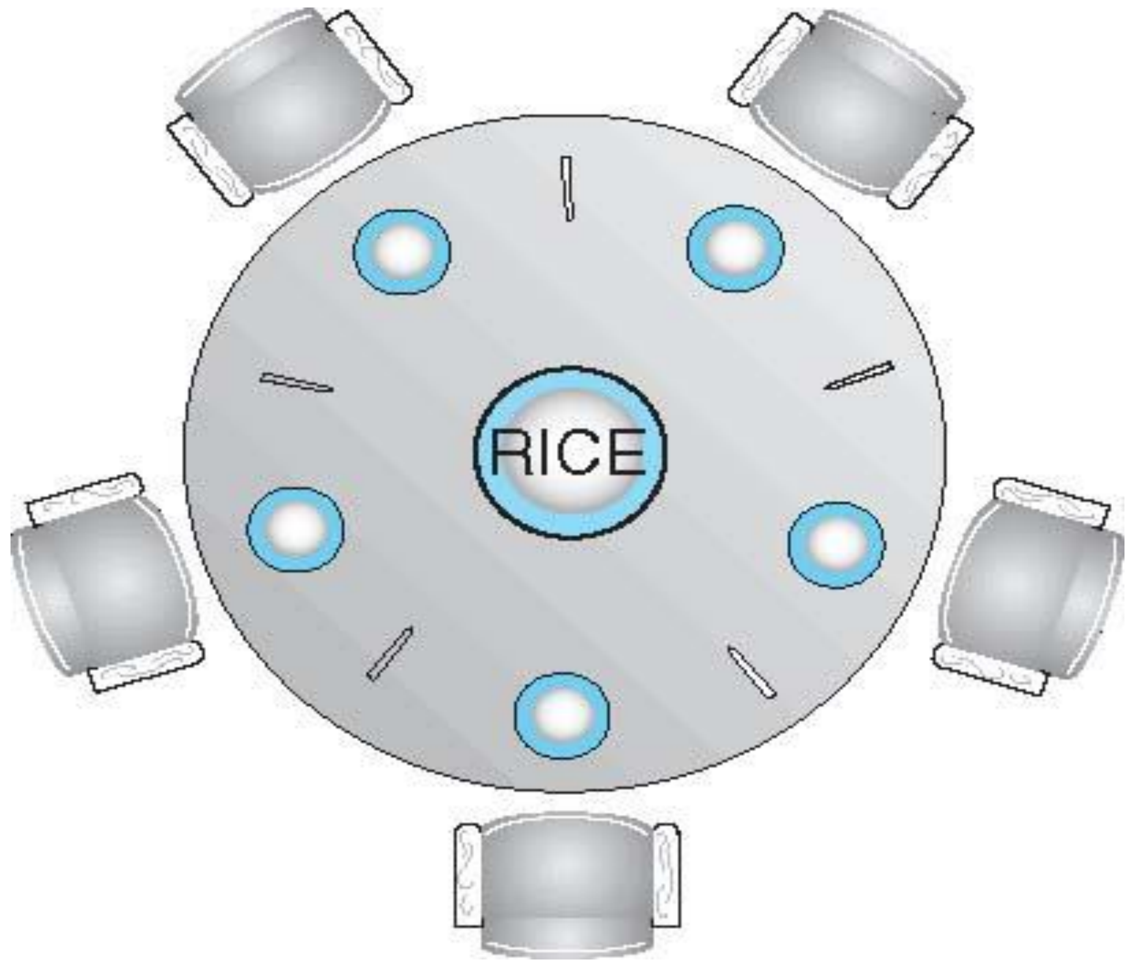
- The structure of a reader process

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
          wait (wrt) ;
        signal (mutex)
                    // reading is performed
        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
          signal (wrt) ;
        signal (mutex) ;
    } while (TRUE);
```

Readers-Writers Problem Variations

- *First* variation – no reader kept waiting unless writer has permission to use shared object
- *Second* variation – once writer is ready, it performs write asap
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem



Five Philosophers on a table; Eat and think alternately. Need 2 chopsticks to eat

- Philosophers spend their lives thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - 4 Bowl of rice (data set)
    - 4 Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

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

- What is the problem with this algorithm?

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation

Monitors

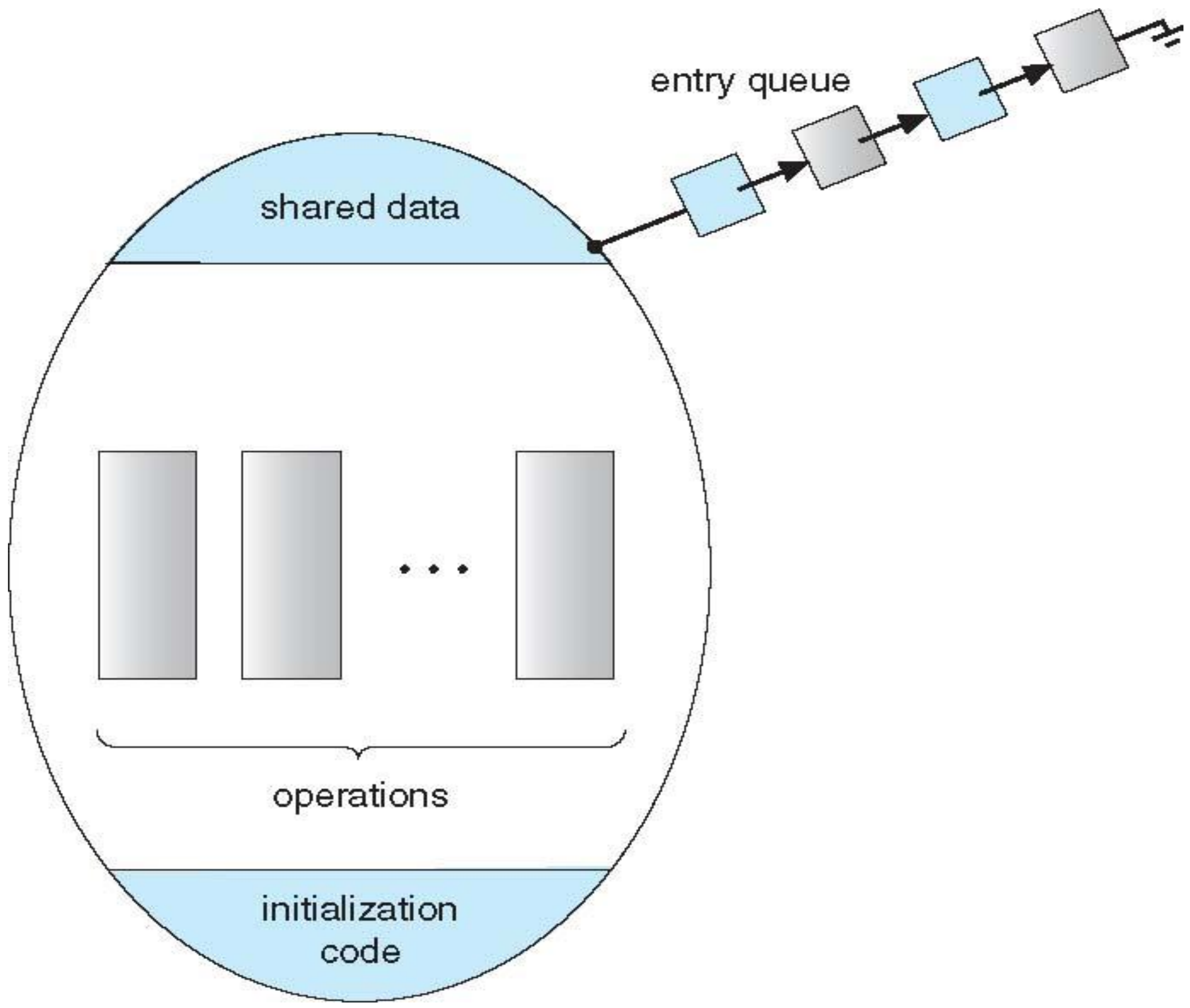
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

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

 procedure Pn (…) {……}

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

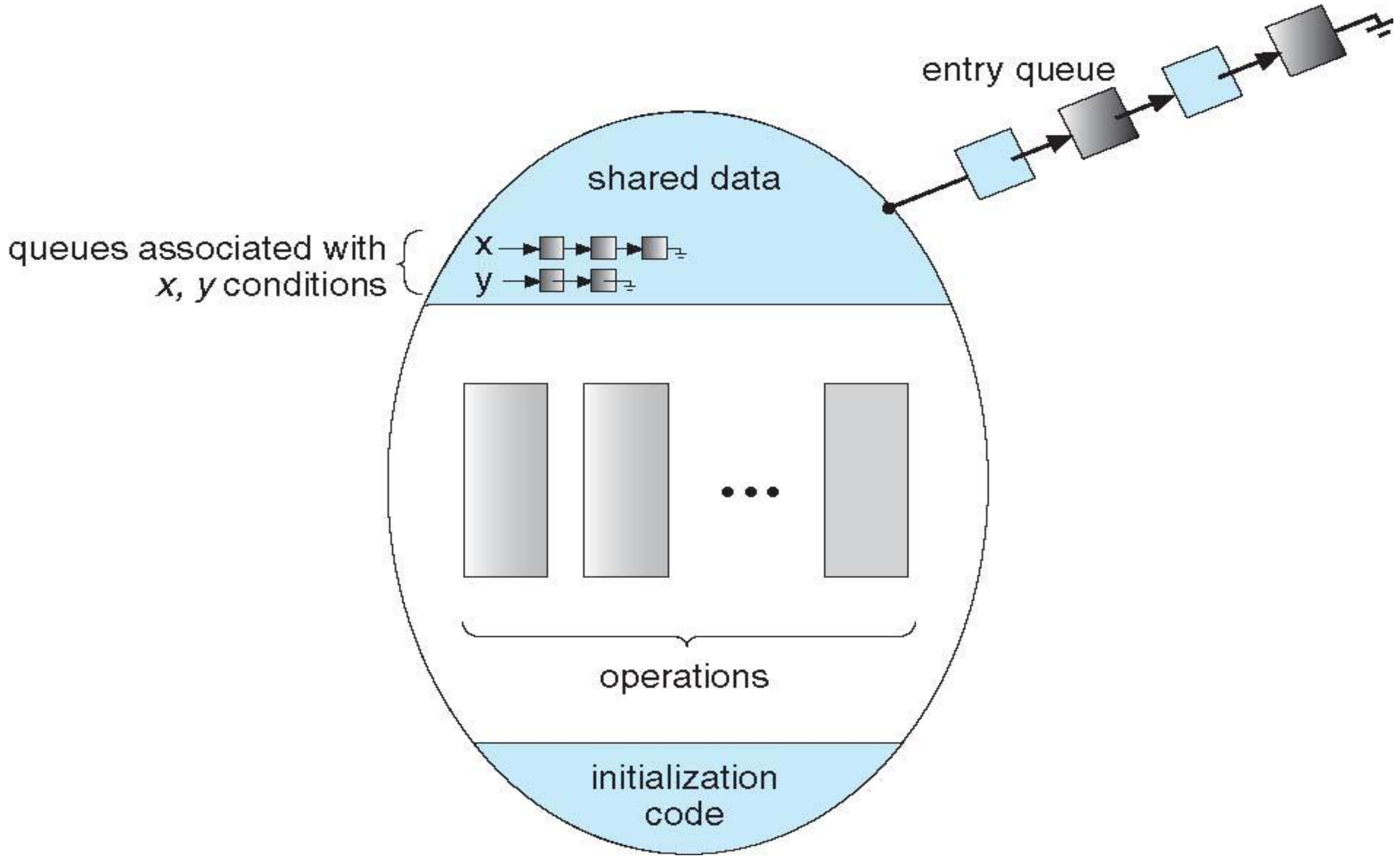
Schematic view of a Monitor



Condition Variables

- condition x, y;
- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended until x.signal ()
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()
    - 4 If no x.wait () on the variable, then it has no effect on the variable

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes x.signal (), with Q in x.wait () state, what should happen next?
  - If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q leaves monitor or waits for another condition.
  - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed

Solution to Dining Philosophers

```
monitor DiningPhilosophers
   {
      enum { THINKING; HUNGRY, EATING) state [5] ;
      condition self [5];

      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:

```
        DiningPhilosophers.pickup (i);

             EAT

         DiningPhilosophers.putdown (i);
```

- No deadlock, but starvation is possible

Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() executed, which should be resumed?

- FCFS frequently not adequate

- **conditional-wait** construct of the form x.wait(c)
  - Where c is **priority number**
  - Process with lowest number (highest priority) is scheduled next

End of Chapter 6