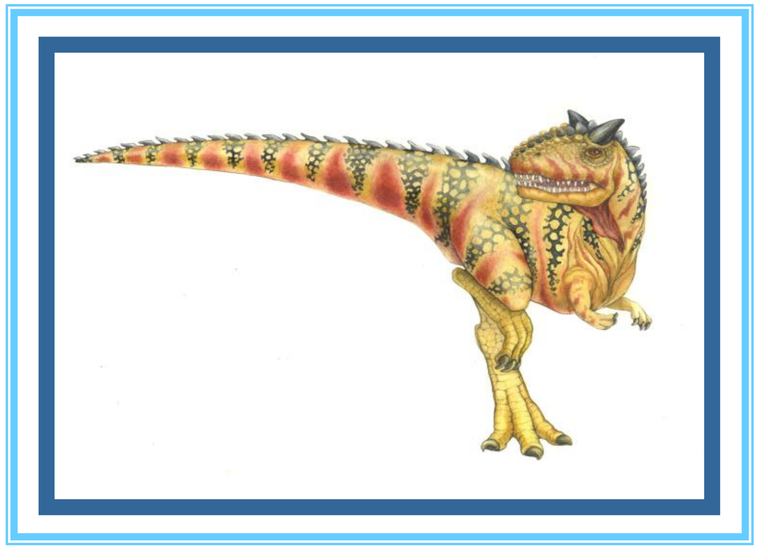
Chapter 6: Process Synchronization



Module 6: Process Synchronization

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

Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity

Background

- **Process Synchronization** is a technique which is used to coordinate the process that use shared Data.
- **Independent Process –**
  The process that does not affect or is affected by the other process while its execution then the process is called Independent Process. Example The process that does not share any shared variable, database, files, etc.
- **Cooperating Process –**
  The process that affect or is affected by the other process while execution, is called a Cooperating Process. Example The process that share file, variable, database, etc are the Cooperating Process
- 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 the buffer. We can do so by having an integer **count** that keeps track of the number of elements in the buffer. Initially, count is set to 0. It is incremented by the producer after it produces a new item into the buffer and is decremented by the consumer after it consumes the item in the 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 $\{p_0, p_1, \dots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
- When one process is executing in its critical section, no other process is to be allowed to execute in critical section. That is, no two processes are executing in their critical sections at the same time. 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**
- Especially challenging with preemptive kernels

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 some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.(decision must be taken within a finite time)
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.

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!

Peterson's Solution-Algorithm for Process P_i

Peterson's solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

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

Algorithm for Process P_j

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

- Provable that

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

Disadv:

Holds good only for 2 processes

Dekker's algorithm

```
//flag[] is boolean array; and turn is an integer
flag[0] = false
flag[1] = false
turn    = 0   // or 1
```

```
P0:
   flag[0] = true;
   while (flag[1] == true) {
      if (turn ≠ 0) {
         flag[0] = false;
         while (turn ≠ 0) {
           // busy wait
         }
         flag[0] = true;
      }
   }

   // critical section
   ...
   turn    = 1;
   flag[0] = false;
   // remainder section
```

```
P1:
   flag[1] = true;
   while (flag[0] == true) {
      if (turn ≠ 1) {
         flag[1] = false;
         while (turn ≠ 1) {
           // busy wait
         }
         flag[1] = true;
      }
   }

   // critical section
   ...
   turn    = 0;
   flag[1] = false;
   // remainder section
```

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:

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

          //    critical section

    lock = FALSE;

          //      remainder section

} while (TRUE);
```

- Definition:

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

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

- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() →P (from the Dutch proberen, "to test");
  - V()→ (from verhogen, "to increment").
- 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**
- Mutual-exclusion implementation with **Binary** semaphore
- Semaphore mutex;   // initialized to 1

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

- Case 1: To control access to a given resource

- Semaphore is initialized to the number of resources
- Semaphore Count =0;all resources are utilized

- Case 2 : process synchronization
- (s2 is executed only after s1 completes)
- synch=0

```
S1;
signal(synch);
```

Disadvantage: Busy waiting thus wastes CPU cycles

```
wait(synch);
S2;
```

- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.
- This type of semaphore is also called spinlock a because the process "spins" while waiting for the lock.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.
- (Spinlocks do have an advantage in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time)
- To overcome the need for busy waiting, we can modify the definition of the wait() and signal() semaphore operations.
- rather than engaging in busy waiting, the process can *block* itself. The block operation places a process into a waiting queue associated with the semaphore

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

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

- 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

semaphore S = 1;

```
do
{
        wait(S);
              Critical Section
        signal(S);
              Remainder Section
} while(1)
```

value = 1

Ready Queue


```
wait(S)
{
        value--;
        if(value<0)
        {
                add process in LIST;
                block();
        }
}
```

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

Semaphore Implementation with no Busy waiting (Cont.)

- Synchronization tool that does not require busy waiting

- Implementation of wait:

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

Entry

Semaphore value –ve indicates the processes waiting to acquire that semaphore

- Implementation of signal:

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

Put the processes from the blocked list to ready queue… not into CS

Exit

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

- L and H share/modify kernel data(CS)
- Assume we have three processes, *L*, *M*, and *H*, whose priorities follow the order *L* < *M* < *H*. Assume that process *H* requires resource *R*, which is currently being accessed by process *L*. Ordinarily, process *H* would wait for *L* to finish using resource *R*.
- *M* becomes runnable, thereby preempting process
- a process with a lower priority—process *M*—has affected how long process *H* must wait for *L* to relinquish resource *R*.
- This problem is known as **priority inversion**.
- priority-inheritance protocol would allow process *L* to temporarily inherit the priority of process *H*, thereby preventing process *M* from preempting its execution. When process *L* had finished using resource *R*, it would relinquish its inherited priority from *H* and assume its original priority. Because resource *R* would now be available, process *H*—not *M*—would run next.

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

- *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);
```

- *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 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
- 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
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

Readers-Writers Problem

- **semaphore wrt functions as a mutual-exclusion semaphore for the writers.**
- The structure of a writer process

- Semaphore mutex initialized to 1
- Semaphore wrt initialized to 1
- Integer readcount initialized to 0

```
do {
        wait (wrt) ;

            //    writing is performed

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

Readers-Writers Problem (Cont.)

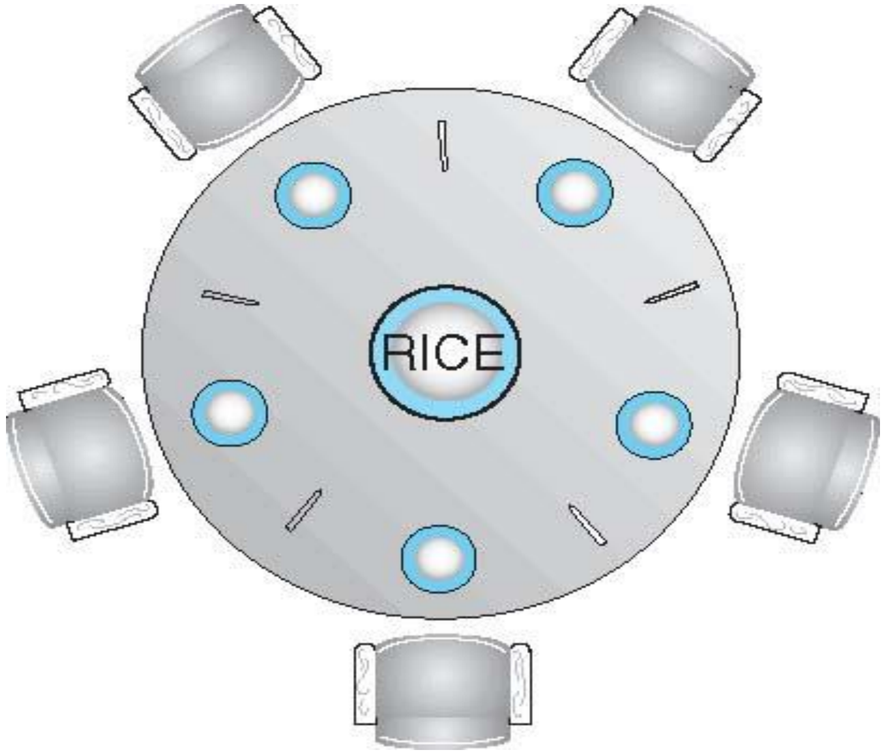
- The structure of a reader process

```
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);
```

- Semaphore mutex initialized to 1
- Semaphore wrt initialized to 1
- Integer readcount initialized to 0

wrt is a binary semaphore

Dining-Philosophers Problem



- 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
    - Bowl of rice (data set)
    - 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?
- If all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of chopstick will now be equal to 0.When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are listed next.

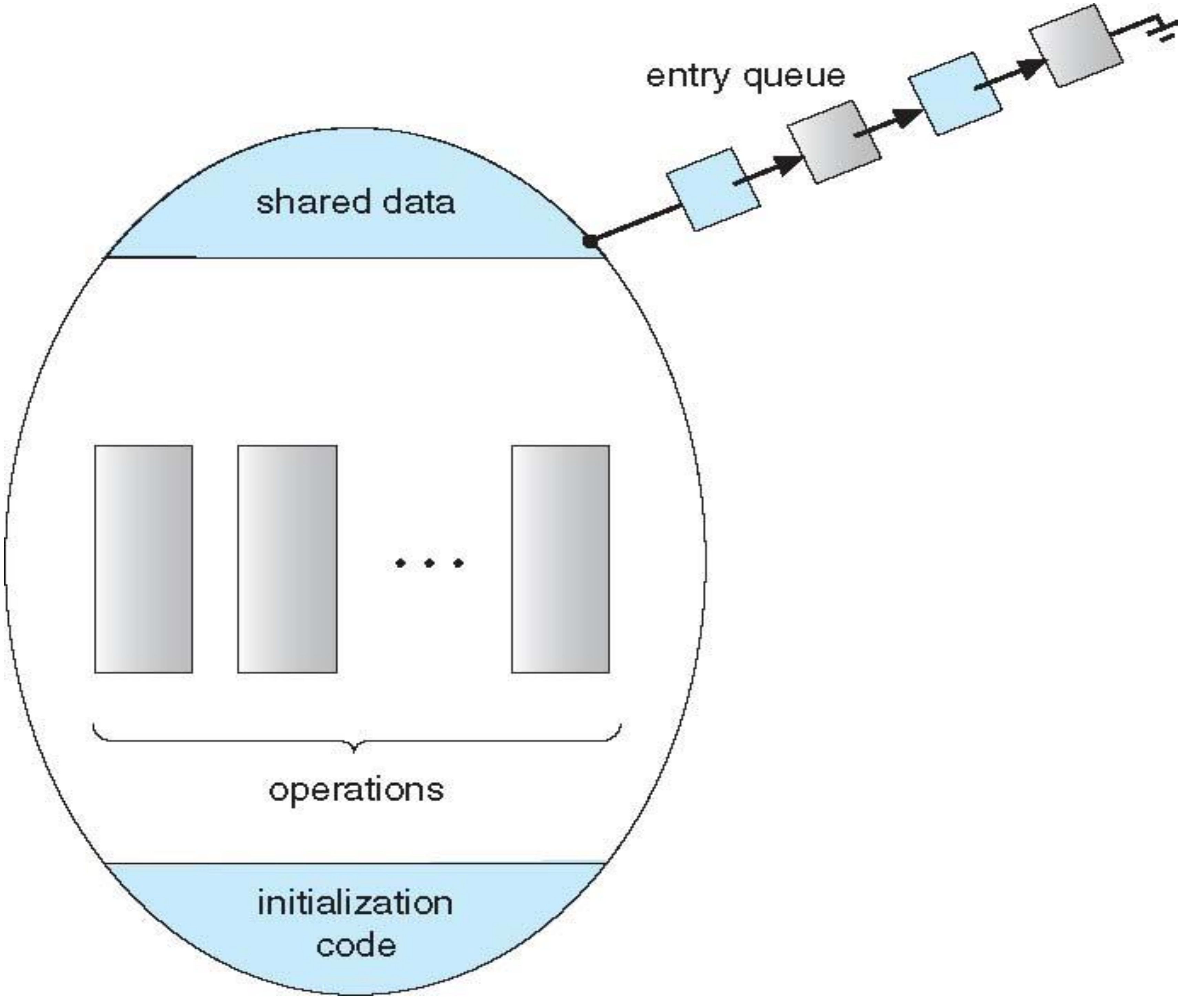
- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available .
- Allow odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

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

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

    procedure Pn (…) {……}

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

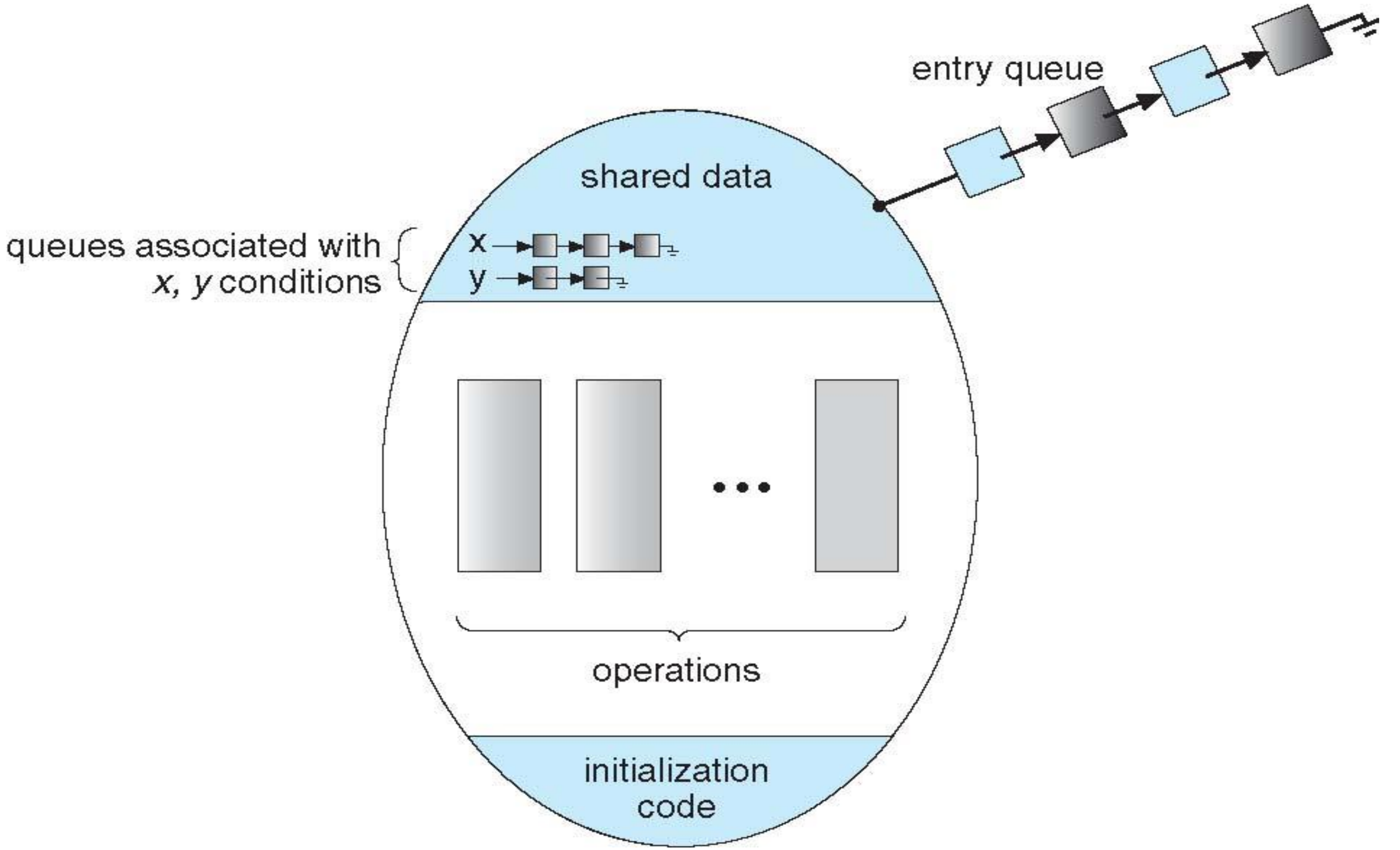
Schematic view of a Monitor

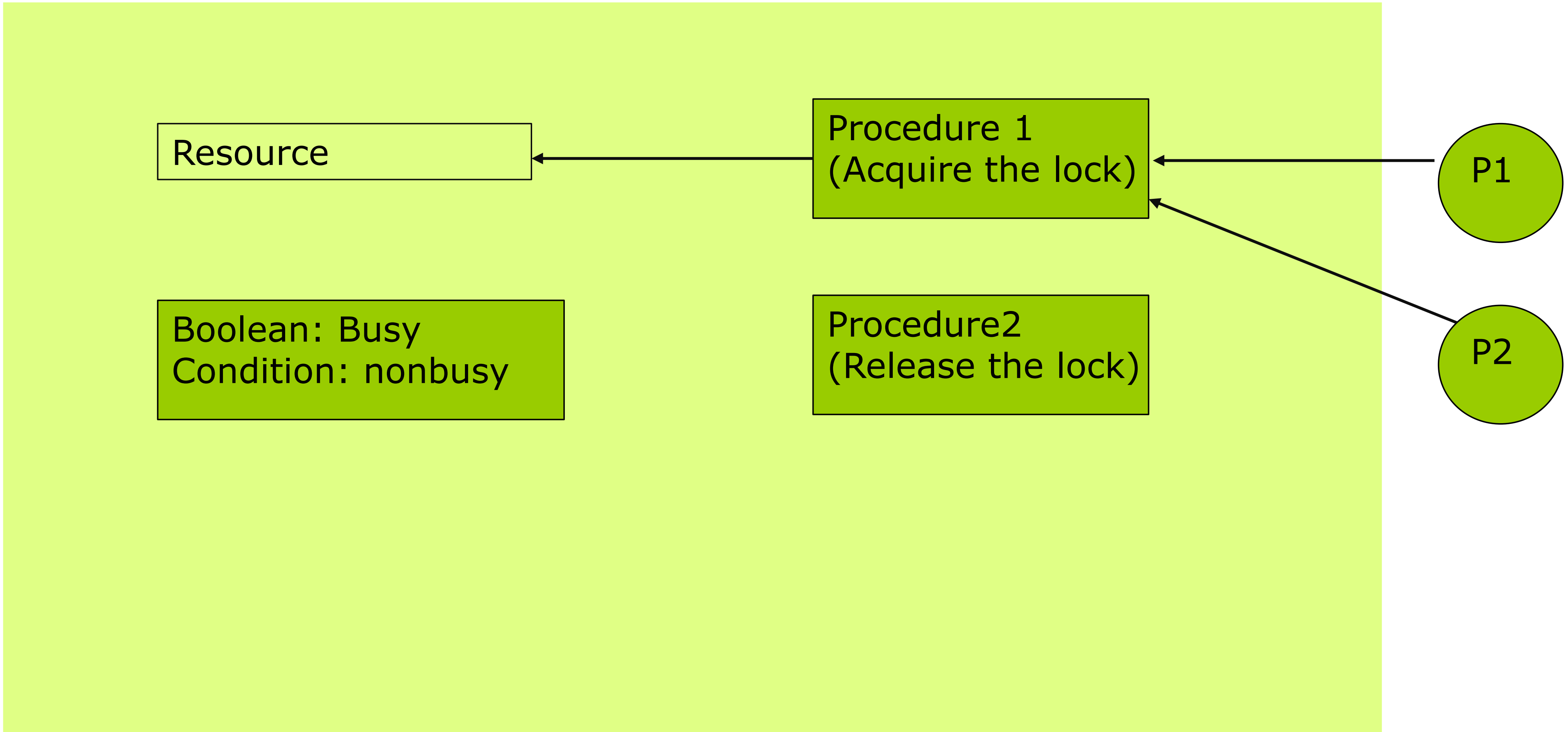


Condition Variables

- 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 ()

Monitor with Condition Variables





if Busy then nonbusy.wait()
Else Busy=TRUE

Busy=FALSE
nonbusy.signal()

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
  - Implemented in other languages including Mesa, C#, Java

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);
     }

    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;
     }
}
```

- a philosopher moves to his/her eating state only if both neighbors are not in their eating states
- if one of my neighbors is eating, and I'm hungry, ask them to signal() me when they're done

- *"solution ensures that no two neighbors are eating simultaneously and that no deadlock will occur "*

End of Chapter 6