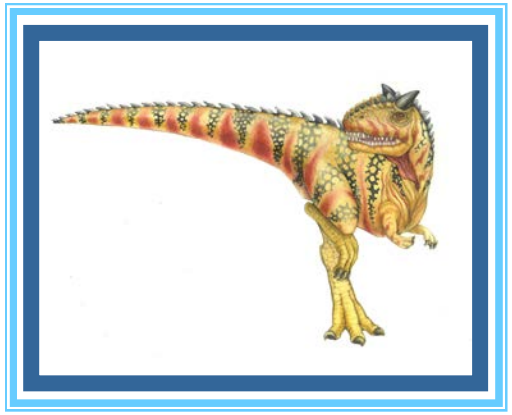
Chapter 5: Process Synchronization



Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

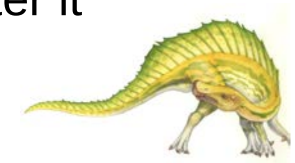
Objectives

- To present the concept of process synchronization.
- 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 examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- In this chapter, what we discuss for “processes” almost always applies to “threads” as well
  - Can change “processes” to “threads”
- Illustration of the problem:

  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 **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** 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 in next produced */

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

Consumer

```
while (true) {
       while (counter == 0)
              ; /* do nothing */
       next_consumed = buffer[out];
       out = (out + 1) % BUFFER_SIZE;
        counter--;
       /* consume the item in next consumed */
}
```

Race Condition

- **`counter++`** could be implemented as

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

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = 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 – 1`** **`register2 = register2 – 1`**	{register2 = 4}
S4: producer execute **`counter = register1`**	{counter = **6**}
S5: consumer execute **`counter = register2`**	{counter = **4**}

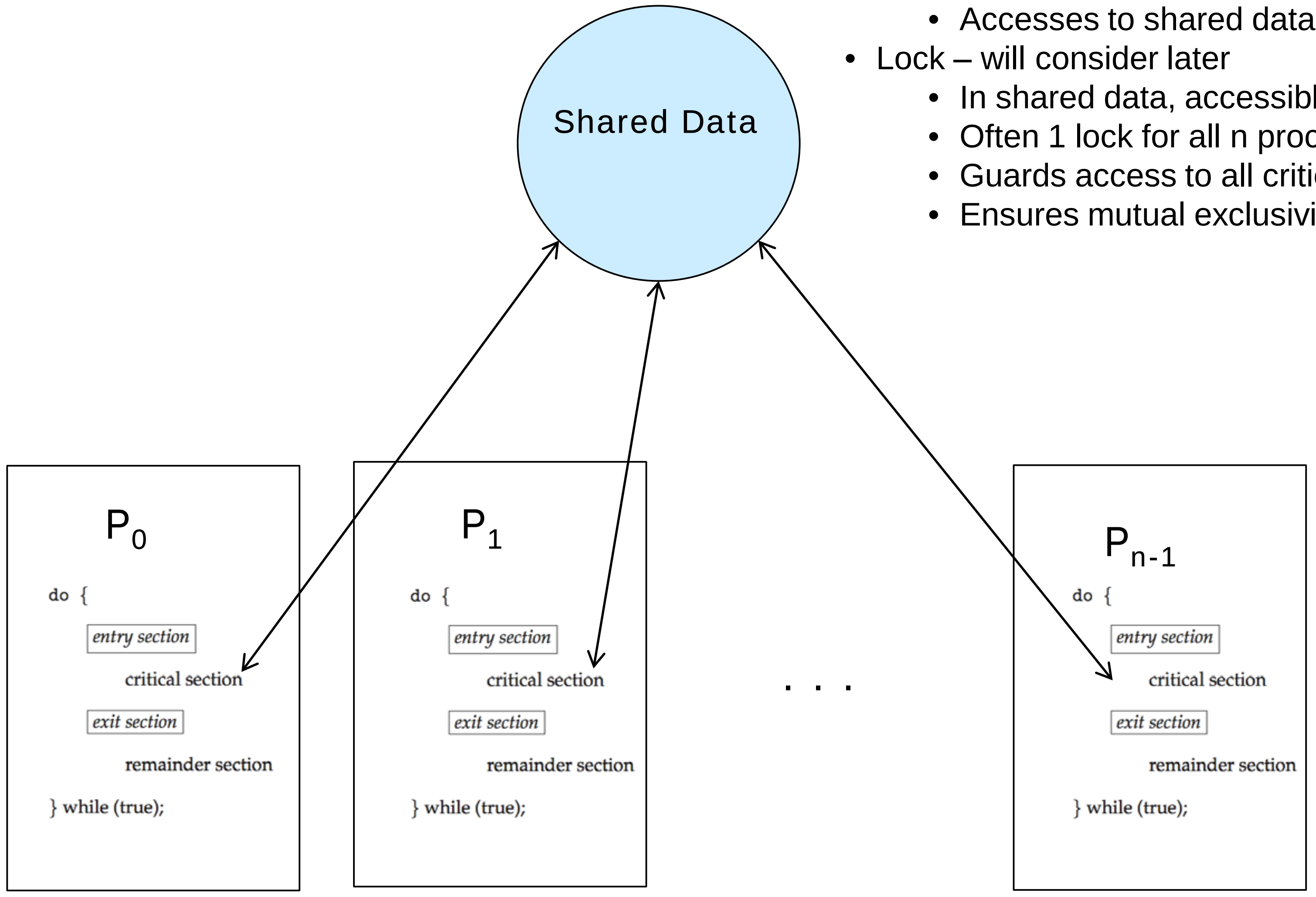
- **Race condition** – a situation when:
  - several processes access the same data concurrently
  - the outcome of the execution depends on the order of the accesses

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 is allowed to 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**

```
do {

     entry section

          critical section

     exit section

          remainder section

} while (true);
```

Diagram of Critical Section Problem



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

- Each process P_i has its own critical section
  - Accesses to shared data only in CS
- Lock – will consider later
  - In shared data, accessible by all
  - Often 1 lock for all n processes
  - Guards access to all critical sections
  - Ensures mutual exclusivity

Requirements to Critical-Section Problem

- Assume that each process executes at a nonzero speed
- No assumption concerning **relative speed** of the ***n*** processes

1. **Mutual Exclusion**
   - Only one process can be in the critical section at a time –
     - otherwise what critical section?
2. **Progress**
   - **Intuition:** No process is forced to wait for an available resource –
     - otherwise very wasteful.
   - **Formal:** 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**
   - **Intuition:** No process can wait forever for a resource –
     - otherwise an easy solution: no one gets in
   - **Formal:** 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

Some material taken from http://www.cs.cmu.edu/~gkesden/412-18/fall01/ln/lecture6.html

Types of solutions to CS problem

- Software-based
- Hardware-based

Software-based solution to CS

- Will consider 2-process case first:
  - Processes P_i and P_j
  - Will examine various incorrect and correct solutions next
- Solutions assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
  - In general, this is **not true** for modern architectures
    - Peterson's algorithm (we will consider shortly) **does not work in general**
    - Can work on some machines correctly, but can fail on others
  - But good algorithmic description, allows to understand various issues
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the CS
- The **`flag`** array is used to indicate if a process is ready to enter the CS
  - **`flag[i] = `*`true`*** implies that process P_i is ready!

Algorithm 1

- Case: 2 processes Pi and Pj only
- Shared Variables:
  - **var** *turn*: (0..1);

    initially *turn* = 0;
  - *turn* = *i* ☑ P*i* can enter its critical section
- Process *Pi*

```
while(true) {
        while (turn != i) no-op;
                critical section
        turn = j;
                remainder section
}
```

- Satisfies mutual exclusion, but not progress
  - If it is turn i and if Pi is not in its CS, then if Pj wants to enter its CS, then Pj has to wait – but why wait?: Pi is not in its CS

Algorithm 2

- Shared Variables
  - **var** flag: **array** (0..1) **of** boolean;

    initially: flag[0] = flag[1] = false;
  - flag[i] = true ☑ Pi ready to enter its critical section
- Process *Pi*

```
while(true) {
      flag[i] = true;
      while (flag[j]) no-op;
            critical section
      flag[i]= false;
            remainder section
}
```

- Can block indefinitely… Progress requirement not met.
  - Both can set flag[] to true and then both block busy-waiting

Algorithm 3

- Shared Variables
  - **var** *flag*: **array** (0..1) **of** boolean;
    initially *flag[0]* = *flag[1]* = false;
  - *flag[i]* = *true* ☑ Pi ready to enter its critical section
- Process *Pi*

```
while(true) {
        while (flag[j]) no-op;
        flag[i] = true;
                critical section
        flag[i] = false;
                remainder section
}
```

- Does not satisfy mutual exclusion requirement …
  - (1) Pj flag[j]=false;
  - (2) Pi moves to after while, before flag[i] = true;
  - (3) Pj passes its **while;** sets flag[j] = true; enters its CS
  - (4) Pi sets flag[i] = true; enters its CS => both in their CSes

Algorithm 4: Peterson's solution 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 the three CS requirement are met:
  1. Mutual exclusion is preserved

     P$_i$ enters CS only if:

     either **flag[j] = false** or **turn = i**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

(Lamport's) "Bakery Algorithm"

- Critical section for ***n*** processes
- **Key idea:** Before entering its CS, the process receives a number (=token)
  - Holder of the smallest number enters the CS
  - Like in a bakery, get a number at the entrance, served according to it
- If processes **Pi** and **Pj** receive the same number,
  - if **i ≤ j**, then **Pi** is served first; else **Pj** is served first
  - Will say **Pi** has higher **priority** than **Pj**
- The numbering scheme always generates numbers in non-decreasing order of enumeration;
  - For example: 1,2,3,3,3,3,4,4,5,5

Bakery Algorithm (cont.)

- Notation
  - Lexicographic order
    - `(ticket#, process_id#)`
    - `(a,b) < (c,d), same as: if (a<c or ((a=c) and (b<d))`

- Shared Data

```
var
  choosing: array[0..n-1] of boolean; (initialized to false)
  number: array[0..n-1] of integer; (initialized to 0)
```

Bakery Algorithm (cont.)

Code for process **Pi**

```
while(true) {
     choosing[i] = true;
     number[i] = max(number[0], number[1],…,number[n-1]) +1;
     choosing[i] = false;

     for (int j = 0; j <= n-1; j++)
     {
        // wait until process Pj receives its number (i.e, token)
        while (choosing[j])
               ; // no-op

              // wait until all processes
              // (a) with smaller number, or
              // (b) with the same number but with higher priority
              // finish their work
        while (number[j] != 0 && (number[j] ,j) < (number[i],i))
               ; // no-op
      }

        critical section
      number[i]= 0;
        remainder section
}
```

Synchronization Hardware

- Many systems provide **hardware support** for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
    - That is, without being interrupted
  - Generally too inefficient on multiprocessor systems
    - OSes using this are not broadly scalable
- Modern machines provide special **atomic hardware instructions**
  - **Atomic** = non-interruptible
  - **test_and_set** instruction
    - test memory word and set value
  - **compare_and_swap** instruction
    - swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
while (true) {
    acquire lock
            critical section
    release lock
            remainder section
}
```

test_and_set Instruction

Definition:

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

1. Executed atomically – cannot be interrupted in the middle
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to **true**.

Solution using test_and_set()

- Shared Boolean variable lock
  - initialized lock = **false**
  - unlocked initially

- Solution:

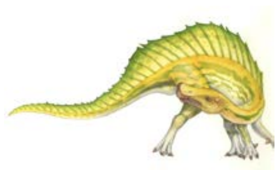
```
while (true) {
    while (test_and_set(&lock))
        ; // wait until lock becomes equal false
    // here lock=true
            /* critical section */
    lock = false;
            /* remainder section */
}
```

- Bounded waiting is **not** satisfied
  - Other processes might keep acquiring the lock first

Bounded-waiting Mutual Exclusion with test_and_set

- The common data structures:
  - boolean waiting[n]; // initialized to false
  - boolean lock; // initialized to false

```
while (true) {
        waiting[i] = true; //  wants the lock
        key = true;
        while (waiting[i] && key)
        key = test_and_set(&lock); // trying to get the lock
    // got the lock here
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;

    while ((j != i) && !waiting[j]) // find the next in sequence Pj waiting for the lock
        j = (j + 1) % n;

    if (j == i) lock = false; // no one is waiting for the lock, simply release it
    else         waiting[j] = false; // let Pj go into its CS, "lock" is kept locked
    /* remainder section */
}
```



- Satisfies all the three CS requirements, less fair than Lamport's Bakery algorithm

compare_and_swap Instruction

Definition:

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

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter "value"
3. Set the variable value to new_value but only if value ==expected
   - The swap takes place only under this condition

Solution using compare_and_swap

- Shared integer lock
  - initialized to lock=0; // unlocked
- Solution:

```
while (true) {
  while (compare_and_swap(&lock, 0, 1) != 0)
    ; // wait for lock to be 0

    // here lock = 1
    /* critical section */
    lock = 0;
    /* remainder section */
}
```

- Bounded waiting is **not** satisfied

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
- Protect a critical section by:
  - first **`acquire()`** a lock
  - then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

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

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

-
  ```
  while (true) {
     acquire lock
        critical section
     release lock
        remainder section
  }
  ```

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
  - Originally called **`P()`** and **`V()`**-- important to remember, used often
    - **`P()`** (wait) – from Dutch proberen: "to test"
    - **`V()`** (signal) – from Dutch verhogen: "to increment"
- Definition of the **`wait() operation`**

```
wait(S) {
    while (S <= 0)
        ; // Notice, busy waits, spending CPU cycles, not (always) good
    S--;
}
```

- Definition of the **`signal() operation`**

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

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
  - Can control access to resource that has a finite number of instances
  - Initialized to the number of resources available
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider ***P***

Consider $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ that require $\boldsymbol{S_1}$ to happen before $\boldsymbol{S_2}$

Create a semaphore "**synch**" initialized to 0

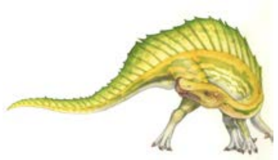
```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Implementing a counting semaphore via binary

- Can implement a counting semaphore ***S*** via binary semaphores
- Data Structures

```
int S = n;           // semaphore count
mutex S_guard = 1; // initial value 1 (FREE)
mutex delay = 0;   // for delaying other processes
```

Solution adopted from Jim Mooney of WVU.



Implementing S

```
wait()
{
    wait(S_guard);
    S --;

    if (S <= -1) {        // check if someone's already waiting on this sem
        signal(S_guard);  // relesase S_guard
        wait(delay);      // join the waiting queue
    }
    else signal(S_guard);
}

signal()
{
    wait(S_guard);
    S = ++;

    if (S < 0) signal(delay);

    signal(S_guard);
}
```

Semaphore Implementation

- Operations **`wait()`** and **`signal()`** must be executed atomically
- Thus, the implementation becomes the critical section problem
  - **`wait`** and **`signal`** code are placed in the critical section
- Hence, can now have **busy waiting** in critical section implementation
  - If critical section rarely occupied
    - Little busy waiting
  - But, applications may spend lots of time in critical sections
    - Hence, the busy-waiting approach is not a good solution

No Busy Waiting Implementation of Semaphore

- A **waiting queue** is associated with each semaphore
  - It stores the processes waiting on the semaphore

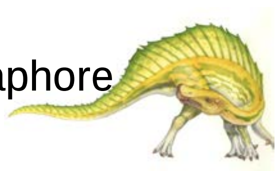
-
  ```
  typedef struct{
     int value;
     struct process *list; // waiting queue
   } semaphore;
  ```

- 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

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block(); //suspends self, sleeps, avoids CPU cycles
   }
}

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

- S->value can be negative
  - |S->value| is then the number of processes waiting on the semaphore
  - It is never negative in classical definition



Deadlock and Starvation

- **Synchronization problems**
- **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)**
  - Related to deadlocks (but different)
  - Occurs when a process waits indefinitely in the semaphore queue
  - For example, assume a LIFO semaphore queue
    - Processed pushed first into it might not get a chance to execute
- **Priority Inversion**
  - A situation when a higher-priority process needs to wait for a lower-priority process that holds a lock

Classical Problems of Synchronization

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

The Bounded-Buffer Problem

- Producer and Consumer processes
- Buffer pool consists of ***n*** buffers
  - each can hold one item
- Shared data structures:

```
int n;
semaphore mutex = 1;   // Guards the access to the buffer pool
semaphore empty = n;   // Counts the number of empty buffers
semaphore full = 0;    // Counts the number of full buffers
```

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true) {
        ...
         /* produce an item in next_produced */
        ...
     wait(empty); // suspend self if the buffer is full
     wait(mutex);  // get access to the buffer
         ...
        /* add next produced to the buffer */
         ...
     signal(mutex); // release the lock to the buffer
     signal(full);  //
 }
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
      wait(full); //=P() suspend self if the buffer is empty
      wait(mutex); // get access to the buffer
         ...
      /* remove an item from buffer to next_consumed */
         ...
      signal(mutex); // V() release the lock to the buffer
      signal(empty);
         ...
      /* consume the item in next consumed */
         ...
 }
```

Discussion

- A symmetry
  - Producer does: `P(empty), V(full)`
  - Consumer does: `P(full), V(empty)`
  - Producer producing full buffers for consumer
  - Consumer producing empty buffers for the producer
- Is order of P's important?
  - Yes! Can cause deadlock
- Is order of V's important?
  - No, except that it might affect scheduling efficiency

The 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 (shared data) at the same time
  - Only one single writer can access shared data at the same time
    - Readers cannot read during this time
    - Other writers cannot write during this time

Readers-Writers Problem (Cont.)

- Shared Data:
  - Data set

```
semaphore rw_mutex = 1;  // mutual exclusion for writers
semaphore mutex = 1;        // guards read_count
int read_count = 0;         // #processes currently reading the object
```

- The structure of a writer process

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

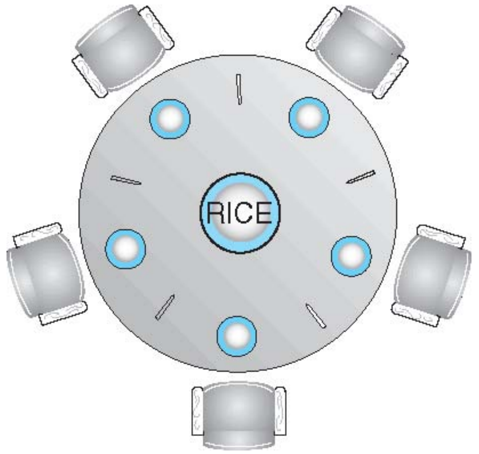
Readers-Writers Problem (Cont.)

- The structure of a reader process

```
while (true) {
       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);
}
```

- Writers can starve in this solution
  - How would you design a solution where writers don't starve?

The Dining-Philosophers Problem



- Philosophers spend their lives alternating: thinking and eating
- Occasionally try to pick up 2 chopsticks (left and right) to eat from bowl
  - One chopstick at a time
  - Need both chopsticks to eat, then release both when done
  - Problem: not enough chopsticks for all
    - N philosophes and N chopsticks (not 2N)
- The case of 5 philosophers (and 5 chopsticks only)
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1 (free)

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
    while (true) {
        wait (chopstick[i]);    // wait to get the left stick
        wait (chopstick[(i + 1) % 5]);  // get the right

        //  eat

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

       //  think
    };
```

- What is the problem with this algorithm?
  - A deadlock is possible!

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling possibilities:
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the chopsticks only if both are available
    - Picking must be done in a critical section
  - Use an asymmetric solution:
    - an odd-numbered philosopher
      - picks up first the left and then right chopstick
    - even-numbered philosopher
      - picks up first the right and then left chopstick

Problems with Semaphores

- Incorrect use of semaphore operations:
  - Can occur relatively easily and be difficult to detect
    - Timing errors – could occur only under certain circumstances and won't occur otherwise
      - Run a program and it crashes, run it again and it doesn't
    - Can be more complex to debug
  - Wrong order: signal (mutex) …. wait (mutex)
  - Wrong calls: wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

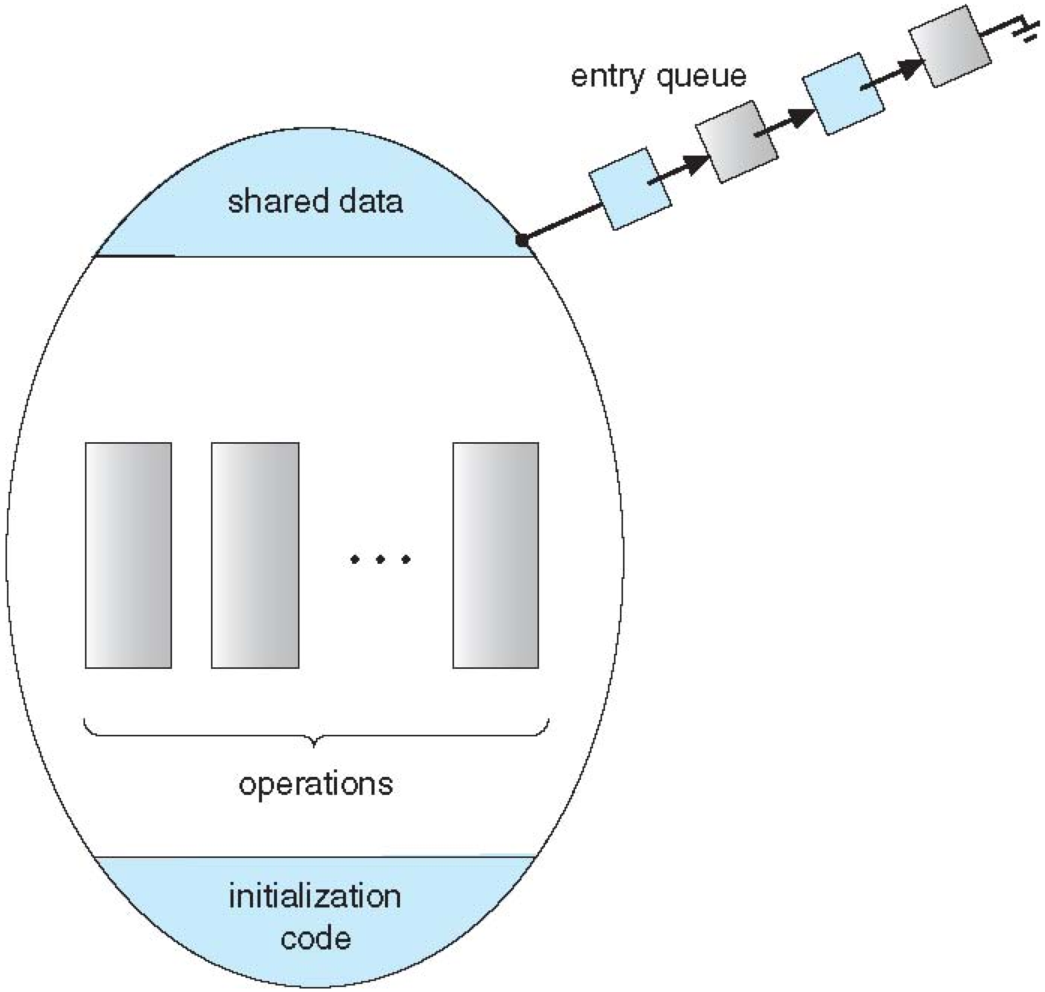
- Deadlock and starvation are possible
- How to deal with this?
  - Researchers developed high-level language constructs
  - Monitor type – one such construct

Monitors

```
monitor monitor-name
{
  // shared variable declarations
  function P1 (…) { … }
  …
  function Pn (…) {……}
  initialization_code (…) { … }
}
```

- Recall that: an *Abstract data type (ADT)* -- encapsulates data
  - internal variables only accessible by code within the procedure
- A monitor type – is an ADT that provides a convenient and effective mechanism for process synchronization
  - For several concurrent processes
  - Encapsulation
    - Local variables accessed only via local functions
    - Local functions access only local vars and params
- Only one process at a time may be active within the monitor
  - A lock guards shared data
  - Hence, the programmer does not need to code this constraint

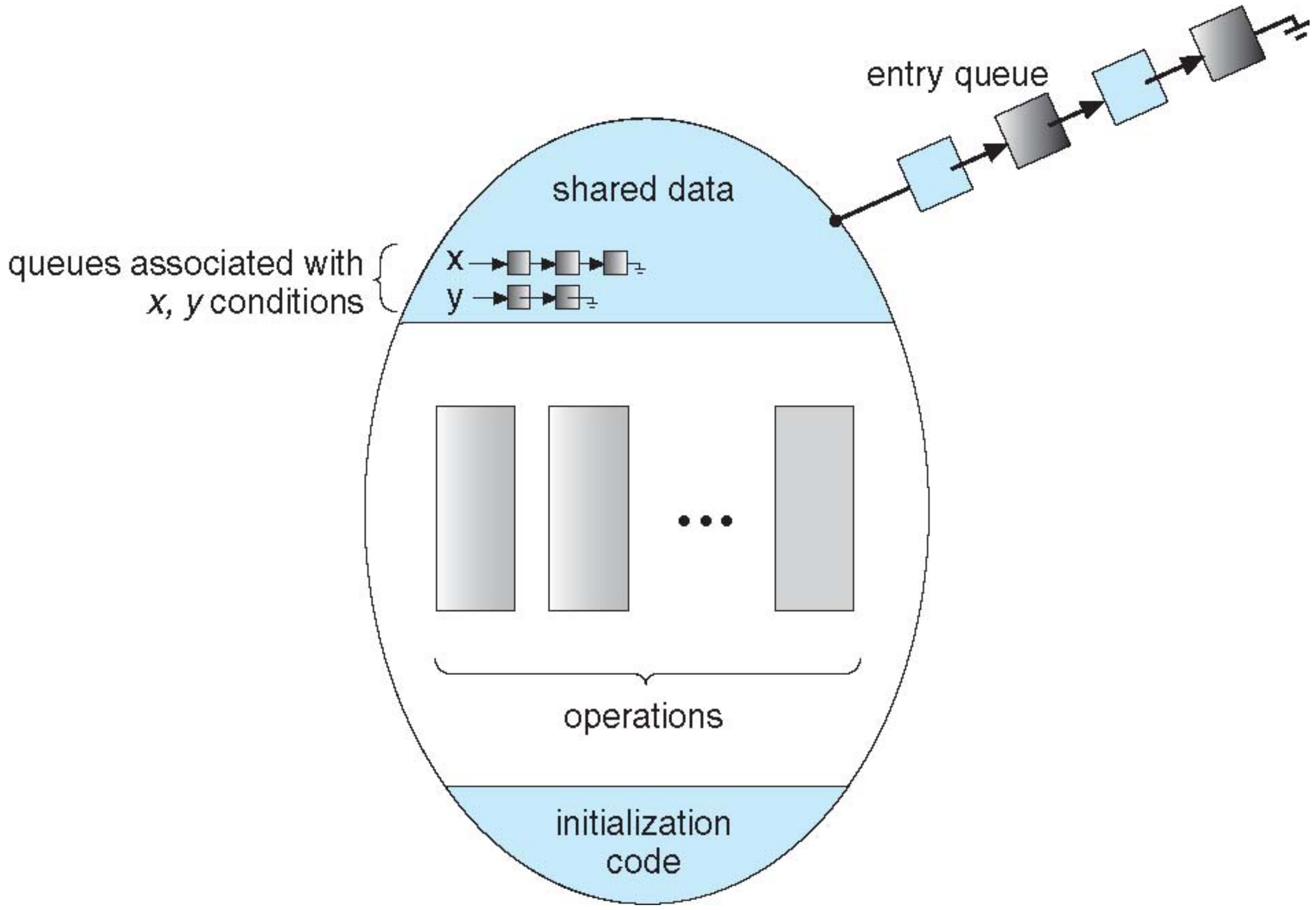
Schematic view of a Monitor



Condition Variables

- Monitors not powerful enough to model some synchr. schemes
  - Hence, additional synchr. mechanisms are added:
  - Provided by the **condition** construct
- **`condition x, y;`**
- Two operations are allowed on a condition variable **x**:
  - **`x.wait()`**
    - A process that invokes the operation is suspended (sleeps)
      - until **`x.signal()`**`is called`
    - Lets other processes enter the monitor
      - releases the lock to shared data, atomically with sleep
  - **`x.signal()`**
    - Resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** was called, then **`x.signal()`**has no effect **x**

Monitor with Condition Variables



Condition Variables Choices

- Issues with monitors: assume
  - Process P invokes **`x.signal(),`** and
  - Process Q is suspended in **`x.wait()`**
  - Who proceeds next?
    - Both Q and P cannot execute in parallel
    - Because they are within a monitor
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide

Monitor Solution to Dining Philosophers

- Each philosopher i invokes the operations
  - **`pickup()`** and **`putdown()`** in an infinite loop:

```
 DiningPhilosophers.pickup(i);

 // EAT

DiningPhilosophers.putdown(i);
```

- No deadlocks, but starvation is possible

Solution to Dining Philosophers (Cont.)

```
monitor DiningPhilosophers
{
   //-- shared data, local to the  monitor --
   enum {THINKING; HUNGRY, EATING) state [5];
   condition self[5]; // for suspending self when chopsticks are not available



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

Solution to Dining Philosophers (cont.)

```
//-- local functions --
void pickup (int i) {
        state[i] = HUNGRY;
        test(i);                              // try to start eating.
        if (state[i] != EATING)  // if cannot start eating -- wait
              self[i].wait;      // suspend self -- until a direct neighbor bumps you
}

// test(i) will set state of i to EATING only if:
// (a) i is hungry and (b) its left & right neighbors are not eating
void test (int i)
{
    if ((state[i] == HUNGRY) && // access to shared data is protected automatically
         (state[(i + 1) % 5] != EATING) &&
         (state[(i + 4) % 5] != EATING) )
         {
            state[i] = EATING;
            self[i].signal(); //release i if it is waiting
         }
}
```

Solution to Dining Philosophers (cont.)

```
    void putdown (int i) {
            state[i] = THINKING;
              // test the left/right neighbors (who perhaps are waiting for your chopsticks)
             test((i + 1) % 5);
             test((i + 4) % 5);
    }
}
```

- Recall what test(i) does:

```
// test(i) will set state of i to EATING only if:
// (a) i is hungry and (b) the left & right neighbors are not eating
void test (int i)
{
    if ((state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) &&
        (state[(i + 4) % 5] != EATING) )
        {
           state[i] = EATING;
           self[i].signal(); //release i if it is waiting
        }
}
```

End of Chapter 5