Shared Memory Model

Alessio Vecchio
alessio.vecchio@unipi.it
Dip. di Ingegneria dell'Informazione
Università di Pisa

Based on original slides by Silberschatz, Galvin, and Gagne
Operating System Concepts, IX edition

Overview

- The Critical-Section Problem
- Software Solutions
- Synchronization Hardware
- Semaphores
- Monitors
- Synchronization Examples

Overview

- **The Critical-Section Problem**
- Software Solutions
- Synchronization Hardware
- Semaphores
- Monitors
- Synchronization Examples

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

Producer-Consumer Problem

- The *Producer* process produces data that must processed by the *Consumer* process
- The inter-process communication occurs through a shared buffer (shared memory)
- Bounded Buffer Size
  - The Producer process cannot insert a new item if the buffer is full
  - The Consumer process cannot extract an item if the buffer is empty

Producer-Consumer Problem

- Shared data

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

Producer-Consumer Problem

- Producer process

```
item nextProduced;

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

Producer-Consumer Problem

- Consumer process

```
item nextConsumed;

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

Producer-Consumer Problem

- The statements

  counter++;
  counter--;

  must be performed atomically.

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

Producer-Consumer Problem

- The statement “counter++” may be implemented in machine language as:

  register1 = counter
  register1 = register1 – 1
  counter = register1

- The statement “counter—” may be implemented as:

  register2 = counter
  register2 = register2 – 1
  counter = register2

Producer-Consumer Problem

- 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.

Race Condition

- 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 **count** may be either 4 or 6, where the correct result should be 5.

Race Condition

- Race condition
  - The situation where several processes access and manipulate shared data concurrently.
  - The final value of the shared data depends upon how instructions are interleaved.

- Show example about balance and num.Ops.

- To prevent race conditions, concurrent processes must be synchronized.

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**

Solution to Critical-Section Problem

- **1 - Mutual Exclusion**
  - If process Pi 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.

General Process Structure

- General structure of process P_i

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

Possible Solutions

- Software approaches
- Hardware solutions
  - Interrupt disabling
  - Special machine instructions
- Operating System Support
  - Semaphores
- Programming language Support
  - Monitor
  - …

Overview

- The Critical-Section Problem
- **Software Solutions**
- Synchronization Hardware
- Semaphores
- Monitors
- Synchronization Examples

A Software Solution

```
Boolean lock=FALSE;
Process Pi {
        do {
                while (lock);         // do nothing
                lock=TRUE;
                critical section
                lock=FALSE;
                remainder section
        } while (TRUE);
}
```

Does it work?

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic
- 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 Pi 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);
}
```

Solution to Critical-section Problem Using Locks

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

Overview

- The Critical-Section Problem
- Software Solutions
- **Synchronization Hardware**
- Semaphores
- Monitors
- Synchronization Examples

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
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Interrupt Disabling

```
do {
        disable interrupt;
        critical section
        enable interrupt;
        remainder section
    } while (1);
```

Previous Solution

```
do {
      while (lock);   // do nothing
      lock=TRUE;
      critical section
      lock=FALSE;
      remainder section
} while (1);
```

The solution does not guaranteed the mutual exclusion because the test and set on lock are not atomic

Test-And-Set Instruction

- Definition:

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

Solution using Test-And-Set

```
boolean lock=FALSE;

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

Swap Instruction

```
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

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

This solution guarantees mutual exclusion but not bounded waiting

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);
```

Overview

- The Critical-Section Problem
- Software Solutions
- Synchronization Hardware
- **Semaphores**
- Monitors
- Synchronization Examples

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore S – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - wait() and signal()
  - Originally called P() and V()

Semaphore

```
wait (S) {
        while (S <= 0);    // busy wait
        S--;
}

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

wait() and signal() must be atomic

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
- Can implement a counting semaphore S as a binary semaphore

Semaphore as Mutex Tool

- Shared data:

  semaphore mutex=1;

- Process *Pi:*

```
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
- Could have busy waiting (spinlock)
  - Busy waiting wastes CPU cycles
  - But avoids context switches
  - May be useful when the critical section is short and/or rarely occupied
- However applications may spend lots of time in critical sections and therefore, generally, this is not a good solution.

Semaphore Implementation

- 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.

Implementation

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

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

Semaphore as a Synchronization Tool

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

P_i	P_j
⋮	⋮
A	$wait(flag)$
$signal(flag)$	B

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
$wait(S);$	$wait(Q);$
$wait(Q);$	$wait(S);$
$\vdots$	$\vdots$
$signal(S);$	$signal(Q);$
$signal(Q);$	$signal(S);$

- **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
- Semaphore empty initialized to the value N.

Bounded-Buffer Problem

Producer Process

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

} while (1);
```

Consumer Process

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

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

- Variants
  - No new reader must wait when a writer is waiting for data access
  - No new reader can start reading when a writer is waiting for data access

Readers-Writers Problem

- Shared Data
  - Data set
  - Integer readcount initialized to 0
  - Semaphore mutex initialized to 1
    - Mutual exclusion on readcount
  - Semaphore wrt initialized to 1
    - Mutual exclusion on the data set by writers

Readers-Writers Problem

- The structure of a writer process

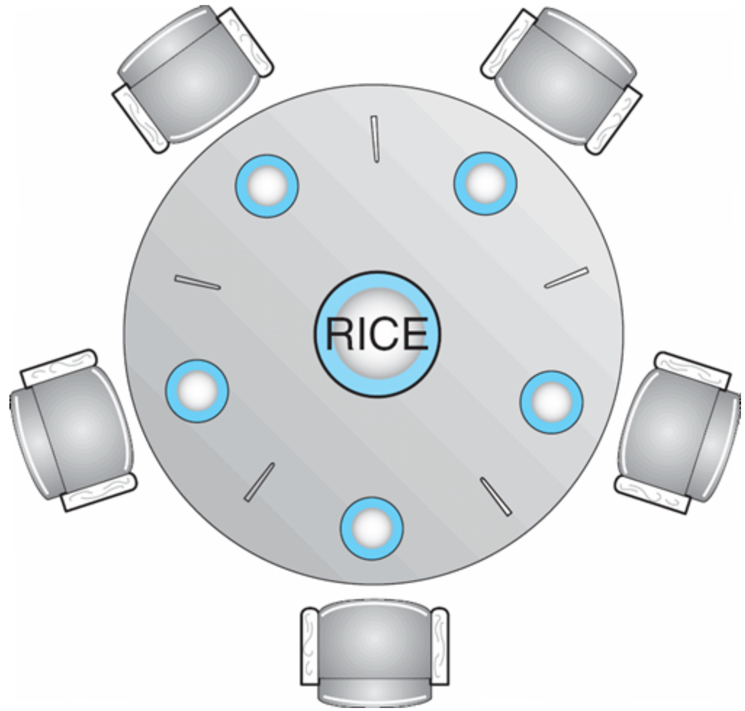
```
do {
        wait (wrt) ;
        // writing is performed
        signal (wrt) ;
} while (TRUE);
```

Readers-Writers Problem

- 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);
```

Dining-Philosophers Problem



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

Dining-Philosophers Problem

- 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);
```

Dining-Philosophers Problem

- Deadlock
  - A deadlock occurs if all philosophers start eating simultaneously
- Possible solutions to avoid deadlocks
  - Only 4 philosophers can sit around the table
  - A philosopher can take his/her chopsticks only if they both are free
  - An odd philosopher takes the chopstick on its left first, and then the one on its right; an even philosopher takes the opposite approach.
- Starvation
  - Any solution must avoid that a philosopher may starve

Problems with Semaphores

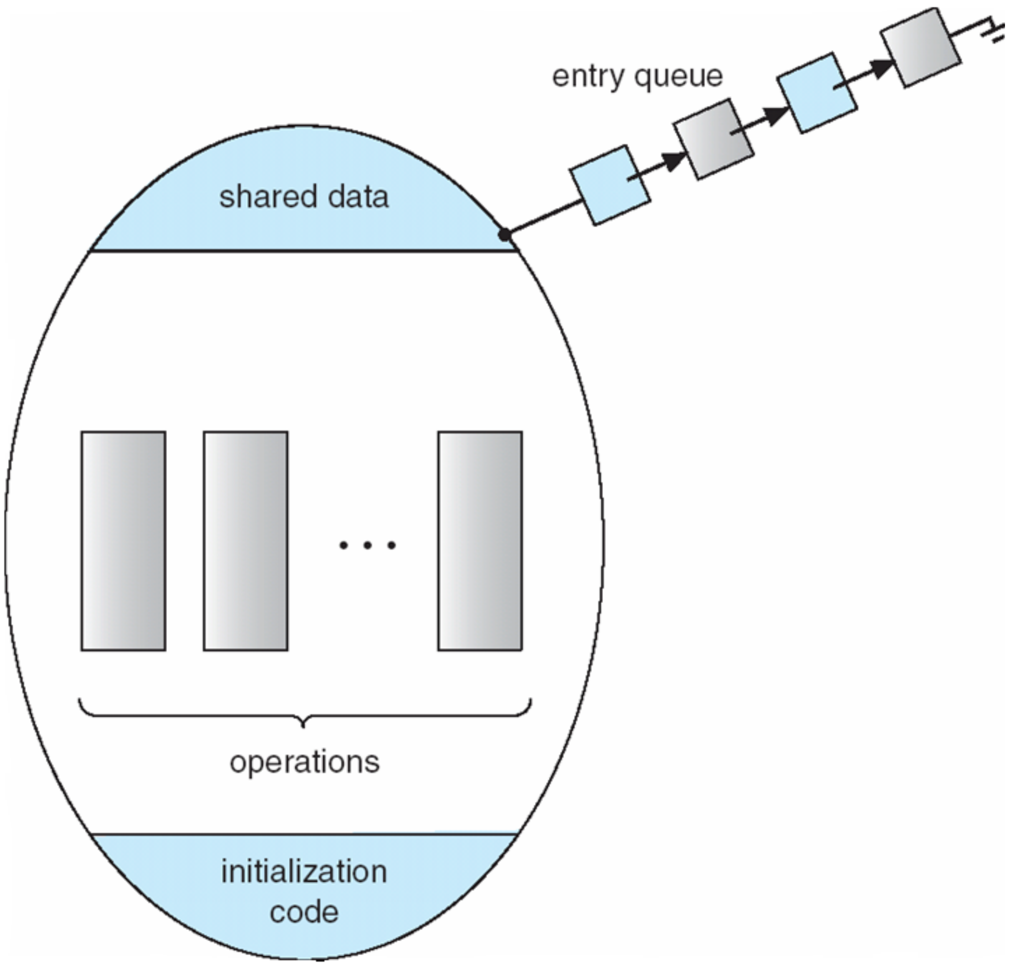
- Incorrect use of semaphore operations:
  - *wait (mutex) … wait (mutex)*
  - *signal (mutex) …. wait (mutex)*
  - Omitting of *wait (mutex) or signal (mutex)* (or both)

Overview

- The Critical-Section Problem
- Software Solutions
- Synchronization Hardware
- Semaphores
- **Monitors**
- Synchronization Examples

Monitor

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization

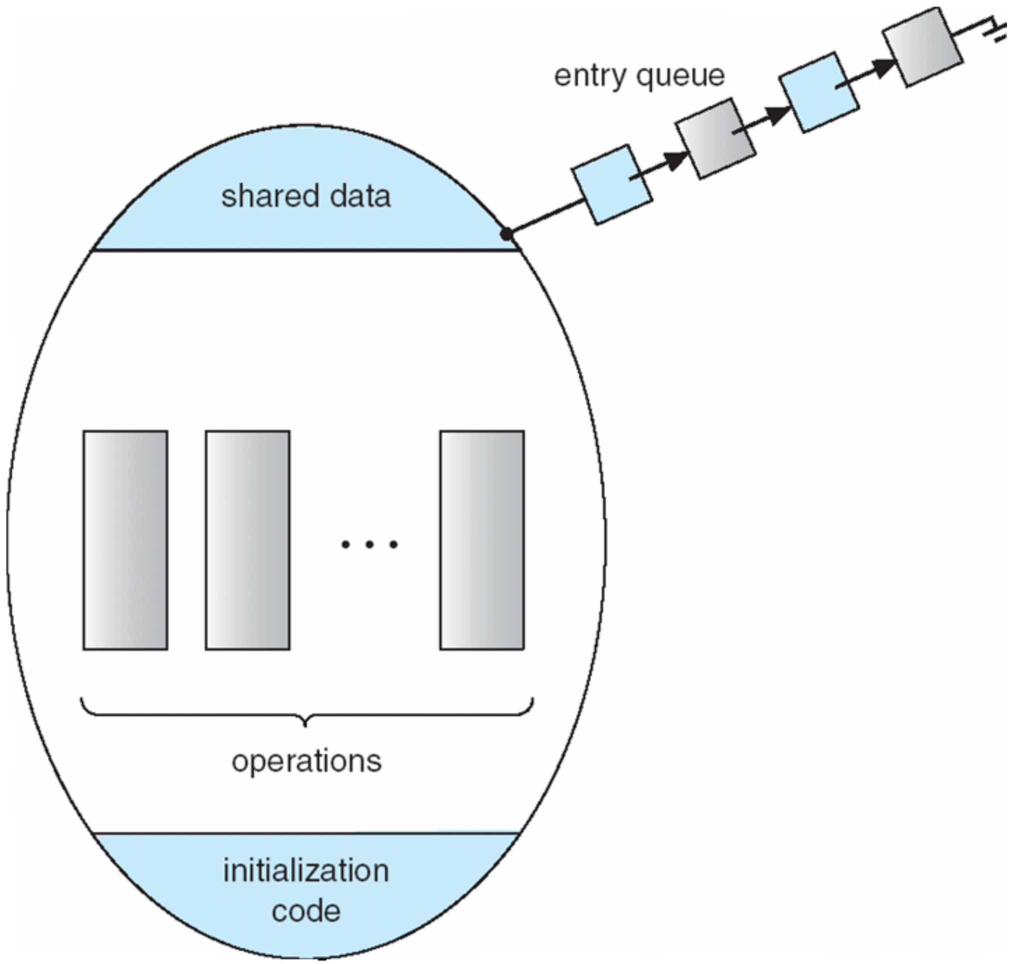


Monitor

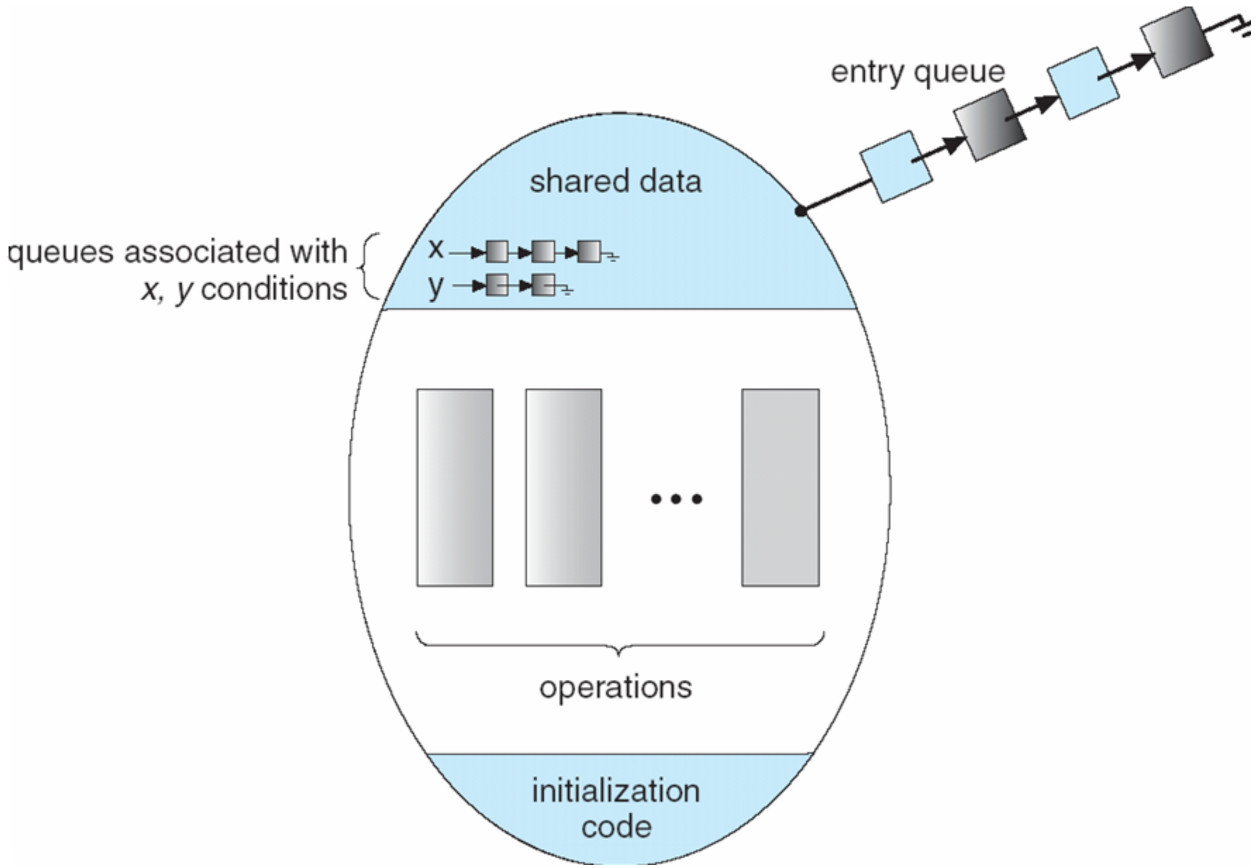
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

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

Basic Monitor



Monitor with Condition Variables



Condition Variables

- condition x, y;
- Two operations on a condition variable:
  - x.wait() – a process that invokes the operation is (always) suspended.
  - x.signal() – resumes one of processes (if any) that invoked x.wait().
- Variants (P executes x.signal() and Q was blocked on x)
  - Signal and wait: P waits for Q leaving the monitor – or blocking on another condition variable – before proceedings on
  - Signal and proceed: Q waits for P leaving the monitor – or blocking on another condition variable – before proceedings on
  - Signal and Leave: P executes signal and leaves the monitor (Concurrent Pascal)

Solution to Dining Philosophers

- Based on monitors
- The solution assumes that
  - A philosopher can take his/her chopsticks only when are both free
- The proposed solution is deadlock-free

Solution to Dining Philosophers

```
monitor DP {
        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

```
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

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

```
while (1) {
    Think;
    DP.pickup (p);
    Eat;
    DP.putdown (p);
}
```

Overview

- The Critical-Section Problem
- Software Solutions
- Synchronization Hardware
- Semaphores
- Monitors
- **Synchronization Examples**

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of mechanisms to support multitasking, multithreading (including real-time threads), and multiprocessing
- Adaptive mutexes for efficiency when protecting data from short code segments
- Uses semaphores, condition variables and readers-writers locks when longer sections of code need access to data

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources from kernel threads on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- For *out-of-kernel* synch provides dispatcher objects
  - may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

Linux Synchronization

- Linux:
  - Prior Version 2.6, non-preemptive kernel
    - A task executed in system mode cannot be interrupted, even by a higher-priority thread
  - Version 2.6 and later, fully preemptive
- Linux provides:
    - semaphores
    - spin locks
- Linux kernel
  - Multi-processor
    - Enable/disable spinlocks (active only for short times)
  - Single-processor
    - Disable/Enable preemption

Pthreads Synchronization

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

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