Chapter 6: Process Synchronization

- The slides do not contain all the information and cannot be treated as a study material for Operating System. Please refer the text book for exams.

TOPICS

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

Too much milk

Time	You	Your Roommate
3:00	Arrive home	
3:05	Look in fridge, no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Arrive at grocery	Look in fridge, no milk
3:25	Buy milk, leave	Leave for grocery
3:30		
3:35	Arrive home	Arrive at grocery
3:36	Put milk in fridge	
3:40		Buy milk, leave
3:45		
3:50		Arrive home
3:51		Put milk in fridge
3:51	Oh, no! ***Too much milk!!***	

The problem is that Look in fridge, no milk to Put milk in fridge is not an **atomic** operation

BACKGROUND

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanism to ensure that cooperating processes access shared data sequentially.

Bounded buffer problem

Shared Data

```
#define BUFFER_SIZE 10

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

Bounded buffer problem

Producer Process

```
item nextProduced;
…

while(1) {
 /*  produce an item and put in nextProduced  */
nextProduced = getitem();
while( counter == BUFFER_SIZE);
buffer[in] = nextProduced;
in = (in + 1) % BUFFER_SIZE;
counter++;
}
```

Bounded buffer problem

Consumer Process

```
item nextConsumed;
…

while(1) {
while( counter == 0);
nextConsumed = buffer[out];
out = (out+ 1) % BUFFER_SIZE;
counter--;
/*  consume the item in nextConsumed  */
}
```

Bounded buffer problem

- counter++ could be implemented as

  register1 = counter
  register1 = register1 + 1
  counter = register1

- counter-- could be implemented as

  register2 = counter
  register2 = register2 - 1
  count = register2

Bounded buffer problem

- Consider this execution interleaving with "count = 5" initially:
- S0: producer register1 =counter {register1= 5}
  S1: producer register1 = register1 + 1 {register1= 6}
  S2: consumer register2 = counter {register2 = 5}
  S3: consumer register2 = register2 - 1 {register2 = 4}
  S4: producer counter = register1 {count = 6 }
  S5: consumer counter = register2 {count = 4}

- The value would be either 4 or 6 based on who updates the values last
- The correct value should be 5

BOUNDED BUFFER PROBLEM

- **Race Condition**
- The situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on which process finish last

- We have to serialize the execution of the process so that only one process can access the shared data at a time.

THE CRITICAL SECTION PROBLEM

- **Critical Section**: A piece of code in a cooperating process in which the process the process may update shared data (variable, file, database, etc.)

- **Critical Section Problem** : Serialize execution of critical sections in cooperating processes.

Solutions of critical section problem

- Software Based solutions
- Hardware Based solutions (CPU based instructions)
- Operating system based solutions

STRUCTURE OF SOLUTION

```
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 and decision is taken by a process that is not executing in its remainder section

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

THE CRITICAL SECTION PROBLEM

- Code executing in kernel mode would have race condition – several kernel mode process execute
- Modifying number of files open list, Memory allocation list, process list
- Two approach to handle critical section
- Preemptive Kernels
  - Allows a process to be preempted while running in kernel mode
  - Can have race conditions
  - Suitable for real time programming
  - More responsive
- Non Preemptive Kernels
  - Does not allow process running in Kernel mode to be preempted
  - No race condition

PETERSON'S SOLUTION

Process Pi

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

Process Pj

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

Peterson's solution

- Does it satisfy ?

1. Mutual exclusion
2. Progress
3. Bounded-waiting

- Because the way modern computer architecture perform basic machine-language instructions, such as load and store, there is no guarantee that Peterson would work correctly

Synchronization Hardware

- Any solution to the critical section requires a lock
- 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 as message is passed to all processors
    - Message passing delays entry to critical section, system efficiency decreases
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable

Solution to Critical-section Problem Using Locks

```
do {
        acquire lock

                critical section

        release lock

                remainder section
   } while (TRUE);
```

TestAndSet Instruction

- Semantics:

```
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

- Semantics:

```
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);
```

Are they good solution?

- These algorithms satisfy mutual exclusion, but they don't satisfy bounded wait.

- Another algorithm uses
- boolean waiting[n];
- boolean lock;

- All initialized to false

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

- Hardware based solutions to the critical-section are complicated for application programmers to use
- Semaphore – synchronization tool

- S is an integer – integer variable – only accessed by 2 standard operations – wait() and signal() – 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**
- Counting semaphores can be used to control access to a given resource consisting of finite number of instances
- Semaphore is intialized to the number of resources available

Semaphore as General Synchronization Tool

- Provides mutual exclusion

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

Semaphore Implementation

- **Disadvantage :** Busy waiting
- Continual looping is a problem in real multiprogramming system
- Busy waiting wastes CPU cycles that some other process might be able to use productively
- This type of semaphore is called **spinlock** as process spins while waiting for the lock
- **Advantage**: No context switch which takes considerable time
- When locks are held for short time , spinlocks are useful

Semaphore Implementation with no Busy waiting

- Modify the definition of wait() and signal() semaphore operations

- Rather than waiting it can block itself
  - **block** – place the process invoking the operation on the appropriate waiting queue and state is switched to waiting

- The process that is blocked on semaphore S, should be restarted when some other process executes a signal()
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Semaphore Implementation with no Busy waiting

- Semaphore as structure

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

- Semaphore value may be negative
- If the values is negative, its magnitude gives the number of process waiting on that semaphore
- List of processes is implemented by link field in each PCB
- List could be implemented as a FIFO queue – to avoid starvation

Semaphore Implementation with no Busy waiting

- 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);
        }
}
```

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- In a single processor – we can inhibit interrupts when wait() and signal() are executing
- In multiprocessor – interrupts must be disabled in every processor else instructions from different process may be interleaved
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section

Semaphore Implementation

- Could now have **busy waiting** in critical section implementation – short code – busy wait for short time
- Note that if applications may spend lots of time in critical sections then this is not a good solution.

Issues with semaphores

- Semaphores provide a powerful tool for enforcing mutual exclusion and coordinating processes
- But wait(S) and signal(S) are scattered among several processes. hence difficult to understand their effects
- Bad use of semaphores would result in
- Deadlocks
- Violation of mutual exclusion
- Priority inversion

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);

Deadlock and Starvation

- **Starvation** – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended – LIFO queue
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**
- All processes that are accessing resources needed by higher priority process inherit the higher priority until they are finished
- Once finished they are reverted to original values

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

- 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

- 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

Readers-Writers Problem

- Several variations of how readers and writers are treated – all involve priorities
- Simplest – first readers – writers problem – no reader is kept waiting unless a writer has already obtained permission to use the shared object
- Second – if a writer is ready , the writer performs its write as soon as possible – no new readers may start reading

Readers-Writers Problem

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

- Mutex is used to ensure mutual exclusion when the variable readcount is updated
- Wrt is semaphore used by the writers, its also used by first and last reader that enters or exits critical section
- The selection of which process gets the critical section next is scheduled by the scheduler

Readers-Writers Problem

- The structure of a writer process

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

Readers-Writers Problem – 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);
```

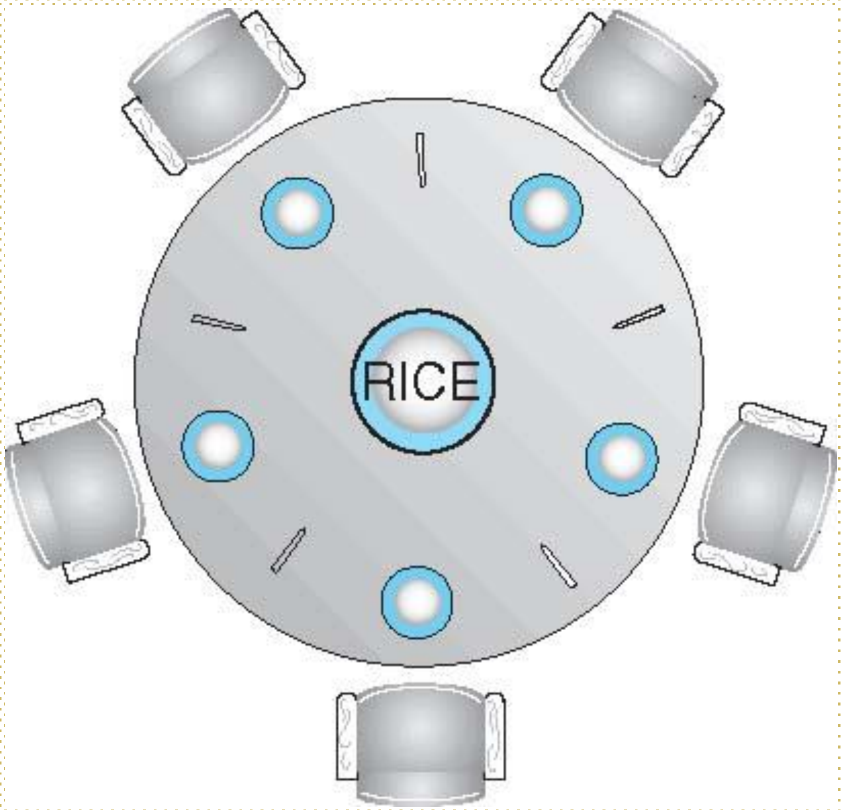
Readers-Writers Problem

- The problem is generalized to provide reader-writer lock
- Can specify the mode in which you want the lock(read/write)

- Many process can get the lock in read mode but only one for writing

- Useful in following situations
- In applications where it is easy to identify which processes only read shared data and which write
- In applications where there are more readers than writers. Reader-writer lock require more overhead to establish than semaphores or mutual exclusion locks

Dining-Philosophers Problem



- 5 philosophers spend their lives thinking and eating
- Philosophers when hungry need to pick 2 chopsticks out of 5 that are closest to her and cannot pick a chopstick from the neighbor

- Important problem that represent need to allocate several resources among several process in deadlock free and starvation free manner

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?
- Deadlock

Dining-Philosophers Problem Algorithm

- Several remedies to deadlock problem
- Allow at most 4 philosophers to be sitting simultaneously to be hungry at a time.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available – pick in critical section
- Use asymmetric solution – odd picks up left and then right while even picks right and then left

Monitors

- Certain timing errors
- Process interchange he order in which wait() and signal() operations on semaphore mutex are executed – several process will execute in their critical section violating mutual exclusion
- Process replace signal(mutex) with wait(mutex) – deadlock will occur
- If they omit either wait() or signal() – deadlock or no mutual exclusion
- If initial value of semaphore is not set right

- High level language synchronization construct – monitors
- Shift the responsibility of enforcing mutual exclusion from the programmer (where it resides when semaphores are used) to the compiler

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type* – encapsulates private data with public methods that operate on that data
- Construct ensures that Only one process is active within the monitor at a time
- But not powerful enough to model some synchronization schemes
- We need to define additional synchronization – condition construct

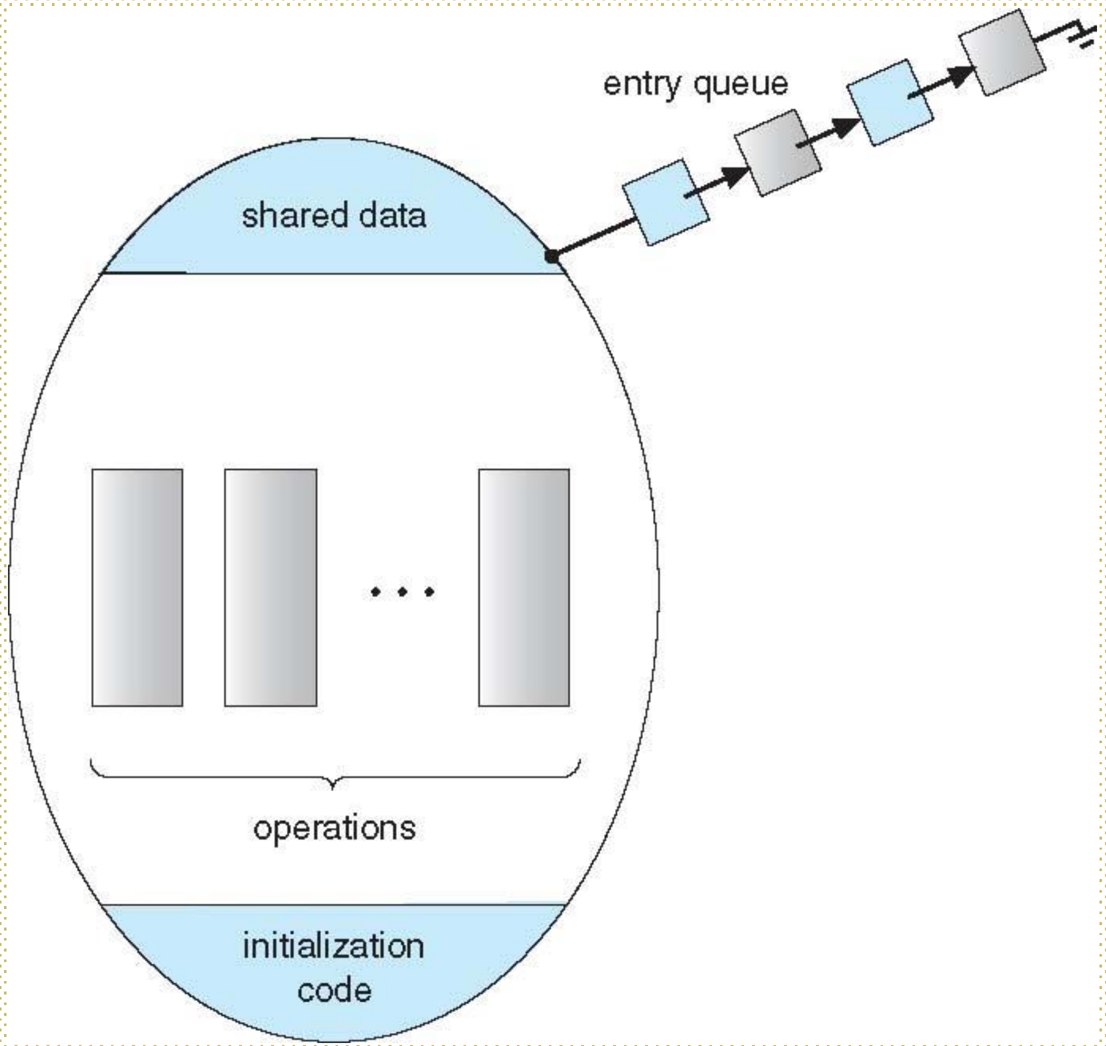
Syntax of Monitor

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

  procedure Pn (…) {……}

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

Schematic view of a Monitor

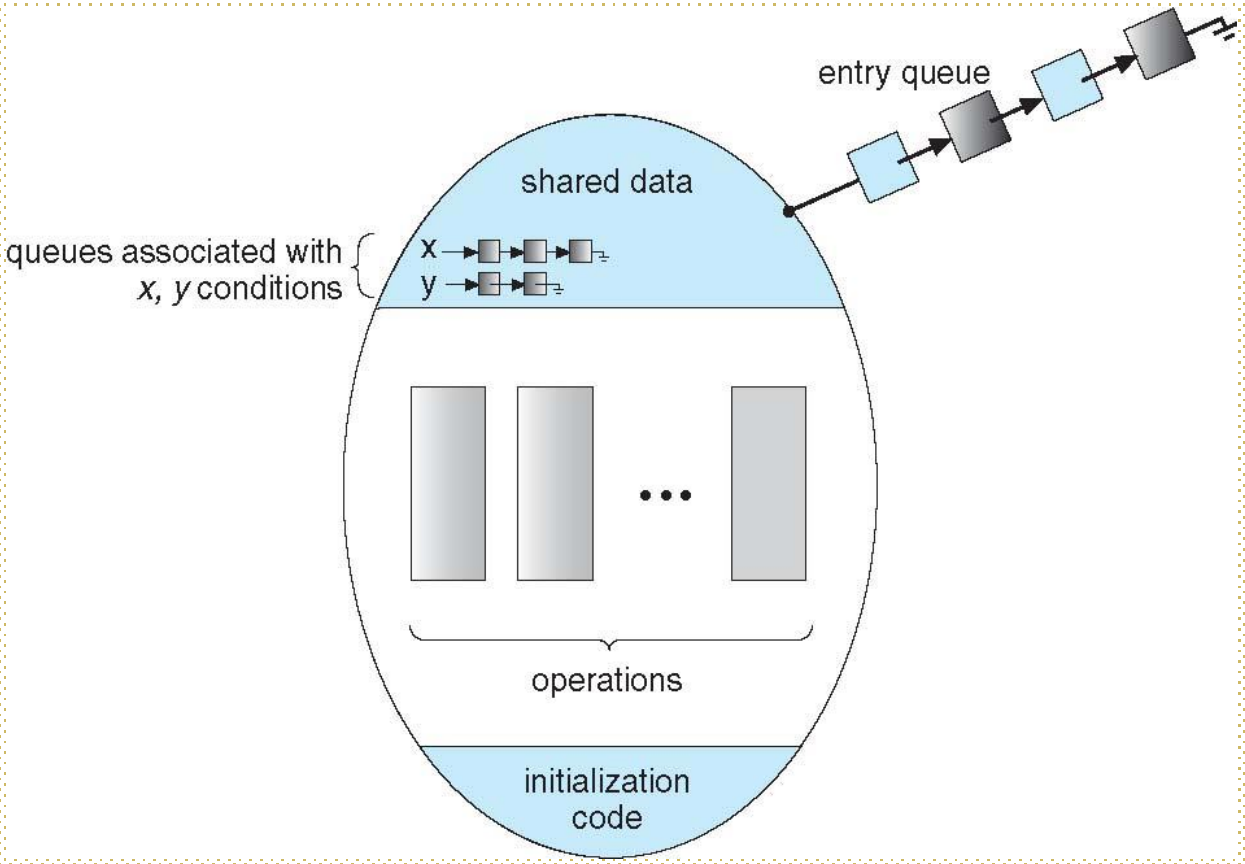


Monitors with Condition Variables

- Additional synchronization constructs can be modeled with condition variables
- condition x, y;

- Only Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended until another process invokes x.signal ()
  - x.signal () – resumes exactly one of processes (if any) that invoked x.wait ()
  - If no x.wait () on the variable, then it has no effect on the variable x
- Different from semaphore wait and signal

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

- Conceptually both process can continue their execution

- 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

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 (P was already executing in the monitor)

  - 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);
       }
```

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 *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

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;
   void acquire(int time) {
                    if (busy)
                          x.wait(time);
                    busy = TRUE;
   }
   void release() {
                    busy = FALSE;
                    x.signal();
   }
initialization code() {
    busy = FALSE;
   } }
```

A Monitor to Allocate Single Resource

A process that needs to access resource must follow the sequence

R.acquire(t);

….

Access the resource;

…

R.release();

R is an instance of type ResourceAllocator

Problems with monitors

- A process might access a resource without first gaining access permission to the resource
- A process might never release a resource once it has been granted access to the resource
- A process might attempt to release a resource that it never requested
- A process might request the same resource twice(without releasing)

- Previously we had to worry about correct usage of semaphores, now correct use of higher level programmer defined operations – compiler can no longer assist

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Linux Synchronization

- Single processor – Disable kernel preemption, enable kernel preemption
- Multi processor – Acquire spin lock, release spin lock

- Each task has a thread_info structure, containing preempt_count
- When lock acquired – increase count – no of locks held by the task
- If value > 0, not safe to preempt the task