CS 343 - Operating Systems

Module-3C

Process Synchronization Using HW Locks and Monitors

Dr. John Jose

Associate Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati

Session Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Semaphores
- Mutex Locks
- Monitors

Semaphore

- Synchronization tool for processes to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations

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

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

Synchronization Hardware - Locks

- 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
- Modern machines provide special atomic hardware instructions
  - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

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

Synchronization Using test_and_set Instruction

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

1. Executed atomically
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 to FALSE

```
do{
   while (test_and_set(&lock))
        ; /* do nothing */

        /* critical section */

        lock = false;

        /* remainder section */

 }while (true);
```

```
boolean test_and_set
(boolean *lock)

{

   boolean rv = *lock;

   *lock = TRUE;

   return rv:

 }
```

Synchronization Using compare_and_swap Instruction

```
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” only if “value” ==“expected”. That is, the swap takes place only under this condition.

Solution using compare_and_swap ()

- Shared integer lock initialized to 0;

```
do {
      while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */
       /* critical section */
      lock = 0;
       /* remainder section */
  } while (true);
```

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**

Synchronization Using acquire() and release()

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

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

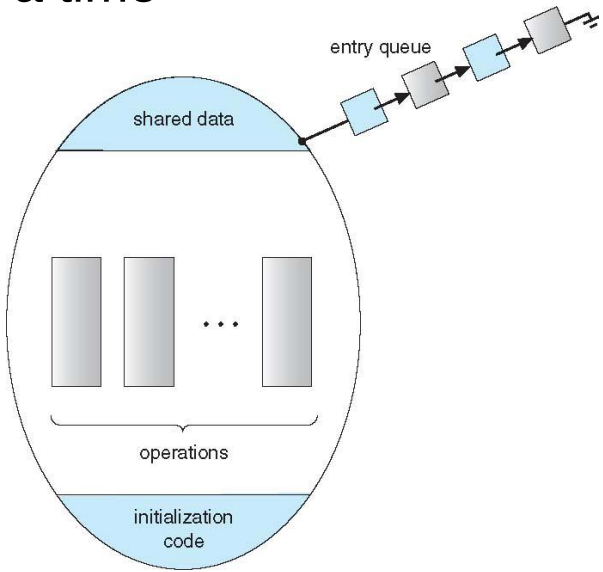
Monitors

- A monitor is a programming language construct that controls access to shared data
- Synchronization code added by compiler, enforced at runtime
- A monitor is a module that encapsulates
  - Shared data structures
  - Procedures that operate on the shared data structures
  - Synchronization between concurrent procedure invocations
- A monitor protects its data from unstructured access
- It guarantees that threads accessing its data through its procedures interact only in legitimate ways

Monitors

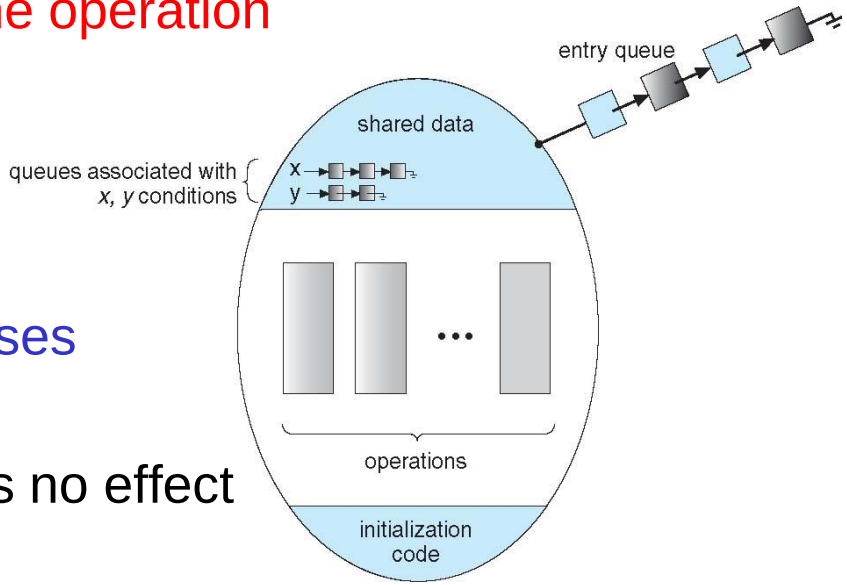
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Abstract data type, internal variables only accessible by code within the procedure
- One process may be active within the monitor at a time

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



Condition Variables

- Two operations are allowed 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()**
- If no **x.wait()** on the variable, then it has no effect on the variable



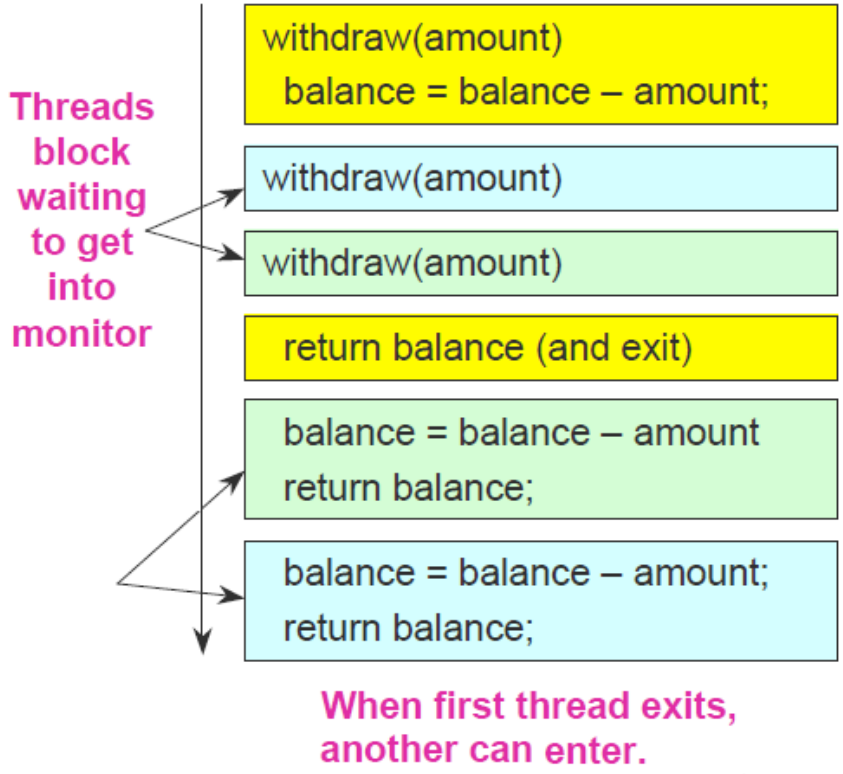
Condition Variables Choices

- If process P invokes **x.signal(),** and process Q is suspended in **x.wait()**, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- 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

Implementation using Monitors

```
Monitor account {
  double balance;

  double withdraw(amount) {
    balance = balance – amount;
    return balance;
  }
}
```



Practice Problem-1

- Consider 4 processes P, Q, R, and S to be scheduled on a processor using round robin CPU scheduling having a time quantum of 5 units. All 4 processes arrive at time T=0 in the order P, Q, R, and S. While applying round robin scheduling, there is exactly one context switch from P to Q, exactly one context switch from P to S and no context switch from P to R. There are exactly 2 context switches from S to P. Switching from a terminated process to a process in ready queue is also counted as context switching. CPU burst of each process is a positive integer value less than or equal to 15 units.

  (a) What should be the CPU burst time of each individual process if the average turnaround is as low as possible? Draw a neat labelled Gantt chart showing the scheduling order and turnaround time calculation.

  (b) What should be the CPU burst time of each individual process if average turnaround is as high as possible? Draw a neat labelled Gantt chart showing the scheduling order and turnaround time calculation.

Practice Problem-2

- Consider 4 processes P0, P1, P2, P3 that was scheduled on a CPU using a dynamic round robin scheduling with a time quantum of ***3x*** clock cycles, where ***x*** indicates the round number, starting with 1, followed by 2, and then 3 and so on for subsequent rounds. ie, time quantum increases in each round. Each process will get at most one slot of running state in each round. The arrival time of P0, P1, P2, P3 are at clock cycles 0, 2, 4 and 6, respectively. It was found that P0 used the entire time quantum of 5 full rounds to complete and P3 used entire time quantum of 2 full rounds to complete their execution. P2 took another 31 cycles more to complete its execution after the completion of P3. P0 waited for 8 clock cycles after its turn in first round to start the execution in second round. With the help of a neat Gantt chart answer the following.

  (a) How long is the CPU burst of each of the processes?

  (b) What is the average turnaround time for the set of these 4 processes?

  (c) What is the waiting time for process P3?

Practice Problem-3

- Consider the 4 tuple entries given for 3 processes to be interpreted as <process_name, T_arrival, CPU burst, initial priority value>: <A, 2, 5, 2>, <B, 5, 7, 0>, <C, 0, 5, 1>. The system uses pre-emptive priority scheduling. Lower the priority value, higher is the priority. The priority value of a process decreases by 1 for every 3 continuous cycles in waiting state (If a process with a priority x enters ready queue at time t, its priority value changes to x-1 at end of t+3 and to x-2 at end of t+6 etc..). Similarly, when a process is running for 3 continuous cycles, its priority increases by 1. Whenever the priority value of a running process is same or higher than that of another process in the ready state, the running process is preempted and the one in ready queue is promoted to running state. If such a preemption happens when the priority value of running process is 0, then its priority is incremented by 1 during the context switch. Tie breaker for same priority level is done with FCFS order.

  (a) Find out the number of context switch that happens before B complete its execution?

  (b) What is the waiting time of process C?

  (c) At time the end of T=15, what is the priority value of each of the incomplete process?

Practice Problem-4

- Consider two processes P and Q with an event Pe and Qe, respectively executing inside a loop. P and Q are executing in a multiprogramming environment on a single processor where context switching can happen arbitrarily at any time. We want events Pe and Qe to strictly execute alternatively in the sequence Pe, Qe, Pe, Qe .. (10 times). Give an algorithmic solution to ensure this using semaphores.

johnjose@iitg.ac.in

http://www.iitg.ac.in/johnjose/