OPERATING SYSTEMS

Threads and Concurrency 09

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

OPERATING SYSTEMS

Course Syllabus - Unit 2

UNIT 2: Threads and Concurrency

Introduction to Threads, types of threads, Multicore Programming, Multithreading Models, Thread creation, Thread Scheduling, PThreads and Windows Threads, Mutual Exclusion and Synchronization: software approaches, principles of concurrency, hardware support, Mutex Locks, Semaphores. Classic problems of Synchronization: Bounded-Buffer Problem, Readers -Writers problem, Dining Philosophers Problem concepts. Synchronization Examples - Synchronisation mechanisms provided by Linux/Windows/Pthreads. Deadlocks: principles of deadlock, tools for detection and Prevention.

OPERATING SYSTEMS

Course Outline - Unit 2

13	4.1,4.2	Introduction to Threads, types of threads, Multicore Programming.	4	42.8
14	4.3,5.4	Multithreading Models, Thread creation, Thread Scheduling	4	
15	4.4	Pthreads and Windows Threads	4	
16	6.1-6.3	Mutual Exclusion and Synchronization: software approaches	6	
17	6.3-6.4	principles of concurrency, hardware support	6	
18	6.5,6.6	Mutex Locks, Semaphores	6	
19	6.7.1-6.7.3	Classic problems of Synchronization: Bounded-Buffer Problem, Readers -Writers problem, Dining Philosophers Problem concepts	6	
20	6.9	Synchronization Examples - Synchronisation mechanisms provided by Linux/Windows/Pthreads.	6	
21	Handouts	Demonstration of programming examples on process synchronization		
22	7.1-7.3	Deadlocks: principles of deadlock, Deadlock Characterization.	7	
23	7.4	Deadlock Prevention, Deadlock example	7	
24	7.6	Deadlock Detection	7	

Topics Outline

- Semaphores
  - Binary
  - Counting
- Advantages and Disadvantages of Semaphores
- Semaphore implementation
- Semaphore implementation with no busy waiting
- Deadlock and Starvation
- Mutex Versus Semaphore

Synchronization: Semaphores

- Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, **wait** and **signal** that are used for process synchronization.

- It is a Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.

Synchronization: Semaphores

The definitions of wait and signal are as follows −

- **Wait**

  The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

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

Synchronization: Semaphores

The definitions of wait and signal are as follows −

- **Signal**

  The signal operation increments the value of its argument S.

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

Synchronization: Semaphores

Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows

- **Binary Semaphores**

  The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

Synchronization: Semaphores

Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows

- **Counting Semaphores**

  These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.

Synchronization: Semaphores

Advantages of Semaphores

- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.

- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.

- Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

Synchronization: Semaphores

Disadvantages of Semaphores

- Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.

- Semaphores are impractical for large scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.

- Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

Synchronization: Semaphores Implementation

- Must guarantee that no two processes can execute the wait() and signal() on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the wait and signal code are placed in the critical section
- Could now have busy waiting in critical section implementation. But implementation code is short
- Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Synchronization: Semaphores Implementation

- With each semaphore there is an associated waiting queue

- Each entry in a waiting queue has two data items: a value of type integer and a pointer to next record in the list

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

Synchronization: Semaphores Implementation with no busy waiting

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

```
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);
   }
```

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

```
      P0
  wait(S);
   wait(Q);
    ...
   signal(S);
     signal(Q);
```

Deadlock and Starvation

- **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

P_0

```
wait(S);
 wait(Q);
 ...
 signal(S);
  signal(Q);
```

P_1

```
wait(Q);
wait(S);
  ...
   signal(Q);
    signal(S);
```

Signal versus Lock

- A signal is an asynchronous notification sent to a process or to a specific thread within the same process in order to notify it of an event that occurred.

- Locks are methods of synchronization used to prevent multiple threads from accessing a resource at the same time. Usually, they are advisory locks, meaning that each thread must cooperate in gaining and releasing locks. More difficult to implement and less common, mandatory locks actually prevent any other thread from accessing a resource, and issue an exception if this occurs.

Mutex versus Semaphore

- Semaphore is a signalling mechanism i.e. processes perform wait() and signal() operation to indicate whether they are acquiring or releasing the resource,

- The Mutex is locking mechanism, the process has to acquire the lock on mutex object if it wants to acquire the resource.

Mutex versus Semaphore

Basis for Comparison	Mutex	Semaphore
Basics	Mutex is a locking mechanism.	Semaphore is a signalling mechanism.
Existence Type	Mutex is an Object	Semaphore is an Integer
Function	Mutex allow multiple program thread to access a single resource but not simultaneously.	Semaphore allow multiple program threads to access a finite instance of resources.
Ownership	Mutex object lock is released only by the process that has acquired the lock on it.	Semaphore value can be changed by any process acquiring or releasing the resource.

Mutex versus Semaphore

Basis for Comparison	Mutex	Semaphore
Categorization	Mutex is not categorized further.	Semaphore can be categorized into counting semaphore and binary semaphore.
Operation	Mutex object is locked or unlocked by the process requesting or releasing the resource.	Semaphore value is modified using wait() and signal() operation.
Resources Occupied	If a mutex object is already locked, the process requesting for resources waits and queued by the system till lock is released.	If all resources are being used, the process requesting for resource performs wait() operation and block itself till semaphore count become greater than one.

Global and Local Semaphore in Client Server Environment

There are two types of semaphores: **local** semaphores and **global** semaphores.

- A **local** semaphore is accessible by **all** processes on the **same** workstation and **only on** the workstation.
- A local semaphore can be created by prefixing the name of the semaphore with a dollar sign ($).
- You use local semaphores to monitor operations among processes executing on the **same workstation**.
- For example, a local semaphore can be used to monitor access to an interprocess array shared by all the processes in your **single-user database** or on the workstation.
- A **global** semaphore is **accessible** to all **users** and all their **processes**. You use global semaphores to monitor operations among users of a **multi-user database**.

Some Frequently Asked Questions

Can a thread acquire more than one lock (Mutex) ?
Yes

Can a mutex be locked more than once ?
No

What happens if a non-recursive mutex is locked more than once ?
Deadlock

Are binary semaphore and mutex same ?
No

What are events ?
The semantics of mutex, semaphore, event, critical section, etc are same. We should consult the OS documentation for more details

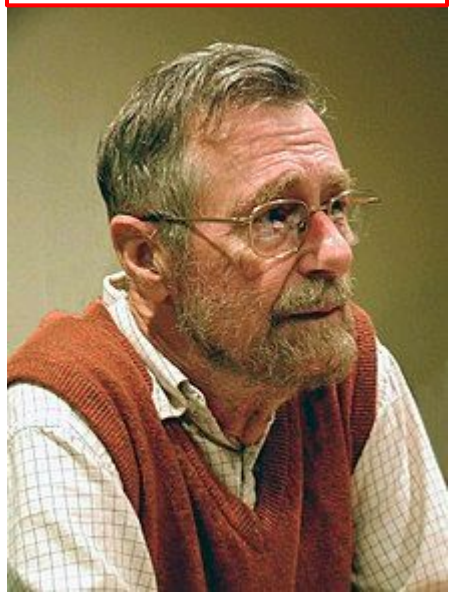
Can we acquire mutex/semaphore in an Interrupt Service Routine ?
It is not recommended

What we mean by "thread blocking on mutex/semaphore" when they are not available ?
Every synchronization primitive has a waiting list associated with it.

Is it necessary that a thread must block always when resource is not available ?
Not necessary.

Semaphores

Edsger Wybe Dijkstra



Topics Uncovered in this Session

- Semaphores
  - Binary
  - Counting
- Advantages and Disadvantages of Semaphores
- Semaphore implementation
- Semaphore implementation with no busy waiting
- Deadlock and Starvation
- Mutex Versus Semaphore

THANK YOU

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

nitin.pujari@pes.edu

For Course Deliverables by the Anchor Faculty click on www.pesuacademy.com