OPERATING SYSTEMS

Mutual Exclusion & Synchronization: Mutex Locks, Semaphores

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

OPERATING SYSTEMS

Course Syllabus - Unit 2

12 Hours

Unit 2: Threads & 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

13	Introduction to Threads, types of threads, Multicore Programming, Multithreading Models	4.1 - 4.3	42.8
14	Thread creation, Thread Scheduling	5.4	
15	Pthreads and Windows Threads	4.4	
16	Mutual Exclusion and Synchronization: software approaches,	6.1-6.2	
17	principles of concurrency, hardware support	6.3-6.4	
18	Mutex Locks, Semaphores	6.5, 6.6	
19	Classic problems of Synchronization: Bounded-Buffer Problem, Readers-Writers problem	6.7-6.8	
20	Dining-Philosophers Problem	6.8	
21	Synchronization Examples: Synchronisation mechanisms provided by Linux/Windows/Pthreads.	6.9	
22	Deadlocks: principles of deadlock, Deadlock Characterization	7.1-7.3	
23	Deadlock Prevention, Deadlock example	7.4-7.5	
24	Deadlock Detection, Algorithm	7.6	

Course Outline

17	principles of concurrency, hardware support	6.3-6.4
18	Mutex Locks, Semaphores	6.5, 6.6

Topic Outline

- Mutex Locks - acquire(), release()

- Recursive Mutexes, Reader / Writer Mutexes, Spinlocks

- Semaphores
    - Binary
    - Counting

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

Synchronization: Mutex Locks

- Mutex is short for MUTual EXclusion.
- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest software tool is mutex lock
- It is used to protect a critical section by first calling acquire() a lock function, then release() the lock release function

Synchronization: Mutex Locks

- 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: Mutex Locks

- Mutex is a mutual exclusion object that synchronizes access to a resource.
- It is created with a unique name at the start of a program.
- The Mutex is a locking mechanism that makes sure only one thread / process can acquire the Mutex at a time and enter its critical section.

Synchronization: Mutex Locks

- This thread / process only releases the Mutex when it exits the critical section.

- Unless the word is qualified with additional terms such as shared mutex, recursive mutex or read/write mutex then it refers to a type of lockable object that can be owned by exactly one thread at a time.

- Only the thread that acquired the lock can release the lock on a mutex. When the mutex is locked, any attempt to acquire the lock will fail or block, even if that attempt is done by the same thread.

Recursive Mutexes

- A recursive mutex is similar to a plain mutex, but one thread may own multiple locks on it at the same time.
- If a lock on a recursive mutex has been acquired by thread A, then thread A can acquire further locks on the recursive mutex without releasing the locks already held.
- However, thread B cannot acquire any locks on the recursive mutex until **all** the locks held by thread A have been released.

Synchronization: Mutex Locks

Recursive Mutexes

- In most cases, a recursive mutex is undesirable, since the it makes it harder to reason correctly about the code. With a plain mutex, if you ensure that the invariants on the protected resource are valid before you release ownership then you know that when you acquire ownership those invariants will be valid.

- With a recursive mutex this is not the case, since being able to acquire the lock does not mean that the lock was not already held, by the current thread, and therefore does not imply that the invariants are valid.

Reader/Writer Mutexes

- Sometimes called shared mutexes, multiple-reader/single-writer mutexes or just read/write mutexes, these offer two distinct types of ownership: shared ownership, also called read ownership, or a read lock, and exclusive ownership, also called write ownership, or a write lock.

- Exclusive ownership works just like ownership of a plain mutex: only one thread may hold an exclusive lock on the mutex, only that thread can release the lock. No other thread may hold any type of lock on the mutex while that thread holds its lock.

Reader/Writer Mutexes

- Shared ownership is more lax. Any number of threads may take shared ownership of a mutex at the same time. No thread may take an exclusive lock on the mutex while any thread holds a shared lock.

- These mutexes are typically used for protecting shared data that is seldom updated, but cannot be safely updated if any thread is reading it. The reading threads thus take shared ownership while they are reading the data. When the data needs to be modified, the modifying thread first takes exclusive ownership of the mutex, thus ensuring that no other thread is reading it, then releases the exclusive lock after the modification has been done.

Spinlocks

- A spinlock is a special type of mutex that does not use OS synchronization functions when a lock operation has to wait. Instead, it just keeps trying to update the mutex data structure to take the lock in a loop.
- If the lock is not held very often, and/or is only held for very short periods, then this can be more efficient than calling heavyweight thread synchronization functions.
- However, if the processor has to loop too many times then it is just wasting time doing nothing, and the system would do better if the OS scheduled another thread with active work to do instead of the thread failing to acquire the spinlock.

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--;
}
```

The definitions of wait and signal are as follows −

- **Signal**

  The signal operation increments the value of its argument S.

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

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.

PES UNIVERSITY ONLINE

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 last 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 no busywaiting

- 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 busywaiting

```
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);
```

```
   P1
wait(Q);
wait(S);
   ...
    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

- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process - Solved via priority-inheritance protocol

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

```
  P1
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.

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