Operating System

Lecture 6 /

Chapter 5 (Process synchronization)

Background

The Critical-Section Problem

Mutex Locks

Semaphores

Chapter 6 (CPU scheduling)

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Background

A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages.

Concurrent access to shared data may result in data inconsistency, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

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

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual Exclusion استبعاد متبادل** - 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
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

Solution to Critical-Section Problem

1. Peterson's Solution
2. Synchronization Hardware (Locking)
3. Mutex Locks
4. Semaphores

Solution to Critical-Section Problem

All these solutions are based on the premise of locking that is, protecting critical regions through the use of locks. As we shall see, the designs of such locks can be quite sophisticated.

The critical-section problem could be solved simply in a single-processor environment if we could **prevent interrupts from occurring while a shared variable was being modified.**

General idea of the solution is to use a counter, an integer variable counter, initialized to 0. counter is incremented every time we add a new item to the queue and is decremented every time we remove one item the queue.

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**

Mutex Locks

operating-systems designers build software tools to solve the critical-section problem.

The simplest of these tools is the **mutex lock**. (In fact, the term ***mutex*** is short for ***mut***ual ***ex***clusion.) We use the mutex lock to protect critical regions and thus prevent race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.
The **acquire()function** acquires يكتسب the lock, and the **release() function** releases the lock.

A mutex lock has a boolean variable available whose value indicates if the lock is available or not. If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released.

Mutex Locks

The main disadvantage of the implementation given here is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire().

In fact, this type of mutex lock is also called a **spinlock** because the process "spins" while waiting for the lock to become available.

Semaphore

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

Semaphore ***S*** – integer variable

Can only be accessed via two indivisible (atomic) operations

wait() and **signal()**

Originally called **P()** and **V()**

Definition of the **wait() operation**

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

Definition of the **signal() operation**

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

Semaphore

Semaphore is a more robust tool that can behave similarly to a mutex lock but can also provide more sophisticated ways for processes to synchronize their activities.

A **semaphore** S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().

The wait() operation was originally termed P (from the Dutch ***proberen,*** "to test"); signal() was originally called V (from ***verhogen,*** "to increment").

All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modification (S--), must be executed without interruption.

Semaphore

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count).

* When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

Question Semaphore

semaphore values are never negative under the classical definition of semaphores with busy waiting. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the wait() operation.

Deadlock and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.

The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be **deadlocked**.

OS Next Lecture

Mid-Term Exam

Chapter 1, 2, 3, 4, 5 & 6 + Linux lecture (handout)

The source book sentences is the only standard answer & valuated

2017.4.11

Chapter 7
Dead Locks