Process Synchronization (3)

Dr. Jun Zheng

CSE325 Principles of Operating Systems

10/2/2019

NEW MEXICO TECH

SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Mutex Locks

- 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**

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

acquire() and release()

```
 acquire() {
      while (!available)
          ; /* busy wait */
      available = false;;
  }
 release() {
      available = true;
  }
 do {
  acquire lock
      critical section
  release lock
     remainder section
} while (true);
```

Question

- Why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems?

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Answer

- Spinlocks are not appropriate for single-processor systems because the condition that would break a process out of the spinlock can be obtained only by executing a different process. If the process is not relinquishing the processor, other processes do not get the opportunity to set the program condition required for the first process to make progress. In a multiprocessor system, other processes execute on other processors and thereby modify the program state in order to release the first process from the spinlock.

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 Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore S as a binary semaphore

Semaphore 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
  - Note that applications may spend lots of time in critical sections and therefore busy waiting is not a good solution

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);
```

- **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

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Priority Inversion Example

- Three jobs with three priorities (L, M, H)
- Assume that process *H* requires resource *R*, which is currently being accessed by process *L*.
- Ordinarily, process *H* would wait for *L* to finish using resource *R*. However, now suppose that process *M* becomes runnable, thereby preempting process *L*.
- Indirectly, a process with a lower priority process M has affected how long process *H* must wait for *L* to relinquish resource *R*.

Mars PathFinder

- Mars Pathfinder, a NASA space probe that landed a robot, the Sojourner rover, on Mars in 1997 to conduct experiments. Shortly after the Sojourner began operating, it started to experience frequent computer resets. Each reset reinitialized all hardware and software, including communications. If the problem had not been solved, the Sojourner would have failed in its mission.
- The problem was caused by the fact that one high-priority task, "bc dist," was taking longer than expected to complete its work. This task was being forced to wait for a shared resource that was held by the lower-priority "ASI/MET" task, which in turn was preempted by multiple medium-priority tasks. The "bc dist" task would stall waiting for the shared resource, and ultimately the "bc sched" task would discover the problem and perform the reset. The Sojourner was suffering from a typical case of priority inversion.
- How to solve it?

Priority Inheritance Protocol

- L executes its critical section at H's (high) priority. As a result, M will be unable to preempt L and will be blocked.
- When L exits its critical section, it regains its original (low) priority and awakens H (which was blocked by L).
- H, having high priority, preempts L and runs to completion.

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

In-Class Work 3

- The following figure shows the sequence of semaphore operations at the beginning and at the end of the tasks A, B.
- Determine for each of the 4 cases a, b, c and d, whether or in which sequence the tasks are executed, using the initializations of the semaphore variables given in Table 1.
- wait() is an atomic operation that waits for semaphore to become positive, then decrements it by 1, i.e. if the semaphore is 0, wait() is blocked.
- signal() is an atomic operation that increments the semaphore by 1, waking up a waiting wait() if any.
- SA, SB are the two semaphores corresponding to Tasks A and B.

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

In-Class Work 3

```
Task A              Task B
wait(SA)            wait(SB)
wait(SA)            wait(SB)
…                       …
…                       …
…                       …
signal(SB)          signal(SA)
END                 END
```

Table 1

	a	b	c	d
SA	2	1	0	2
SB	0	1	2	1