CSCE 311 - Operating Systems
Synchronization Review

Qiang Zeng, Ph.D.
Fall 2018
Previous Class...

• Single-slot producer-consumer problem
• Multi-slot producer-consumer problem
Outline

• Barrier Problem
• Readers-Writers Problem
• Review of Synchronization
• Issues of lock-based programming
Barrier Problem
Barrier problem

• Goal: given a number, N, of processes, each process has to wait at some point of its program until all processes reach the point
• Implement the API `Barrier()`, which is called by each process
  – The N-1 processes block until the last one calls it
Solution

\[
\begin{align*}
n &= \text{the number of threads} \\
count &= 0 \\
mutex &= \text{Semaphore}(1) \\
barrier &= \text{Semaphore}(0)
\end{align*}
\]

Barrier() {
    down(mutex)
    count += 1
    if (count == n)
        for (i = 0; i < n; ++i)
            up(barrier)
    up(mutex)
    up(mutex)
    down(barrier)
}
Another solution

```plaintext
n = the number of threads
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)

mutex.wait()
    count = count + 1
mutex.signal()

if count == n: barrier.signal()

barrier.wait()
barrier.signal()
```

Is it possible that two processes both arrive here and find “count == n”?

A: It is possible. But extra up() operations will not cause errors. Certainly, you can move the “if” statement into mutex-guarded region.
Readers-Writers Problem
Readers-Writers Problem

• Problem statement:
  – *Reader* threads only read the object
  – *Writer* threads modify the object
  – Writers must have exclusive access to the object
  – Unlimited number of readers can access the object

• Occurs frequently in real systems, e.g.,
  – Online airline reservation system
  – Multithreaded caching Web proxy
Solution

Shared:

```c
int readcnt;    /* Initially = 0 */
semaphore r, whole; /* Initially = 1 */
```

Writers:

```c
void writer(void)
{
    while (1) {
        down(whole);

        /* Critical section */
        /* Writing here */

        up(whole);
    }
}
```
What if the “whole” lock is already acquired by the writer, and the first reader comes in?

```c
void reader(void)
{
    while (1) {
        /* Increment readcnt */
        down(r); /* Only one reader a time */
        readcnt++;
        if (readcnt == 1) /* First reader in */
            down(whole); /* Lock out writers */
        up(r);

        /* Read; multiple readers may be here */

        /* Decrement readcnt */
        down(r);
        readcnt--; 
        if (readcnt == 0) /* Last out */
            up(whole); /* Let in writers */
        up(r);
    }
}
```
Big picture of synchronization primitives

Sync. Primitives

Busy-waiting

Software solutions (Dekker’, Bakery, etc.)

Hardware-assisted solutions (based on atomic read-modify-write instructions)

Semaphore: it contains an internal counter indicating the number of resources.

Binary Semaphore is a special semaphore, whose counter value can only be 0 or 1; it can be used as a mutex

Monitor: a high-level synchronization construct; shared data + procedures + lock

Condition variable: guarded by a lock to wait on some event
Summary of the uses of Semaphore

• Mutual exclusion (using binary semaphores)
• Synchronizing the use of shared resources, e.g.,
  – The single-slot restroom problem
  – The bar problem
  – The producer-consumer problem
  – The counter of the semaphore should be initialized to
    the # of resources available
• Enforcing order, e.g.,
  – Operation O1 in Process P1 has to occur after O2 in P2
Relations between Condition Variable & Monitor

• A Monitor may contain zero or more CVs
  – Very often, procedures in Monitor rely on CVs to implement complex synchronization
  – Recall that a CV has to be used with a lock; a Monitor can provide the lock, so you do not have to explicitly use a lock for employing a CV in a Monitor

• The use of CVs is not limited to Monitors
  – E.g., Pthread library provides CVs but not Monitors
Condition variable VS. Semaphore

- A CV has to work with a lock (e.g., the lock provided by a monitor), while a Semaphore does not
- Condition Variables allow broadcast() operation, while Semaphores do not
- A Semaphore has a counter and a wait queue, while a Condition Variable only has a wait queue
  - You need to initialize the counter when using a Semaphore. A Condition Variable has no notion of “the number of resources”
  - If there are no processes in the wait queue
    - The up() operation of a semaphore increments the counter
    - The signal() of a CV has no effect (i.e., the “signal” gets lost)
Issues with locks

• You may introduce deadlock

• Nobody can make progress sometimes (when the lock owner is scheduled out)

• Priority inversion:
  – Given priority-based scheduling, L is running in CS ; H also needs to run in CS ; H waits for L to come out of CS ; M interrupts L and starts running ; M runs till completion and relinquishes control ; L resumes and starts running till the end of CS ; H enters CS and starts running.
  – It is problematic as H is delayed by M now

• Solution: non-blocking programming / data structures (but they are very complex and, very often, slow)
Summary

• Barrier Problem
• Readers-Writers Problem
• Review
  – Summary of uses of Semaphores
  – Relation between CV and Monitor
  – CV vs. Semaphore
• Issues of lock-based programming