Lecture 11: Chapter 6 “Process Synchronization” (cont'd)
Today's class

- Will cover sections 6.4—6.5
- Terms and keywords
  - Atomic operation, test-and-set, semaphores, mutexes
- Learning objectives
  - To have an understanding of why atomic operations are needed
  - To use semaphores and mutexes,
Atomic operations

- They are operations (a sequence of actions) that happen as if they were a single instruction \((atomically)\)
- Atomic, from “atom” which means “indivisible”
The critical section problem

Problem description

- How to regulate execution of a portion of the code called “critical section” so that only one thread can execute it at a time, in an atomic way.
- Programming model:
  1. entry code
  2. critical section
  3. exit code

Criteria for correct solution

- Mutual exclusion
  - If one thread is in the critical section, then no other should be in there
- Progress
  - If the critical section is “unoccupied” and there is a thread attempting to get into the critical section, then it should be allowed to enter without indefinite delay
- Fairness
  - A thread competing with other threads should eventually get access to the critical section
TestAndSet

- An example of atomic operation
- Pseudocode

```
TestAndSet(*a)
{
  Response = *a;
  *a = TRUE;
  Return reponse;
}
```

Usage example

```
Do {
  While (TestAndSet(&lock))
    ; /* busy wait */
    // critical section
    lock = FALSE;
    // remainder section
} while (TRUE);
```
Swap Instruction

Another atomic operation
• Definition:

```c
void Swap (boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```
Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key
- Solution:

```c
do {
    key = TRUE;
    while ( key == TRUE)
    Swap (&lock, &key);

    // critical section

    lock = FALSE;

    // remainder section

} while (TRUE);
```
Busy waiting

- It is “waiting” while being in RUNNING state.
  - Usually, looping over testing a variable's value
- Ideally waiting should be done in in WAIT state.
- Synchronization mechanisms should exhibit blocked waiting
Bounded-waiting Mutual Exclusion with TestAndSet()

do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;
    // critical section
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // remainder section
} while (TRUE);
Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore $S$ – integer variable
- Two standard operations modify $S$: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - `wait (S) {
      while $S \leq 0$
      ; // no-op
      $S--$;
  }
  - `signal (S) {
        $S++$;
    }`
Semaphore as General Synchronization Tool

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as *mutex locks*
- Can implement a counting semaphore $S$ as a binary semaphore
- Provides mutual exclusion

```c
Semaphore mutex;  // initialized to 1
do {
    wait (mutex);
    // Critical Section
    signal (mutex);
    // remainder section
} while (TRUE);
```
Semaphore Implementation

- Must guarantee that no two processes can execute `wait()` and `signal()` on the same semaphore at the same time
- Thus, 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.
Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - 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.
Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:
  ```c
  wait(semaphore *S) {
    S->value--;  
    if (S->value < 0) {
      add this process to S->list;
      block();
    }
  }
  ```

- Implementation of signal:
  ```c
  signal(semaphore *S) {
    S->value++;  
    if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
    }
  }
  ```
POSIX semaphores

- sem_init
- sem_post
- sem_wait
- sem_overview
- sem_destroy
POSIX mutexes

- pthread_mutex_init
- pthread_mutex_lock
- pthread_mutex_trylock
- pthread_mutex_unlock
- pthread_mutex_destroy