Synchronization Part 2

CSE 410, Spring 2004
Computer Systems

http://www.cs.washington.edu/education/courses/410/04sp/

Readings and References

- Reading
  - Chapter 7, Sections 7.4 through 7.7, *Operating System Concepts*, Silberschatz, Galvin, and Gagne

- Other References
  - The Java Tutorial, Synchronizing Threads

Shared Stack

```c
void Stack::Push(Item *item) {
    item->next = top;
    top = item;
}
```

- Suppose two threads, red and blue, share this code and a Stack `s`
- The two threads both operate on `s`
  - each calls `s->Push(...)`
- Execution is interleaved by context switches

Stack Example

- Now suppose that a context switch occurs at an “inconvenient” time, so that the actual execution order is

  1. `item->next = top;`
  2. `item->next = top;`
  3. `top = item;`
  4. `top = item;`

  context switch from red to blue

  context switch from blue to red
Disaster Strikes

Shared Stack Solution

- How do we fix this using locks?

```c
void Stack::Push(Item *item) {
    lock->Acquire();
    item->next = top;
    top = item;
    lock->Release();
}
```

Correct Execution

- Only one thread can hold the lock

```c
lock->Acquire();
item->next = top;
top = item;
lock->Release();
```
How can Pop wait for a Stack item?

Synchronized stack using locks

Stack::

```cpp
Stack::Push(Item * item) {
    lock->Acquire();
    push item on stack
    lock->Release();
}

Stack::Pop() {
    lock->Acquire();
    pop item from stack
    lock->Release();
    return item;
}
```

- This works okay if we don’t want to wait inside Pop and can just return <no data available>
  » in order to wait we want to go to sleep inside the critical section
  » other threads won’t be able to run because Pop holds the lock
  » condition variables make it possible to go to sleep inside a critical section, by releasing the lock and going to sleep in one atomic operation

Condition Variables

- A condition variable is a queue of threads waiting for something inside a critical section
- There are three operations
  » Wait()—release lock & go to sleep (atomic); reacquire lock upon awakening
  » Signal()—wake up a waiting thread, if any
  » Broadcast()—wake up all waiting threads
- A thread must hold the lock when doing condition variable operations

Monitors

- Monitor: a lock and condition variables
- Key addition is the ability to inexpensively and reliably wait for a condition change
- Can be implemented as a separate class
  » The class contains code and private data
  » Since the data is private, only monitor code can access it
  » Only one thread is allowed to run in the monitor at a time
- Can be implemented directly in other classes using locks and condition variables

Stack with Condition Variables

Pop can now wait for something to be pushed onto the stack

Stack::

```cpp
Stack::Push(Item * item) {
    lock->Acquire();
    push item on stack
    condition->signal(lock);
    lock->Release();
}

Stack::Pop() {
    lock->Acquire();
    while( nothing on stack ) {
        condition->wait(lock);
    }
    pop item from stack
    lock->Release();
    return item;
}
```
Synchronization in Win2K/XP

- Windows has locks (known as mutexes)
  - CreateMutex--returns a handle to a new mutex
  - WaitForSingleObject--acquires the mutex
  - ReleaseMutex--releases the mutex

- Windows has condition variables (known as events)
  - CreateEvent--returns a handle to a new event
  - WaitForSingleObject--waits for the event to happen
  - SetEvent--signals the event, waking up one waiting thread

Synchronization in Java

- Java has locks (on any object)
  - The Java platform associates a lock with every object that has synchronized code
  - A method or a code block {...} can be synchronized
  - The lock is acquired before the block is entered and released when the block is exited

- Java has condition variables (wait lists)
  - The Object class defines wait(), notify(), notifyAll() methods
  - By inheritance, all objects of all classes have those methods