Synchronization Part 2

CSE 410 - Computer Systems
November 28, 2001

Readings and References

• Reading
  » Chapter 7, Sections 7.4 through 7.7, Operating System Concepts, Silberschatz, Galvin, and Gagne

• Other References

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 \( s->\text{Push}(\ldots) \)
• 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. \( \text{item->next = top; } \) (context switch from red to blue)
  2. \( \text{item->next = top; } \)
  3. \( \text{top = item; } \)
  4. \( \text{top = item; } \) (context switch from blue to red)
Disaster Strikes

- How do we fix this using locks?

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

Correct Execution

- Only one thread can hold the lock

```cpp
lock->Acquire();
item->next = top;
top = item;
lock->Release();
```

Correct Execution

- Red acquires the lock
- Blue tries to acquire the lock
- Red releases the lock
- Blue acquires the lock
- Blue releases the lock
How can Pop wait for a Stack item?

- Synchronized stack using locks

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

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

- 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 atomically releasing the lock and going to sleep

Monitors

- Monitor: a lock and condition variables

- Key addition is the ability to inexpensively and reliably wait for a condition change
- Often 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 also implement directly in other classes using locks and condition variables

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

Stack with Condition Variables

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

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

Item * Stack::Pop() {
  lock->Acquire();
  while( nothing on stack ) {
    condition->wait( lock );
    pop item from stack
    lock->Release();
    return item;
  }
Database Readers and Writers

- Many threads may read the database at the same time
- If any thread is writing the database, then no other thread may read or write
  - when a reader enters, it must wait if there is a writer inside
  - when a writer enters, it must wait if there is a reader or writer inside
  - writers have priority over readers

Constraints

- Reader can access the database when no writers are active
  - condition okToRead
- Writer can access the database when no readers or writers are active
  - condition okToWrite
- Only one thread of any type can manipulate the shared state variables at a time
  - lock

Basic Algorithm

Database::read()
wait until no writers
access database
checkout -- wake up waiting writer (if any)

Database::write()
wait until no readers or writers
access database
checkout -- wake up waiting readers or writers

State Variables

Condition okToRead = TRUE;  // “signaled”
Condition okToWrite = TRUE; // “signaled”
Lock lock = FREE;           // “signaled”

AR=0; // number of active readers
AW=0; // number of active writers
WR=0; // number of waiting readers
WW=0; // number of waiting writers
Semaphores

- Semaphores were first synchronization mechanism
  - Don't use semaphores, use condition variables instead
- The semaphore is an integer variable that has two **atomic** operations:
  - P() (the entry procedure) wait for semaphore to become positive and then decrement it by 1
  - V() (the exit procedure) increment semaphore by 1, wake up a waiting P if any
  - P and V are from the Dutch for **probieren** (to try) and **verhogen** (to increment) - named by Dijkstra

Synchronization in NT

- NT has locks (known as mutexes)
  - CreateMutex--returns a handle to a new mutex
  - WaitForSingleObject--acquires the mutex
  - ReleaseMutex--releases the mutex
- NT has **events** instead of condition variables
  - CreateEvent--returns a handle to a new event
  - WaitForSingleObject--waits for the event to happen
  - SetEvent--signals the event, waking up one waiting thread
Advice for Threads Programming #1

- Always do things the same way
  - you can focus on the core problem because the standard approach becomes a habit
  - makes it easier for other people to read (modify and debug) your code
  - you might be able to cut corners occasionally and save a line or two of code
    - spend time convincing yourself it works
    - spend time convincing others that it works with your comments
    - NOT WORTH IT!

Advice for Threads Programming #2

- Always use monitors (locks + condition variables) or events
  - 99% monitor/event code is more clear than semaphore code because monitor code is "self-documenting"
  - occasionally a semaphore might fit what you are doing perfectly
  - what if the code needs to change, is it still a perfect fit?

Advice for Threads Programming #3

- Always acquire the lock at the beginning of a procedure and release it before returning
  - if there is a logical chunk of code that requires holding a lock, then it should probably be its own procedure
  - we are sometimes lazy about creating new procedures when we should (don't be lazy)
  - always do things the same way (rule #1)

Advice for Threads Programming #4

- Always use while instead of if when checking a synchronization condition
- Many implementations allow for a thread to be waked up even though the condition is not true. Must wait again.

```cpp
Item * Stack::Pop() {
    lock->Acquire();
    while ( nothing on stack ) {
        condition->wait( lock );
    }
    pop item from stack
    lock->Release();
    return item;
}
```

```cpp
Item * Stack::Pop() {
    lock->Acquire();
    if ( nothing on stack ) {
        condition->wait( lock );
    }
    pop item from stack
    lock->Release();
    return item;
}
```