CS 162 Operating Systems and Systems Programming

Professor: Anthony D. Joseph

Spring 2001

Lecture 9: Readers-Writers; Language Support for Synchronization

9.0 Main points:

Review definition of monitors and condition variables
Illustrate use of monitors and condition variables by solving readers-writers problem
Language support can make thread programming easier.
Threads are a fundamental OS abstraction, but be careful about how you use them.
Summarize concurrency section

9.1 Readers/Writers

9.1.1 Motivation

Shared database (for example, bank balances, or airline seats)

Two classes of users:

- Readers – never modify database
- Writers – read and modify database

Using a single lock on the database would be overly restrictive. Want:

- many readers at same time
- only one writer at same time

9.1.2 Constraints

1. Readers can access database when no writers (Condition okToRead)
2. Writers can access database when no readers or writers (Condition okToWrite)

3. Only one thread manipulates state variables at a time.

9.1.3 Solution

Basic structure of solution

```
Reader
            wait until no writers
            access database
            check out – wake up waiting writer

Writer
            wait until no readers or writers
            access database
            check out – wake up waiting readers or writer

State variables:
            # of active readers – AR  = 0
            # of active writers – AW  = 0
            # of waiting readers – WR  = 0
            # of waiting writers – WW  = 0

            Condition okToRead = NIL
            Condition okToWrite = NIL
            Lock lock = FREE
```

Code:

```
Reader() {
            // first check self into system
            lock.Acquire();
            while ((AW + WW) > 0) {    // check if safe to read
                                       // if any writers, wait
                 WR++;
                 okToRead.Wait(&lock);
                 WR--;
            }
            AR++;
            lock.Release();

            Access DB

            // check self  out of system
            lock.Acquire();
            AR--;
            if (AR == 0 && WW > 0)     //if no other readers still
                                       // active, wake up writer
                 okToWrite.Signal(&lock);
            lock.Release();
}
```

```
Writer() {                  //  symmetrical
          // check in
          lock.Acquire();
          while ((AW + AR) > 0) {    // check if safe to write
                                     // if any readers or
                                     // writers, wait
               WW++;
               okToWrite->Wait(&lock);
               WW--;
          }
          AW++;
          lock.Release();

          Access DB

          // check out
          lock.Acquire();
          AW--;
          if (WW > 0)     // give priority to other writers
               okToWrite->Signal(&lock);
          else if (WR > 0)
               okToRead->Broadcast(&lock);
          lock.Release();
}
```

Questions:

1. Can readers starve?

2. Why does checkRead need a while?

9.2 Comparison between semaphores and monitors

Illustrate the differences by considering: can we build monitors out of semaphores? After all, semaphores provide atomic operations and queuing.

Does this work?

```
Wait() { semaphore->P(); }

Signal() { semaphore->V(); }
```

Condition variables only work inside of a lock. If you try to use semaphores inside of a lock, you have to watch for deadlock.

Does this work?

```
Wait(Lock *lock) {
           lock->Release();
           semaphore->P();
           lock->Acquire();
}
Signal() {
           semaphore->V();
}
```

Condition variables have no history, but semaphores do have history.

What if thread signals and no one is waiting?

No op.

What if thread later waits?

Thread waits.

What if thread V's and no one is waiting?

Increment.

What if thread later does P?

Decrement and continue.

In other words, P + V are commutative – result is the same no matter what order they occur. Condition variables are not commutative. That's why they must be in a critical section – need to access state variables to do their job.

Does this fix the problem?

```
Signal() {
          if semaphore queue is not empty
               semaphore->V();
}
```

For one, not legal to look at contents of semaphore queue. But also: race condition – signaler can slip in after lock is released, and before wait. Then waiter never wakes up!

Need to release lock and go to sleep atomically.

Is it possible to implement condition variables using semaphores? Yes, but exercise left to the reader!

9.3 Summary of Monitors

Monitors represent the logic of the program – wait if necessary, signal if change something so waiter might need to wake up.

```
lock
while (need to wait) {
          wait();
}
unlock

do something so no need to wait

lock
signal();
unlock
```

Fall 1999 1.5 hour Lecture 8 continued here…

9.4 Language support for thread synchronization

The problem with requiring the programmer to specify lock acquire and release statements is that he might forget to put a release everywhere it is needed.

9.4.1 Languages like C

This is not too bad in a language like C: just make sure you know *all* the code paths out of a critical section.

```
int Rtn() {
lock.acquire();
…
if (exception) {
          lock.release();
          return errReturnCode;
}
…
lock.release();
return OK;
}
```

Watch out for `setjmp/longjmp`!

9.4.2 Languages like C++

Languages that support exceptions – like C++ – are more problematic:

```
void Rtn() {
lock.acquire();
…
DoFoo();
…
lock.release();
}
```

```
void DoFoo() {
…
if (exception) throw errException;
…
}
```

`Rtn()` needs to catch the exception, release the lock, and then re-throw the exception:

```
void Rtn() {
lock.acquire();
try {
            …
            DoFoo();
            …
            }
catch (…) { // C++ syntax for catching any exception.
            lock.release();
            throw; // C++ syntax for re-throwing an exception.
}
lock.release();
}
```

9.4.3 Java

Java has explicit support for threads and thread synchronization.

Bank account example:

```
class Account {
private int balance;

// object constructor
public Account (int initialBalance) {
            balance = initialBalance;
}

public synchronized int getBalance() {
            return balance;
```

```
}

public synchronized void deposit(int amount) {
balance += amount;
}
```

Each Account object has an associated lock, which gets automatically acquired and released on entry and exit from each *synchronized* method.

Java also has *synchronized* statements:

```
synchronized (object) {
            …
}
```

Every Java object has an associated lock. Any Java object can be used to control access to a *synchronized* block of code. The synchronizing object's lock is acquired on entry and released on exit, *even if exit is by means of a thrown exception:*

```
synchronized (object) {
…
DoFoo();
…
}
void DoFoo() {
…
throw errException;
…
}
```

How to wait in a synchronized method or code block:

- `void wait(long timeout);`
- `void wait(long timeout, int nanoseconds);`
- `void wait();`

How to signal in a synchronized method or code block:

- `void notify();` *`wakes up the longest waiting waiter`*
- `void notifyAll();` *`like broadcast,wakes up all waiters`*

Notes:

- Only *one* condition variable per lock.
- Condition variables can wait for a bounded length of time. This is useful for handling exception cases where something has failed. For example:

```
   t1 = time.now();
   while (!ATMRequest()) {
wait(LONG_TIME);
t2 = time.now();
if (t2 – t1 > LONG_TIME) CheckIfMachineBroken();
}
```

One reason why all Java Virtual Machines are not equivalent:

Different thread scheduling policies. The language specification does not stipulate whether preemptive scheduling is required or what granularity of time slice should be used if preemptive scheduling is provided.

9.5 Concurrency Summary

Basic idea in all of computer science is to abstract complexity behind clean interfaces. We've done that!

Physical Hardware	Programming Abstraction
Single CPU, interrupts, test&set	Concurrent sequential execution, infinite # of CPUs, semaphores and monitors

Every major operating system built since 1985 has provided threads – Mach, OS/2, NT (Microsoft), Solaris (SUN), OSF (DEC/Compaq Alphas), and Linux. Why? Because makes it a lot easier to write concurrent programs, from Web servers, to databases, to embedded systems.

So does this mean you should all go out and use threads?

9.6 Cautionary Tale: OS/2

Illustrates why an abstraction doesn't always work the way you want it to.

Microsoft OS/2 (around 1988): initially, a spectacular failure. Since then, IBM has completely re-written it from scratch.

Used threads for everything – window systems, communication between programs, etc. Threads are a good idea, right?

Thus, system created lots of threads, but few actually running at any one time – most waiting around for user to type in a window, or for a network packet to arrive.

Might have 100 threads, but just a few at any one time on the ready queue. And each thread needs its own execution stack, say, 9KB, whether it is runnable or waiting.

Result: system needs an extra **1 MB** of memory, mostly consumed by waiting threads. 1 MB of memory cost $200 in 1988.

Put yourself in the customer's shoes. Did OS/2 run Excel or Word better? OK, it gave you the ability to keep working when you use the printer, but is that worth $200?

Moral: threads are cheap, but they're not free.

On the other hand: today 1MB of memory costs less than $2.50. The definition of what's cheap and what's expensive has changed!

Who are operating systems features for?

- Operating system developer?
- End user?

Lots of operating systems research has been focused on making it easier for operating systems **developers**, because it is so complicated to build operating systems.

But the trick to selling it is to make it better for the **end user**.

So, why might making things easier for the OS developer be advantageous to the end user and, more importantly, to the company selling the OS?

Things that make an OS slower will fail because the end user will see the slowness. Things that are neutral to the end user, but enable better OS development will succeed because of improved:

- time-to-market
- cost of maintenance