Thread Synchronization

Dickinson College
Computer Science 354
Spring 2006

slides courtesy of Professor Grant Braught

Road Map

- Past:
  - What an OS is, why we have them, what they do.
  - Base hardware and support for operating systems
  - Process Management
  - Process Scheduling
  - Multi-Threading
- Present:
  - Thread Synchronization
- Future:
  - Memory management
  - Storage management
  - Protection and Security

Thread Synchronization

- A motivating example
  - Explanation of example
  - Terminology
- Synchronization from user's perspective
  - The semaphore mechanism
  - Synchronization patterns
  - Synchronization problems
- Synchronization from the OS's perspective
  - Within the kernel
  - Providing synchronization primitives
  - Java synchronization

Motivating Example

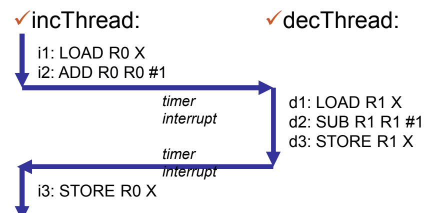
- What does the ThreadSynch program do?
- What output should be produced by the ThreadSynch program?
- What output is produced by the ThreadSynch program?

Inside ThreadSynchExample

- The machine code produced for the IncThread and DecThread would contain some code similar to the following.
  - IncThread:
    i1: LOAD R0 X
    i2: ADD R0 R0 #1
    i3: STORE R0 X
  - DecThread:
    d1: LOAD R1 X
    d2: SUB R1 R1 #1
    d3: STORE R1 X

ThreadSynchExample Execution

- Consider the following execution sequence for incThread and decThread:
  - Assume X is initially 0



Random OS Humor

- Windows Haiku Error Messages
  - Three things are certain:
    Death, taxes and lost data.
    Guess which has occurred.
  - Having been crashed
    The document you're seeking
    Must now be retyped.

Terminology

- Thread synchronization
  - Mutual Exclusion
    - Critical sections
  - Serialization
  - Race condition
  - Non-determinism

Thread Synchronization

- Recall that threads within a process execute *concurrently*.
- *Thread synchronization* is the process of imposing *synchronization constraints* on otherwise concurrently executing threads causing them to run:
  - One at a time (*mutual exclusion*)
  - In some predetermined order (*serialization*).

Mutual Exclusion

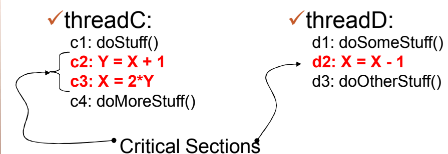
- A *mutual exclusion* constraint requires that two or more events be prevented from occurring concurrently.
  - Note that mutual exclusion does not impose a specific order on the events.
    - They may occur in any order, so long as they are forced to occur sequentially.

Mutual Exclusion Example

- Recall the ThreadSynchExample program:
  - IncThread:
    i1: LOAD R0 X
    i2: ADD R0 R0 #1
    i3: STORE R0 X
  - DecThread:
    d1: LOAD R1 X
    d2: SUB R1 R1 #1
    d3: STORE R1 X
    - A mutual exclusion constraint is necessary to ensure proper execution.

Critical Sections

- The sections of each thread that cannot be permitted to execute concurrently are called *critical sections*.
  - threadC:
    c1: doStuff()
    c2: Y = X + 1
    c3: X = 2*Y
    c4: doMoreStuff()
  - threadD:
    d1: doSomeStuff()
    d2: X = X - 1
    d3: doOtherStuff()



- Every section of code that involves an update to a shared variable should be treated as a critical section and should have a mutual exclusion constraint imposed upon it.

Serialization

- A *serialization* constraint requires that two or more events be forced to execute in a specific order.

Serialization Example

- Consider two threads, threadA that generates a value of X and threadB that uses the value of X to calculate the value of Y.
  - ✓ Assume: X=1, Y=0 are stored in the address space shared by the threads.
    - ✓ threadA:
      - a1: X = 5
      - a2: Print X
    - ✓ threadB:
      - b1: Y = X + 3
      - b2: Print Y
    - A serialization constraint is necessary in order to ensure proper execution.

Race Conditions

- A *race condition* is any situation in which the order of execution affects the final result.
  - ✓ The mutual exclusion and serialization examples both contained race conditions.

Non-Determinism

- In the absence of explicit synchronization constraints, the order of execution of concurrent threads can be non-deterministic.
  - ✓ The good
  - ✓ The bad
  - ✓ The ugly

Random OS Humor

- Alternative acronyms:
  - ✓ PCMCIA: People Can't Memorize Computer Industry Acronyms
  - ✓ SCSI: System Can't See It
  - ✓ DOS: Defective Operating System
  - ✓ BASIC: Bill's Attempt to Seize Industry Control

Thread Synchronization from the User's Perspective

Outline

- Synchronization mechanisms
  - ✓ Semaphore
  - ✓ Others
- Synchronization patterns
  - ✓ Signaling
  - ✓ Rendezvous
  - ✓ Mutex
  - ✓ Multiplex
  - ✓ Barrier
- Synchronization problems
  - ✓ Producer-Consumer
  - ✓ Readers-Writers
  - ✓ Unisex Bathroom

Semaphores

- A semaphore is an integer with three differences:
  - ✓ When created, its value must be initialized, thereafter the only operations allowed are increment and decrement.
  - ✓ When a thread decrements a semaphore, if the result is negative the thread is blocked on the semaphore.
  - ✓ When a thread increments a semaphore, if its value was negative then one of the threads blocked on the semaphore is woken up.

Semaphore Syntax

- Creating a semaphore with initial value 1:

```
mySem = new Semaphore(1);
```

- Decrementing a semaphore:

```
mySem.wait();
```

- Incrementing a semaphore:

```
mySem.signal();
```

Alternative Syntax

- Decrement:

```
mySem.decrement();
mySem.P();

mySem.decrementAndBlockIfResultIsNegative();
```

- Increment:

```
mySem.increment();
mySem.V();

mySem.incrementAndWakeAWaitingThreadIfAny();
```

Alternative Mechanisms

- In addition to semaphores there are a variety of other mechanisms that are also used for thread synchronization:
  - ✓ Locks
  - ✓ Monitors
    - ❖ Condition Variables

Synchronization Patterns

Signaling Pattern

- The signaling pattern can be used to enforce a serialization constraint on two threads.
  - E.g. Thread A must complete section a1 before thread B executes section b2.

Shared Data
1. mySem = new Semaphore(0);

Thread A	Thread B
1. *section a1*	1. *section b1*
2. mySem.signal();	2. mySem.wait();
3. *section a2*	3. *section b2*

Rendezvous Pattern

- Rendezvous generalizes signaling to work both ways.
  - E.g. Thread A must complete section a1 before thread B executes section b2 and thread B must complete section b1 before thread A completes section a2.
  - How can Rendezvous be implemented using semaphores?

Thread A	Thread B
1. *section a1*	1. *section b1*
2. *aReady.signal();*	2. *bReady.signal();*
3. *bReady.wait();*	3. *aReady.wait();*
4. *section a2*	4. *section b2*

Rendezvous Non-Solution

- What's wrong with the following proposed solution to the Rendezvous problem?

Shared Data
1. aReady = new Semaphore(0);
2. bReady = new Semaphore(0);

Thread A	Thread B
1. *section a1*	1. *section b1*
2. *bReady.wait();*	2. aReady.wait();
3. aReady.signal();	3. bReady.signal();
4. *section a2*	4. *section b2*

Mutex Pattern

- The mutex pattern can be used to enforce a mutual exclusion constraint on two threads.
  - E.g. We know that statement 2 in Thread A should not be executed concurrently with statement 2 in Thread B.
  - How can mutex be implemented using a semaphore?

Shared:
mutex = new Semaphore(1);

Thread A	Thread B
1. *section a1*	1. *section b1*
2. mutex.wait()	2. mutex.wait();
3. *X = X + 1;*	3. *X = X - 1;*
4. mutex.signal();	4. mutex.signal();

Multiplex Pattern

- The multiplex pattern generalizes the mutex pattern to allow only a fixed number of threads to execute specific sections of code concurrently.
  - E.g. Allow up to N copies of thread A to execute section a2 concurrently.
  - How can the multiplex pattern be implemented with a semaphore?

Shared:
multiplex = new Semaphore(N);

Thread A
1. *section a1*
2. multiplex.wait();
3. *section a2*
4. multiplex.signal();

Barrier Pattern

- The barrier pattern generalizes the rendezvous pattern to cases with N threads.
  - E.g. Each copy of thread A must wait for all N copies before executing section a2.
    - Could use one semaphore for each thread… but…

Thread A
1. *section a1*
2. *section a2*

Barrier Non-Solution #1

- What is wrong with the following proposed solution for the barrier problem?

```
Shared Data
1.  int N;
2.  int count = 0;
3.  mutex = new Semaphore(1);
4.  barrier = new Semaphore(0);
```

```
Thread A
1.  section a1
2.  mutex.wait();
3.    count++;
4.  mutex.signal();
5.  if (count == N)
6.    barrier.signal();
7.  barrier.wait();
8.  section a2
```

Barrier Solution

- The following code solves the Barrier problem.

```
Shared Data
1.  int N;
2.  int count = 0;
3.  mutex = new Semaphore(1);
4.  barrier = new Semaphore(0);
```

```
Thread A
1.  section a1
2.  mutex.wait();
3.    count++;
4.  mutex.signal();
5.  if (count == N)
6.    barrier.signal();
7.  barrier.wait();      <- Turnstile
8.  barrier.signal();    <- Turnstile
9.  section a2
```

Barrier Non-Solution #2

- What is wrong with the following proposed solution for the barrier problem?

```
Shared Data
1.  int N;
2.  int count = 0;
3.  mutex = new Semaphore(1);
4.  barrier = new Semaphore(0);
```

```
Thread A
1.  section a1
2.  mutex.wait();
3.    count++;
4.    if (count == N)
5.      barrier.signal();
6.    barrier.wait();
7.    barrier.signal();
8.  mutex.signal();
9.  section a2
```

Random OS Humor

Synchronization Problems

Producer-Consumer Problem

- Producer Threads: produce items and adds them to a shared data structure.
- Consumer Threads: remove items from a shared data structure and processes them.

```
Producer
1.  while (true)
2.    section p1
3.    pItem = generateItem();
4.    buffer.add(pItem);
5.    section p2
```

```
Consumer
1.  while (true)
2.    section c1
3.    cItem = buffer.get();
4.    processItem(cItem);
5.    section c2
```

Producer-Consumer Solution

Solution:
mutex = new Semaphore(1);
items = new Semaphore(0);

```
Producer:                         Consumer:

while (true)                      while (true)
    Section p1                        Section c1
    pItem = generateItem();           items.wait();
    mutex.wait();                     mutex.wait();
    buffer.add(pItem);                cItem = buffer.get();
    items.signal();                   mutex.signal();
    mutex.signal();                   processItem(cItem);
    Section p2                        Section c2
```

Producer-Consumer Non-Solution

- What's wrong with the following code for the Consumer threads?

Consumer
1. while (true)
2. *section c1*
3. mutex.wait();
4. items.wait();
5. cItem = buffer.get();
6. mutex.signal();
7. process(cItem);
8. *section c2*

Producer-Consumer with a Finite Buffer

- Often the buffer for a producer-consumer problem will have a practical limit on its size.
  - What synchronization constraint does this add?
  - How can we augment our solution to deal with a finite buffer?

Producer-Consumer with a Finite Buffer Solution

Solution:
mutex = new Semaphore(1);
items = new Semaphore(0);
Spaces = new Semaphore(N); // N = # of spaces in buffer.

```
Producer:                         Consumer:
while (true)                      while (true)
    Section p1                        Section c1
    pItem = generateItem();           items.wait();
    mutex.wait();                     mutex.wait();
    buffer.add(pItem);                cItem = buffer.get();
    items.signal();                   mutex.signal();
    mutex.signal();                   processItem(cItem);
    Section p2                        Section c2
```

Readers-Writers Problem

- Readers Threads: read information from a shared data structure (database / variable / file etc…)
- Writers Threads: write information to a shared data structure.

Writer:
1. *section w1*
2. *write data*
3. *section w2*

Reader:
1. *section r1*
2. *read data*
3. *section r2*

  - What are the synchronization constraints in this problem?

Readers-Writers Solution

Shared Data
1. readers = 0;
2. mutex = new semaphore(1);
3. roomEmpty = new semaphore(1);

Writers:
1. *section w1*
2. roomEmpty.wait();
3. *write data*
4. roomEmpty.signal();
5. *section w2*

Readers:
1. *section r1*
2. mutex.wait();
3. readers++;
4. if readers == 1
5. roomEmpty.wait();
6. mutex.signal();
7. *read data*
8. mutex.wait();
9. readers--;
10. if readers == 0
11. roomEmpty.signal();
12. mutex.signal();

Lightswitch Pattern

Shared Data

```
1.  inRoom = 0;
2.  mutex = new semaphore(1);
3.  light = new semaphore(1);
```

Thread A:

```
1.   section a1
2.   mutex.wait();
3.     inRoom++;
4.     if inRoom == 1   // first in…
5.       light.wait();  // turn on light if off (or block)
6.   mutex.signal();
7.   Critical Section
8.   mutex.wait();
9.     inRoom--;
10.    if inRoom == 0      // last out…
11.      light.signal();   // turn off light
12.  mutex.signal();
13.  section a2
```

Unisex Bathroom Problem

- A high-tech startup company can only afford space with a single bathroom with 3 stalls. A CS major working for the company proposes to solve the problem of allowing both men and women to use the bathroom using semaphores.
  - Synchronization constraints:
    - Men and women cannot be in the bathroom at the same time.
    - There should never be more than 3 people in the bathroom at once.

Man Thread:

```
1.  while (true)
2.    doWork();
3.    goToBathroom();
```

Woman Thread:

```
1.  while (true)
2.    doWork();
3.    goToBathroom();
```

Synchronization from the OS Perspective

Outline

- How do semaphores do what they do?
  - Basic semaphore structure
    - A new critical section!
  - Semaphore implementations
- Other synchronization mechanisms

Semaphore Structure

- A semaphore consists of an integer variable (value) and two methods for manipulating it.

```
semWait() {
  value = value - 1;
  if (value < 0) {
    block calling thread
  }
}
```

```
semSignal() {
  value = value + 1;
  if (value <= 0) {
    wakeup a thread
  }
}
```

  - Notice that each semaphore introduces a new critical section of it own!

Semaphore Implementations

- To implement a semaphore, it is necessary to ensure that execution of the critical sections is mutually exclusive.
  - Three possibilities:
    - In kernel mode by disabling interrupts.
    - With hardware support:
      - Using system calls for blocking.
      - Without any OS support.

Semaphores in Kernel Mode

- Semaphores can be implemented in kernel mode by disabling and enabling interrupts.

```
semWait() {
  disable interrupts
  value = value - 1;
  if (value < 0) {
    enable interrupts
    block calling thread
  }
  enable interrupts
}
```

```
semSignal() {
  disable interrupts
  value = value + 1;
  if (value <= 0) {
    wakeup a thread
  }
  enable interrupts
}
```

- ✓ Semaphore creation as well as semWait and semSignal must be system calls.

Hardware and Mutual Exclusion

- Additional instructions can be provided by the hardware to enable mutual exclusion without disabling interrupts.
  - ✓ One such instruction is TSL (test and set lock):

```
TSL addr  { reg = mm[addr]
          { mm[addr] = true
          { return reg
```

- ❖ TSL like all machine language instructions is executed *atomically*.

Mutual Exclusion with TSL

- The TSL instruction can be used to enforce mutual exclusion as follows:

Shared:

```
boolean lock = false;
int x = 0;
```

Thread A:

```
while(TSL(lock));
/* Critical section */
x = x + 1;
lock = false;
```

Thread B:

```
while(TSL(lock));
/* Critical section */
x = x - 1;
lock = false;
```

Semaphores with TSL

- Semaphores can be implemented using the TSL instruction to protect the critical sections.
  - ✓ Each semaphore now has both an integer variable value and a boolean variable lock.
  - ✓ System calls are required for blocking and waking up threads.

```
semWait() {
  while(TSL(lock));
  value = value - 1;
  if (value < 0) {
    add thread to semaphore queue
    lock = false;
    yield
  } else
    lock = false;
}
```

```
semSignal() {
  while(TSL(lock));
  value = value + 1;
  if (value <= 0) {
    wakeup a thread
  }
  lock = false;
}
```

Busy Waiting

- With busy waiting, a thread requires CPU cycles while waiting.
  - ✓ The semaphore implementation using TSL uses busy waiting while waiting for the lock.
  - ✓ This type of waiting is also called a *spin-lock*.

Semaphores with Busy-Waiting

- If an OS does not provide system calls for blocking threads, semaphores can be implemented entirely by using busy waiting.

```
semWait() {
  while (value <= 0);
  while(TSL(lock));
  value = value - 1;
  lock = false;
}
```

```
semSignal() {
  while(TSL(lock));
  value = value + 1;
  lock = false;
}
```

Random OS Humor

- "DOS computers manufactured by companies such as IBM, Compaq, Tandy, and millions of others are by far the most popular, with about 70 million machines in use worldwide. Macintosh fans, on the other hand, may note that cockroaches are far more numerous than humans, and that numbers alone do not denote a higher life form."

  — New York Times, November 26, 1991.

Other Synchronization Mechanisms

- Semaphores are only one mechanisms for enforcing synchronization constraints. There are several others:
  - ✓Locks
  - ✓Monitors
    - Synchronization in Java

Locks

- A lock is a mechanism for enforcing mutual exclusion.

```
Shared:
Lock mutex = new Lock();
int x = 0;

Thread A:
mutex.lock();
/* Critical section */
x = x + 1;
mutex.unlock();

Thread B:
mutex.lock();
/* Critical section */
x = x - 1;
mutex.unlock();
```

- ✓Locks can be implemented using:
  - Enable/disable interrupts and blocking
  - Busy-waiting with TSL

Monitors

- Code within a monitor may only be executed by a single thread at a time.

```
monitor <name> {
  <shared variables>
  initialization(<params>) {
    // initialization code
  }
  procedure P1(<params>) {
    // code in P1
  }
  procedure P2(<params>) {
    // code in P2
  }
  …
}
```

Condition Variables

- Condition variables are a mechanism by which threads executing within a monitor can block themselves and be woken up by other threads.
  - ✓Condition variables have two operations:
    - wait() - causes the calling thread to block.
    - signal() - wakes up a thread blocked on the condition variable.
      - Either the signaling thread or the woken thread must wait until the other exits the monitor before continuing.
      - If no threads are blocked on the condition variable then a signal has no effect and the signaling thread continues.

Semaphore via a Monitor

```
monitor Semaphore {
  int value;
  condition blocked;
  initialization(int initVal) {
    value = initVal;
  }
  procedure semWait() {
    value--;
    if (value < 0)
      blocked.wait();
  }
  procedure semSignal() {
    value++;
    if (value <= 0)
      blocked.signal();
  }
}
```

Java Synchronization

- Synchronization in Java is accomplished with a mechanism similar to a monitor.
  - Every object has a lock and a condition variable.
  - Methods can be declared synchronized.
    - Before a thread is permitted to execute a synchronized method, the thread must acquire the object's lock.
      - This happens automatically.
    - Threads may wait() and notify() the object's condition variable.
      - notify() is a signal().

Java Semaphore Implementation

```
public class Semaphore {
  private int value;
  public Semaphore(int initVal) {
    value = initVal;
  }
  public void synchronized semWait() {
    value--;
    if (value < 0)
      try {  wait();
      } catch(InterruptedException e) {}
  }
  public void synchronized semSignal() {
    value++;
    if (value <= 0)
      notify();
  }
}
```