

CS162  
Operating Systems and  
Systems Programming  
Lecture 6

Synchronization

September 19, 2005  
Prof. John Kubitowicz  
<http://inst.eecs.berkeley.edu/~cs162>

Review: ThreadFork() : Create a New Thread

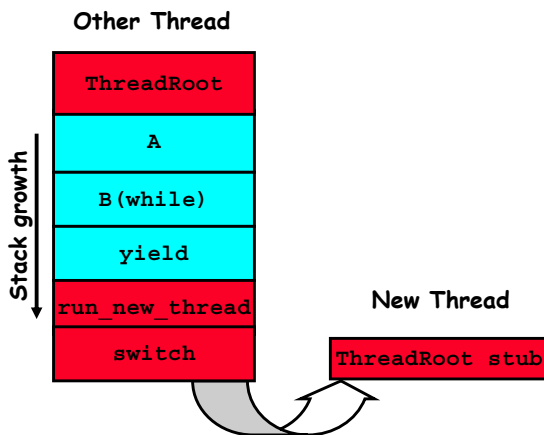
- ThreadFork() is a user-level procedure that creates a new thread and places it on ready queue
- Arguments to ThreadFork()
  - Pointer to application routine (fcnPtr)
  - Pointer to array of arguments (fcnArgPtr)
  - Size of stack to allocate
- Implementation
  - Sanity Check arguments
  - Enter Kernel-mode and Sanity Check arguments again
  - Allocate new Stack and TCB
  - Initialize TCB and place on ready list (Runnable).

9/19/05

Kubitowicz CS162 ©UCB Fall 2005

Lec 6.2

Review: How does Thread get started?



- Eventually, run\_new\_thread() will select this TCB and return into beginning of ThreadRoot()
  - This really starts the new thread

9/19/05

Kubitowicz CS162 ©UCB Fall 2005

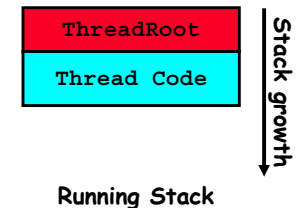
Lec 6.3

Review: What does ThreadRoot() look like?

- ThreadRoot() is the root for the thread routine:

```
ThreadRoot() {
    DoStartupHousekeeping();
    UserModeSwitch(); /* enter user mode */
    Call fcnPtr(fcnArgPtr);
    ThreadFinish();
}
```

- Startup Housekeeping
  - Includes things like recording start time of thread
  - Other Statistics
- Stack will grow and shrink with execution of thread
- Final return from thread returns into ThreadRoot() which calls ThreadFinish()
  - ThreadFinish() wake up sleeping threads



9/19/05

Kubitowicz CS162 ©UCB Fall 2005

Lec 6.4

## Goals for Today

- More concurrency examples
- Need for synchronization
- Examples of valid synchronization

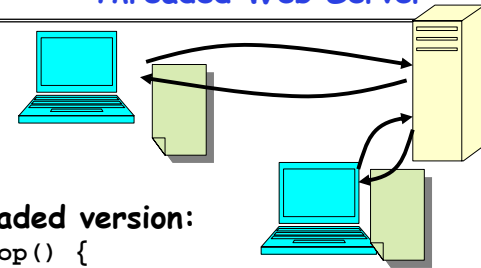
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.5

## Threaded Web Server



- Multithreaded version:

```
serverLoop() {  
    connection = AcceptCon();  
    ThreadFork(ServiceWebPage(), connection);  
}
```

- Advantages of threaded version:

- Can share file caches kept in memory, results of CGI scripts, other things
- Threads are *much* cheaper to create than processes, so this has a lower per-request overhead
- What if too many requests come in at once?

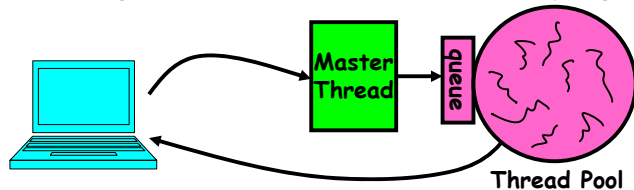
9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.6

## Thread Pools

- Problem with previous version: Unbounded Threads
  - When web-site becomes too popular - throughput sinks
- Instead, allocate a bounded "pool" of threads, representing the maximum level of multiprogramming



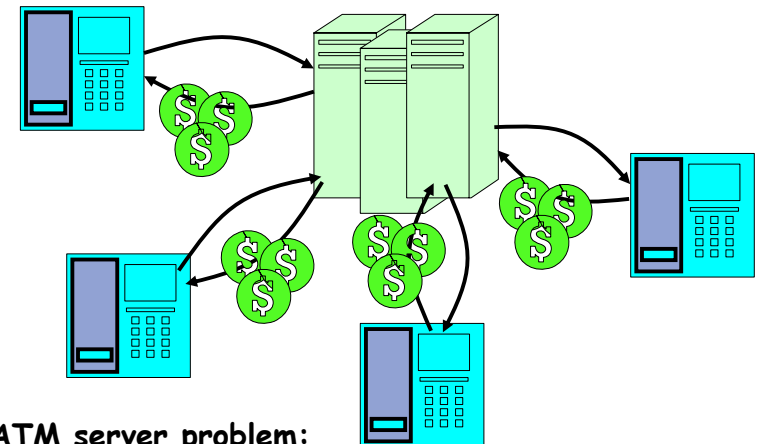
```
master() {  
    allocThreads(slave, queue);  
    while(TRUE) {  
        con=AcceptCon();  
        Enqueue(queue, con);  
        wakeUp(queue);  
    }  
}  
  
slave(queue) {  
    while(TRUE) {  
        con=Dequeue(queue);  
        if (con==null)  
            sleepOn(queue);  
        else  
            ServiceWebPage(con);  
    }  
}
```

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.7

## ATM Bank Server



- ATM server problem:

- Service a set of requests
- Do so without corrupting database
- Don't hand out too much money

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.8

## ATM bank server example

- Suppose we wanted to implement a server process to handle requests from an ATM network:

```
BankServer() {
    while (TRUE) {
        ReceiveRequest(&op, &acctId, &amount);
        ProcessRequest(op, acctId, amount);
    }
}

ProcessRequest(op, acctId, amount) {
    if (op == deposit) Deposit(acctId, amount);
    else if ...
}

Deposit(acctId, amount) {
    acct = GetAccount(acctId); /* may use disk I/O */
    acct->balance += amount;
    StoreAccount(acct); /* Involves disk I/O */
}
```

- How could we speed this up?
  - More than one request being processed at once
  - Event driven (overlap computation and I/O)
  - Multiple threads (multi-proc, or overlap comp and I/O)

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.9

## Event Driven Version of ATM server

- Suppose we only had one CPU
  - Still like to overlap I/O with computation
  - Without threads, we would have to rewrite in event-driven style
- Example

```
BankServer() {
    while(TRUE) {
        event = WaitForNextEvent();
        if (event == ATMRequest)
            StartOnRequest();
        else if (event == AcctAvail)
            ContinueRequest();
        else if (event == AcctStored)
            FinishRequest();
    }
}
```

- What if we missed a blocking I/O step?
- What if we have to split code into hundreds of pieces which could be blocking?
- This technique used for graphical programming

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.10

## Can Threads Make This Easier?

- Threads yield overlapped I/O and computation without “deconstructing” code into non-blocking fragments
  - One thread per request
- Requests proceeds to completion, blocking as required:

```
Deposit(acctId, amount) {
    acct = GetAccount(acctId); /* may use disk I/O */
    acct->balance += amount;
    StoreAccount(acct); /* Involves disk I/O */
}
```

- Unfortunately, shared state can get corrupted:

<u>Thread 1</u>	<u>Thread 2</u>
load r1, acct->balance	load r1, acct->balance
	add r1, amount2
	store r1, acct->balance
add r1, amount1	
store r1, acct->balance	

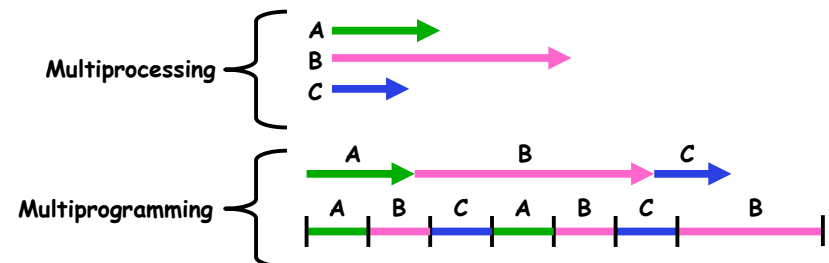
9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.11

## Review: Multiprocessing vs Multiprogramming

- What does it mean to run two threads “concurrently”?
  - Scheduler is free to run threads in any order and interleaving: FIFO, Random, ...
  - Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks



- Also recall: Hyperthreading
  - Possible to interleave threads on a per-instruction basis
  - Keep this in mind for our examples (like multiprocessing)

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.12

## Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A

x = 1;

Thread B

y = 2;

- However, What about (Initially, y = 12):

Thread A

x = 1;

x = y+1;

Thread B

y = 2;

y = y\*2;

- What are the possible values of x?
- (13, 5, 3)
- Or, what are the possible values of x below?

Thread A

x = 1;

Thread B

x = 2;

- X could be 1 or 2 (non-deterministic!)
- Could even be 3 for serial processors:
  - » Thread A writes 0001, B writes 0010.
  - » Scheduling order ABABABBA yields 3!

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.13

## Atomic Operations

- To understand a concurrent program, we need to know what the underlying indivisible operations are!
- **Atomic Operation:** an operation that always runs to completion or not at all
  - It is *indivisible*: it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block - if no atomic operations, than have no way for threads to work together
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
- Many instructions are not atomic
  - Double-precision floating point store often not atomic
  - VAX and IBM 360 had an instruction to copy a whole array

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.14

## Administrivia

- Sections in this class are mandatory
  - Make sure that you go to the section that you have been assigned
  - Some of the things presented in section will not show up in class!
- Should be working on first project
  - Make sure to be reading Nachos code
  - First design document due next Monday! (One week)
  - Set up regular meeting times with your group
  - Let's try to get group interaction problems figured out early
- If you need to know more about synchronization primitives before I get to them use book!
  - Chapter 6 (in 7<sup>th</sup> edition) and Chapter 7 (in 6<sup>th</sup> edition) are all about synchronization

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.15

## Correctness Requirements

- Threaded programs must work for all interleavings of thread instruction sequences
  - Cooperating threads inherently non-deterministic and non-reproducible
  - Really hard to debug unless carefully designed!
- Example: Therac-25
  - Machine for radiation therapy
    - » Software control of electron accelerator and electron→Xray production
    - » Software control of dosage
  - Software errors caused the death of several patients
    - » A series of race conditions on shared variables and poor software design
    - » "They determined that data entry speed during editing was the key factor in producing the error condition: If the prescription data was edited at a fast pace, the overdose occurred."

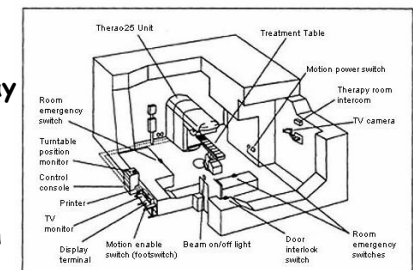


Figure 1. Typical Therac-25 facility

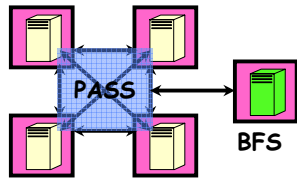
9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.16

## Space Shuttle Example

- Original Space Shuttle launch aborted 20 minutes before scheduled launch
- Shuttle has five computers:
  - Four run the "Primary Avionics Software System" (PASS)
    - Asynchronous and real-time
    - Runs all of the control systems
    - Results synchronized and compared every 3 to 4 ms
  - The Fifth computer is the "Backup Flight System" (BFS)
    - stays synchronized in case it is needed
    - Written by completely different team than PASS
- Countdown aborted because BFS disagreed with PASS
  - A 1/67 chance that PASS was out of sync one cycle
  - Bug due to modifications in **initialization** code of PASS
    - A delayed init request placed into timer queue
    - As a result, timer queue not empty at expected time to force use of hardware clock
  - Bug not found during extensive simulation



9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.17

## Another Concurrent Program Example

- Two threads, A and B, compete with each other
  - One tries to increment a shared counter
  - The other tries to decrement the counter

```

Thread A           Thread B
i = 0;             i = 0;
while (i < 10)     while (i > -10)
  i = i + 1;        i = i - 1;
printf("A wins!"); printf("B wins!");
    
```

- Assume that memory loads and stores are atomic, but incrementing and decrementing are **not** atomic
- Who wins? Could be either
- Is it guaranteed that someone wins? Why or why not?
- What if both threads have their own CPU running at same speed? Is it guaranteed that it goes on forever?

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.18

## Hand Simulation Multiprocessor Example

- Inner loop looks like this:

Thread A		Thread B	
r1=0	load r1, M[i]	r1=0	load r1, M[i]
r1=1	add r1, r1, 1	r1=-1	sub r1, r1, 1
M[i]=1	store r1, M[i]	M[i]=-1	store r1, M[i]

- Hand Simulation:**
  - And we're off. A gets off to an early start
  - B says "hmp, better go fast" and tries really hard
  - A goes ahead and writes "1"
  - B goes and writes "-1"
  - A says "HUH??? I could have sworn I put a 1 there"
- Could this happen on a uniprocessor?
  - Yes! Unlikely, but if you depending on it not happening, it will and your system will break.

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.19

## Motivation: "Too much milk"

- Great thing about OS's - analogy between problems in OS and problems in real life
  - Help you understand real life problems better
  - But, computers are much stupider than people
- Example: People need to coordinate:



Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.20

## Definitions

- **Synchronization**: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic
  - We are going to show that its hard to build anything useful with only reads and writes
- **Mutual Exclusion**: ensuring that only one thread does a particular thing at a time
  - One thread *excludes* the other while doing its task
- **Critical Section**: piece of code that only one thread can execute at once. Only one thread at a time will get into this section of code.
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two way of describing the same thing.

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.21

## More Definitions

- **Lock**: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
  - » **Important idea**: all synchronization involves waiting
- For example: fix the milk problem by putting a key on the refrigerator
  - Lock it and take key if you are going to go buy milk
  - Fixes too much: roommate angry if only wants OJ



- Of Course - We don't know how to make a lock yet

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.22

## Too Much Milk: Correctness Properties

- Need to be careful about correctness of concurrent programs, since non-deterministic
  - Always write down behavior first
  - Impulse is to start coding first, then when it doesn't work, pull hair out
  - Instead, think first, then code
- What are the correctness properties for the "Too much milk" problem??
  - Never more than one person buys
  - Someone buys if needed
- Restrict ourselves to use only atomic load and store operations as building blocks

9/19/05

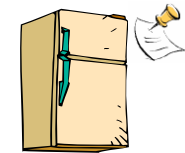
Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.23

## Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of "lock")
  - Remove note after buying (kind of "unlock")
  - Don't buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noMilk) {
    if (noNote) {
        leave Note;
        buy milk;
        remove note;
    }
}
```
- Result?
  - Still too much milk **but only occasionally!**
  - Thread can get context switched after checking milk and note but before buying milk!
- Solution makes problem worse since fails **intermittently**
  - Makes it really hard to debug...
  - **Must work despite what the dispatcher does!**



9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.24

## Too Much Milk: Solution #1 $\frac{1}{2}$

- Clearly the Note is not quite blocking enough
  - Let's try to fix this by placing note first
- Another try at previous solution:

```
leave Note;
if (noMilk) {
  if (noNote) {
    leave Note;
    buy milk;
  }
}
remove note;
```



- What happens here?
  - Well, with human, probably nothing bad
  - With computer: no one ever buys milk

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.25

## To Much Milk Solution #2

- How about labeled notes?
  - Now we can leave note before checking
- Algorithm looks like this:

### Thread A

```
leave note A;
if (noNote B) {
  if (noMilk) {
    buy Milk;
  }
}
remove note A;
```

### Thread B

```
leave note B;
if (noNoteA) {
  if (noMilk) {
    buy Milk;
  }
}
remove note B;
```

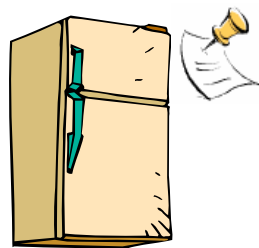
- Does this work?
- Possible for neither thread to buy milk
  - Context switches at exactly the wrong times can lead each to think that the other is going to buy
- Really insidious:
  - Extremely unlikely** that this would happen, but will at worse possible time
  - Probably something like this in UNIX

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.26

## Too Much Milk Solution #2: problem!



- I'm not getting milk, You're getting milk*
- This kind of lockup is called "starvation!"**

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.27

## Too Much Milk Solution #3

- Here is a possible two-note solution:

### Thread A

```
leave note A;
while (note B) {
  do nothing;
}
if (noMilk) {
  buy milk;
}
remove note A;
```

### Thread B

```
leave note B;
if (noNote A) {
  if (noMilk) {
    buy milk;
  }
}
remove note B;
```

- Does this work? **Yes**. Both can guarantee that:
  - It is safe to buy, or
  - Other will buy, ok to quit
- At A:
  - if no note B, safe for A to buy,
  - otherwise wait to find out what will happen
- At B:
  - if no note A, safe for B to buy
  - Otherwise, A is either buying or waiting for B to quit

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.28

### Solution #3 discussion

- Our solution protects a single "Critical-Section" piece of code for each thread:
 

```

      if (noMilk) {
        buy milk;
      }
      
```
- Solution #3 works, but it's really unsatisfactory
  - Really complex - even for this simple an example
    - » Hard to convince yourself that this really works
  - A's code is different from B's - what if lots of threads?
    - » Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    - » This is called "busy-waiting"
- There's a better way
  - Have hardware provide better (higher-level) primitives than atomic load and store
  - Build even higher-level programming abstractions on this new hardware support

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.29

### Too Much Milk: Solution #4

- Suppose we have some sort of implementation of a lock (more in a moment).
  - `Lock.Acquire()` - wait until lock is free, then grab
  - `Lock.Release()` - Unlock, waking up anyone waiting
  - These must be atomic operations - if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock
- Then, our milk problem is easy:
 

```

      milklock.Acquire();
      if (nomilk)
        buy milk;
      milklock.Release();
      
```
- Once again, section of code between `Acquire()` and `Release()` called a "Critical Section"
- Of course, you can make this even simpler: suppose you are out of ice cream instead of milk
  - Skip the test since you always need more ice cream.

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.30

### Where are we going with synchronization?

Programs	Shared Programs				
Higher-level API	Locks	Semaphores	Monitors	Send/Receive	
Hardware	Load/Store	Disable Ints	Test&Set	Comp&Swap	

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.31

### Summary

- Concurrent threads are a very useful abstraction
  - Allow transparent overlapping of computation and I/O
  - Allow use of parallel processing when available
- Concurrent threads introduce problems when accessing shared data
  - Programs must be insensitive to arbitrary interleavings
  - Without careful design, shared variables can become completely inconsistent
- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives
- Showed how to protect a critical section with only atomic load and store ⇒ pretty complex!

9/19/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 6.32