

CS162 Operating Systems and Systems Programming Lecture 5

Cooperating Threads

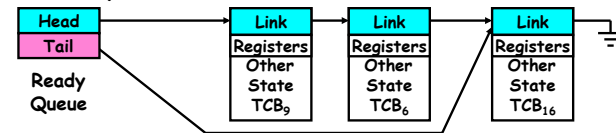
February 2, 2009

Ion Stoica

<http://inst.eecs.berkeley.edu/~cs162>

Review: Per Thread State

- Each Thread has a *Thread Control Block (TCB)*
 - Execution State: CPU registers, program counter, pointer to stack
 - Scheduling info: State (more later), priority, CPU time
 - Accounting Info
 - Various Pointers (for implementing scheduling queues)
 - Pointer to enclosing process? (PCB)?
 - Etc (add stuff as you find a need)
- OS Keeps track of TCBs in protected memory
 - In Arrays, or Linked Lists, or ...



2/2/10

CS162 ©UCB Fall 2009

Lec 5.2

Review: Yielding through Internal Events

- **Blocking on I/O**
 - The act of requesting I/O implicitly yields the CPU
- **Waiting on a "signal" from other thread**
 - Thread asks to wait and thus yields the CPU
- **Thread executes a yield()**
 - Thread volunteers to give up CPU

```
computePI() {
    while(TRUE) {
        ComputeNextDigit();
        yield();
    }
}
```

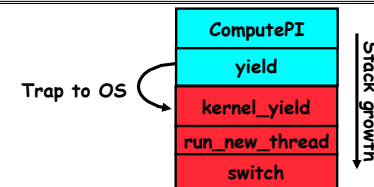
- Note that `yield()` must be called by programmer frequently enough!

2/2/10

CS162 ©UCB Fall 2009

Lec 5.3

Review: Stack for Yielding Thread



- How do we run a new thread?

```
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* Later in lecture */
}
```

- How does dispatcher switch to a new thread?
 - Save anything next thread may trash: PC, regs, stack
 - Maintain isolation for each thread

2/2/10

CS162 ©UCB Fall 2009

Lec 5.4

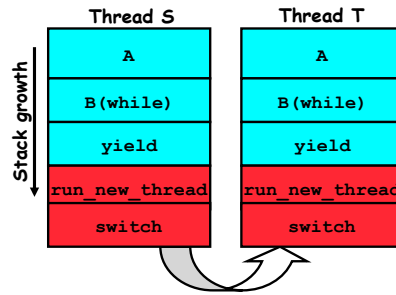
Review: Two Thread Yield Example

- Consider the following code blocks:

```

proc A() {
    B();
}
proc B() {
    while(TRUE) {
        yield();
    }
}
    
```

- Suppose we have 2 threads:
 - Threads S and T



2/2/10

CS162 ©UCB Fall 2009

Lec 5.5

Goals for Today

- More on Interrupts
- Thread Creation/Destruction
- Cooperating Threads

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiawicz.

2/2/10

CS162 ©UCB Fall 2009

Lec 5.6

- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
 - Mask enables/disables interrupts
 - Priority encoder picks highest enabled interrupt
 - Software Interrupt Set/Cleared by Software
 - Interrupt identity specified with ID line
- CPU can disable all interrupts with internal flag
- Non-maskable interrupt line (NMI) can't be disabled

2/2/10

CS162 ©UCB Fall 2009

Lec 5.7

Example: Network Interrupt

```

External Interrupt
{
    add $r1, $r2, $r3
    subi $r4, $r1, #4
    slli $r4, $r4, #2
    ...
    Transfer Network Packet from hardware to Kernel Buffers
    ...
    Restore registers
    Clear current Int
    Disable All Ints
    Restore priority
    RTI
}
    
```

Annotations:

- PC saved** (at the start of the handler)
- Disable All Ints** (at the start of the handler)
- Supervisor Mode** (at the start of the handler)
- Raise priority** (at the end of the handler)
- Reenable All Ints** (at the end of the handler)
- Save registers** (at the end of the handler)
- Dispatch to Handler** (at the end of the handler)
- Restore PC** (at the end of the handler)
- User Mode** (at the end of the handler)
- RTI** (at the end of the handler)

Other annotations:

- Pipeline Flush** (at the start of the handler)
- Interrupt Handler** (bracketed around the entire code block)

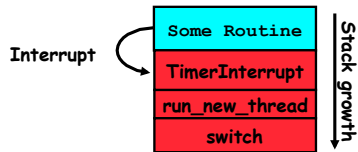
2/2/10

CS162 ©UCB Fall 2009

Lec 5.8

Review: Preemptive Multithreading

- Use the timer interrupt to force scheduling decisions



- Timer Interrupt routine:

```
TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
}
```
- This is often called **preemptive multithreading**, since threads are preempted for better scheduling
 - Solves problem of user who doesn't insert yield();

2/2/10

CS162 ©UCB Fall 2009

Lec 5.9

Administrivia

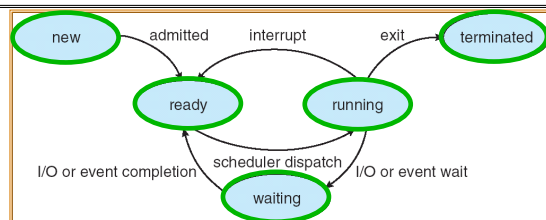
- Information about Subversion on Handouts page
 - Make sure to take a look
- Other things on Handouts page
 - Synchronization examples/Interesting papers
 - Previous finals/solutions
- Sections in this class are mandatory
 - Make sure that you go to the section that you have been assigned!
- Should be reading Nachos code by now!
 - Get working on the first project
 - Set up regular meeting times with your group
 - Try figure out group interaction problems early on

2/2/10

CS162 ©UCB Fall 2009

Lec 5.10

Review: Lifecycle of a Thread (or Process)



- As a thread executes, it changes state:
 - **new**: The thread is being created
 - **ready**: The thread is waiting to run
 - **running**: Instructions are being executed
 - **waiting**: Thread waiting for some event to occur
 - **terminated**: The thread has finished execution
- "Active" threads are represented by their TCBs
 - TCBs organized into queues based on their state

2/2/10

CS162 ©UCB Fall 2009

Lec 5.11

ThreadFork() : Create a New Thread

- ThreadFork() is a user-level procedure that creates a new thread and places it on ready queue
 - We called this CreateThread() earlier
- 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).

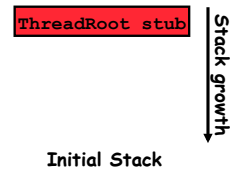
2/2/10

CS162 ©UCB Fall 2009

Lec 5.12

How do we initialize TCB and Stack?

- **Initialize Register fields of TCB**
 - Stack pointer made to point at stack
 - PC return address \Rightarrow OS (asm) routine ThreadRoot()
 - Two arg registers (a0 and a1) initialized to fcnPtr and fcnArgPtr, respectively
- **Initialize stack data?**
 - No. Important part of stack frame is in registers (ra)
 - Think of stack frame as just before body of ThreadRoot() really gets started

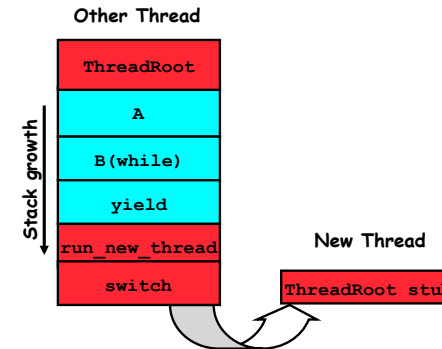


2/2/10

CS162 ©UCB Fall 2009

Lec 5.13

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

2/2/10

CS162 ©UCB Fall 2009

Lec 5.14

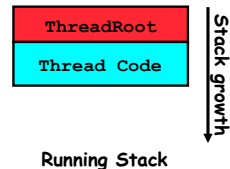
What does ThreadRoot() look like?

- 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() will start at user-level



2/2/10

CS162 ©UCB Fall 2009

Lec 5.15

What does ThreadFinish() do?

- Needs to re-enter kernel mode (system call)
- "Wake up" (place on ready queue) threads waiting for this thread
 - Threads (like the parent) may be on a wait queue waiting for this thread to finish
- Can't deallocate thread yet
 - We are still running on its stack!
 - Instead, record thread as "waitingToBeDestroyed"
- Call run_new_thread() to run another thread:


```
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping();
}
```

 - ThreadHouseKeeping() notices waitingToBeDestroyed and deallocates the finished thread's TCB and stack

2/2/10

CS162 ©UCB Fall 2009

Lec 5.16

Additional Detail

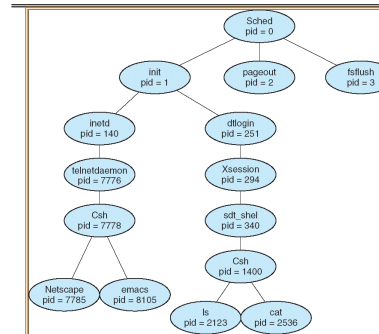
- Thread Fork is not the same thing as UNIX fork
 - UNIX fork creates a new *process* so it has to create a new address space
 - For now, don't worry about how to create and switch between address spaces
- Thread fork is very much like an asynchronous procedure call
 - Runs procedure in separate thread
 - Calling thread doesn't wait for finish
- What if thread wants to exit early?
 - ThreadFinish() and exit() are essentially the same procedure entered at user level

2/2/10

CS162 ©UCB Fall 2009

Lec 5.17

Parent-Child relationship



Typical process tree for Solaris system

- Every thread (and/or Process) has a parentage
 - A "parent" is a thread that creates another thread
 - A child of a parent was created by that parent

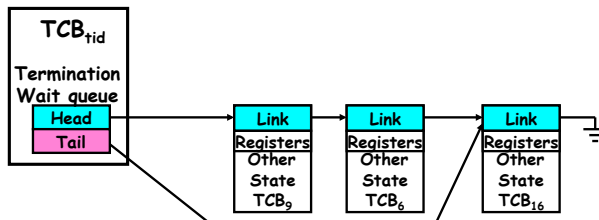
2/2/10

CS162 ©UCB Fall 2009

Lec 5.18

ThreadJoin() system call

- One thread can wait for another to finish with the ThreadJoin(tid) call
 - Calling thread will be taken off run queue and placed on waiting queue for thread tid
- Where is a logical place to store this wait queue?
 - On queue inside the TCB



- Similar to wait() system call in UNIX
 - Lets parents wait for child processes

2/2/10

CS162 ©UCB Fall 2009

Lec 5.19

Use of Join for Traditional Procedure Call

- A traditional procedure call is logically equivalent to doing a ThreadFork followed by ThreadJoin
- Consider the following normal procedure call of B() by A():

```
A() { B(); }
```

```
B() { Do interesting, complex stuff }
```

- The procedure A() is equivalent to A'():

```
A'() {
    tid = ThreadFork(B, null);
    ThreadJoin(tid);
}
```

- Why not do this for every procedure?
 - Context Switch Overhead
 - Memory Overhead for Stacks

2/2/10

CS162 ©UCB Fall 2009

Lec 5.20

Kernel versus User-Mode threads

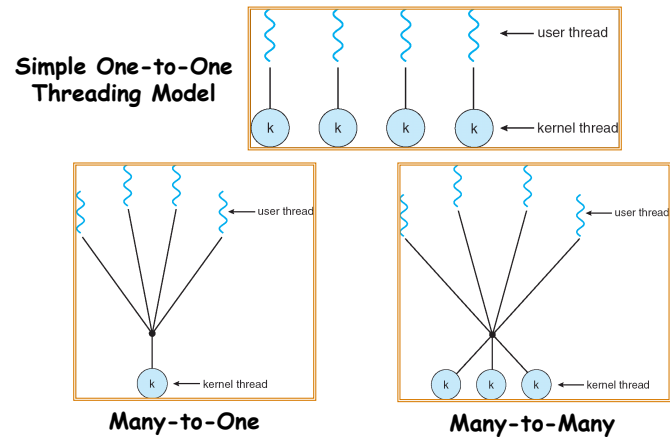
- We have been talking about Kernel threads
 - Native threads supported directly by the kernel
 - Every thread can run or block independently
 - One process may have several threads waiting on different things
- Downside of kernel threads: a bit expensive
 - Need to make a crossing into kernel mode to schedule
- Even lighter weight option: User Threads
 - User program provides scheduler and thread package
 - May have several user threads per kernel thread
 - User threads may be scheduled non-preemptively relative to each other (only switch on yield())
 - Cheap
- Downside of user threads:
 - When one thread blocks on I/O, all threads block
 - Kernel cannot adjust scheduling among all threads
 - Option: *Scheduler Activations*
 - » Have kernel inform user level when thread blocks...

2/2/10

CS162 ©UCB Fall 2009

Lec 5.21

Threading models mentioned by book



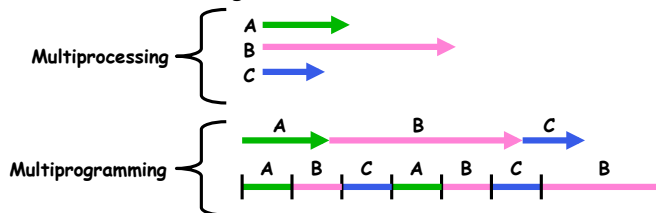
2/2/10

CS162 ©UCB Fall 2009

Lec 5.22

Multiprocessing vs Multiprogramming

- Remember Definitions:
 - Multiprocessing = Multiple CPUs
 - Multiprogramming = Multiple Jobs or Processes
 - Multithreading = Multiple threads per Process
- 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



2/2/10

CS162 ©UCB Fall 2009

Lec 5.23

Correctness for systems with concurrent threads

- If dispatcher can schedule threads in any way, programs must work under all circumstances
 - Can you test for this?
 - How can you know if your program works?
- Independent Threads:
 - No state shared with other threads
 - Deterministic \Rightarrow Input state determines results
 - Reproducible \Rightarrow Can recreate Starting Conditions, I/O
 - Scheduling order doesn't matter (if `switch()` works!!!)
- Cooperating Threads:
 - Shared State between multiple threads
 - Non-deterministic
 - Non-reproducible
- Non-deterministic and Non-reproducible means that bugs can be intermittent
 - Sometimes called "Heisenbugs"

2/2/10

CS162 ©UCB Fall 2009

Lec 5.24

Interactions Complicate Debugging

- Is any program truly independent?
 - Every process shares the file system, OS resources, network, etc
 - Extreme example: buggy device driver causes thread A to crash "independent thread" B
- You probably don't realize how much you depend on reproducibility:
 - Example: Evil C compiler
 - » Modifies files behind your back by inserting errors into C program unless you insert debugging code
 - Example: Debugging statements can overrun stack
- Non-deterministic errors are really difficult to find
 - Example: Memory layout of kernel+user programs
 - » depends on scheduling, which depends on timer/other things
 - » Original UNIX had a bunch of non-deterministic errors
 - Example: Something which does interesting I/O
 - » User typing of letters used to help generate secure keys

2/2/10

CS162 ©UCB Fall 2009

Lec 5.25

Why allow cooperating threads?

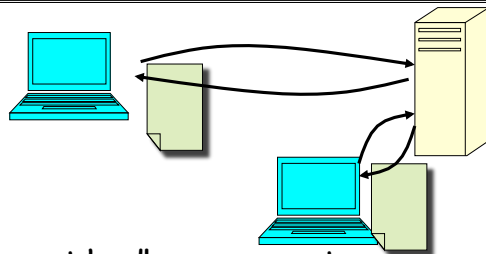
- People cooperate; computers help/enhance people's lives, so computers must cooperate
 - By analogy, the non-reproducibility/non-determinism of people is a notable problem for "carefully laid plans"
- Advantage 1: Share resources
 - One computer, many users
 - One bank balance, many ATMs
 - » What if ATMs were only updated at night?
 - Embedded systems (robot control: coordinate arm & hand)
- Advantage 2: Speedup
 - Overlap I/O and computation
 - » Many different file systems do read-ahead
 - Multiprocessors - chop up program into parallel pieces
- Advantage 3: Modularity
 - More important than you might think
 - Chop large problem up into simpler pieces
 - » To compile, for instance, gcc calls cpp | cc1 | cc2 | as | ld
 - » Makes system easier to extend

2/2/10

CS162 ©UCB Fall 2009

Lec 5.26

High-level Example: Web Server



- Server must handle many requests
- Non-cooperating version:

```
serverLoop() {
    con = AcceptCon();
    ProcessFork(ServiceWebPage(), con);
}
```
- What are some disadvantages of this technique?

2/2/10

CS162 ©UCB Fall 2009

Lec 5.27

Threaded Web Server

- Now, use a single process
- Multithreaded (cooperating) version:

```
serverLoop() {
    connection = AcceptCon();
    ThreadFork(ServiceWebPage(), connection);
}
```
- Looks almost the same, but has many advantages:
 - 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
- Question: would a user-level (say one-to-many) thread package make sense here?
 - When one request blocks on disk, all block...
- What about Denial of Service attacks or digg / Slash-dot effects?

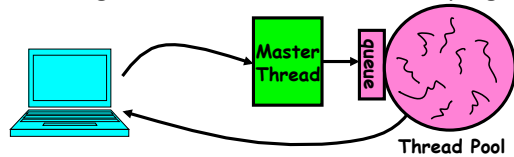
2/2/10

CS162 ©UCB



Thread Pools

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



```
master() {
    allocThreads(worker, queue);
    while (TRUE) {
        con=AcceptCon();
        Enqueue(queue, con);
        wakeUp(queue);
    }
}

worker(queue) {
    while (TRUE) {
        con=Dequeue(queue);
        if (con==null)
            sleepOn(queue);
        else
            ServiceWebPage(con);
    }
}
```

2/2/10 CS162 ©UCB Fall 2009 Lec 5.29

Summary

- Interrupts: hardware mechanism for returning control to operating system
 - Used for important/high-priority events
 - Can force dispatcher to schedule a different thread (preemptive multithreading)
- New Threads Created with ThreadFork()
 - Create initial TCB and stack to point at ThreadRoot()
 - ThreadRoot() calls thread code, then ThreadFinish()
 - ThreadFinish() wakes up waiting threads then prepares TCB/stack for destruction
- Threads can wait for other threads using ThreadJoin()
- Threads may be at user-level or kernel level
- Cooperating threads have many potential advantages
 - But: introduces non-reproducibility and non-determinism
 - Need to have Atomic operations

2/2/10

CS162 ©UCB Fall 2009

Lec 5.30