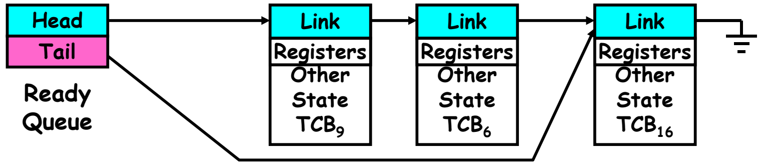
CS162
Operating Systems and Systems Programming
Lecture 5

Cooperating Threads

September 15, 2010
Prof. John Kubiatowicz
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 …



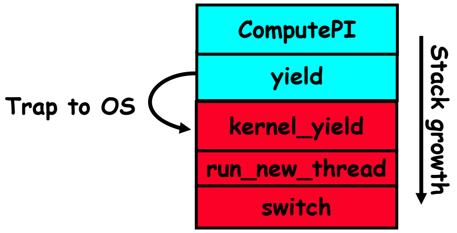
Review: Yielding through Internal Events

- 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!

Review: Stack for Yielding Thread



- How do we run a new thread?

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

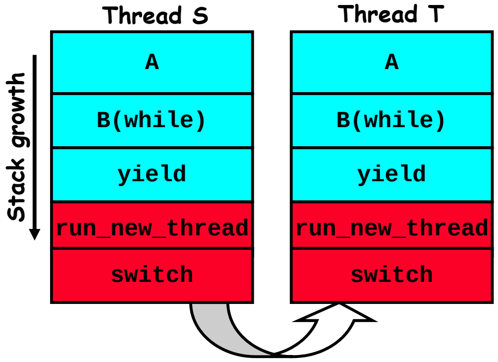
- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, stack
  - Maintain isolation for each thread

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

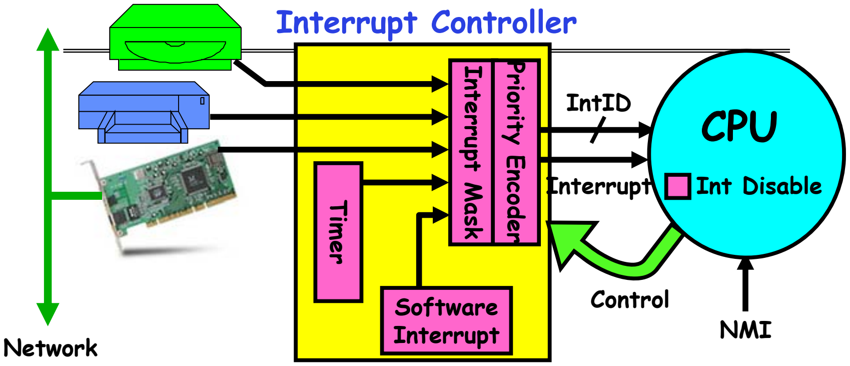


Goals for Today

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

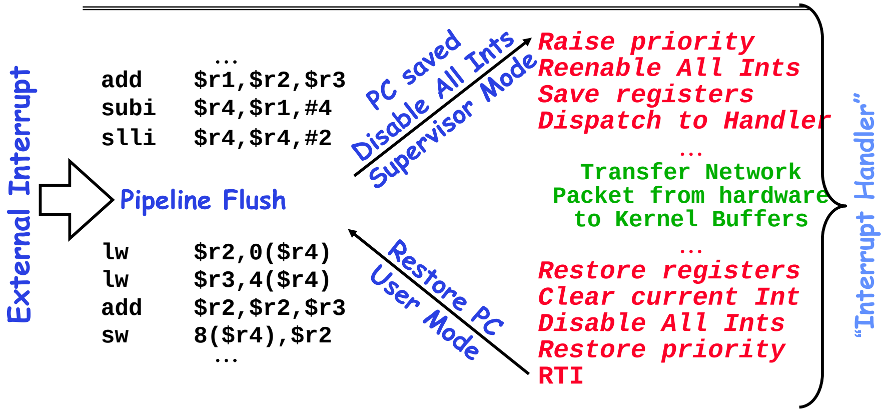
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Interrupt Controller



- 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

Example: Network Interrupt



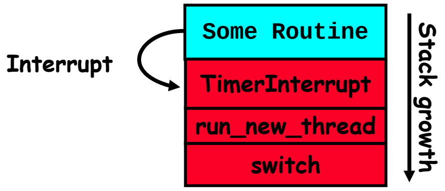
```
          ...
add     $r1,$r2,$r3
subi    $r4,$r1,#4
slli    $r4,$r4,#2

lw      $r2,0($r4)
lw      $r3,4($r4)
add     $r2,$r2,$r3
sw      8($r4),$r2
          ...
```

```
Raise priority
Reenable All Ints
Save registers
Dispatch to Handler
      ...
  Transfer Network
 Packet from hardware
  to Kernel Buffers
      ...
Restore registers
Clear current Int
Disable All Ints
Restore priority
RTI
```

- Disable/Enable All Ints ⇒ Internal CPU disable bit
  - RTI reenables interrupts, returns to user mode
- Raise/lower priority: change interrupt mask
- Software interrupts can be provided entirely in software at priority switching boundaries

Review: Preemptive Multithreading

- Use the timer interrupt to force scheduling decisions


Interrupt

Some Routine

TimerInterrupt

run_new_thread

switch

Stack growth

- Timer Interrupt routine:

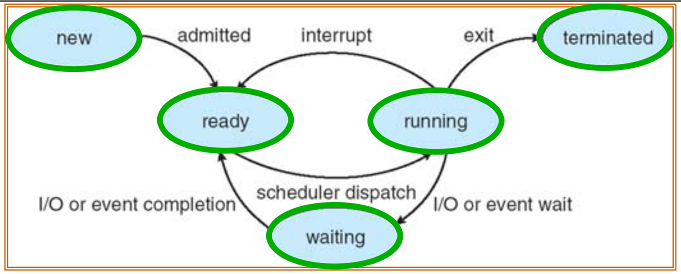
```
TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
}
```

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

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!
- Reader is available at Vics Copy on Hearst
  - Any problems getting copies of it?
- 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
- Chance that I cannot be here on Wednesday 9/29. However:
  - If this is true, Eric Brewer will take over for that lecture
  - He should be a great lecturer!

Review: Lifecycle of a Thread (or Process)


new

admitted

interrupt

exit

terminated

ready

running

scheduler dispatch

I/O or event completion

I/O or event wait

waiting

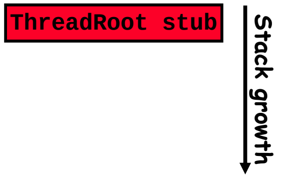
- As a thread executes, it changes state:
  - new: The thread is being created
  - ready: The thread is waiting to run
  - running: Instructions are being executed
  - waiting: Thread waiting for some event to occur
  - terminated: The thread has finished execution
- "Active" threads are represented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

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).

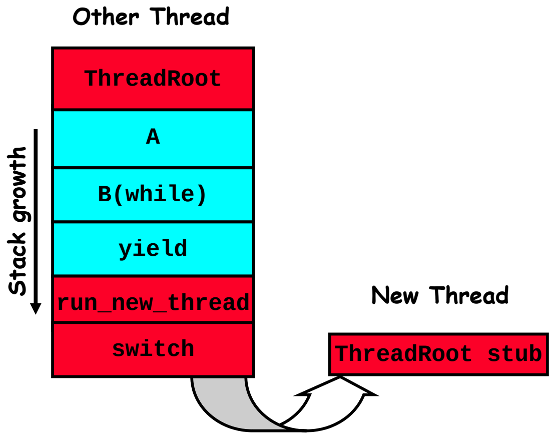
How do we initialize TCB and Stack?

- Initialize Register fields of TCB
  - Stack pointer made to point at stack
  - PC return address ⇒ 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



Initial Stack

How does Thread get started?

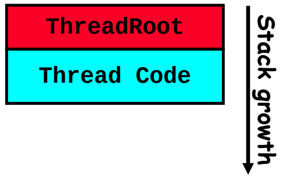


- Eventually, `run_new_thread()` will select this TCB and return into beginning of `ThreadRoot()`
  - This really starts the new thread

What does `ThreadRoot()` look like?

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

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



Running Stack

- 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

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:

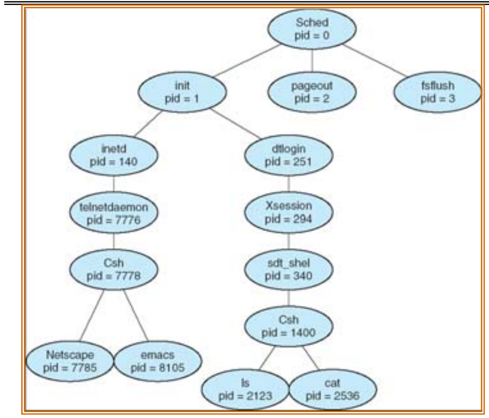
```
run_new_thread() {
   newThread = PickNewThread();
   switch(curThread, newThread);
   ThreadHouseKeeping();
}
```

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

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

Parent-Child relationship

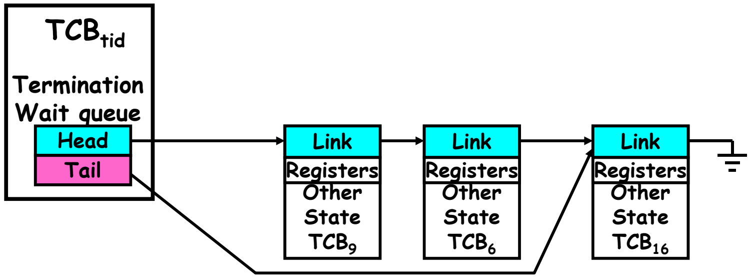


Typical process tree for Solaris system

- 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

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



TCB_{tid}

Termination Wait queue

Head

Tail

Link	Link	Link
Registers	Registers	Registers
Other State	Other State	Other State
TCB_9	TCB_6	TCB_{16}

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

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);
}
```

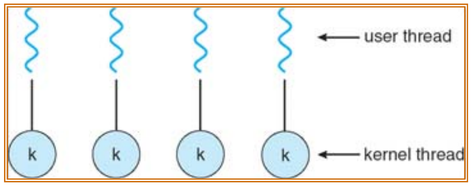
- Why not do this for every procedure?
  - Context Switch Overhead
  - Memory Overhead for Stacks

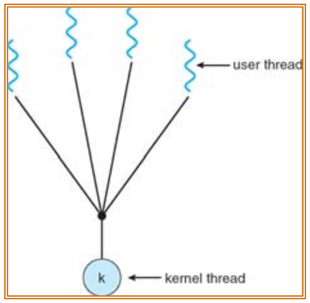
Kernel versus User-Mode threads

- 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-premptively 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…

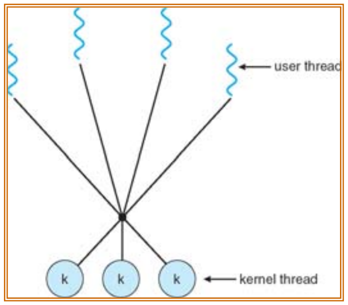
Threading models mentioned by book

Simple One-to-One Threading Model





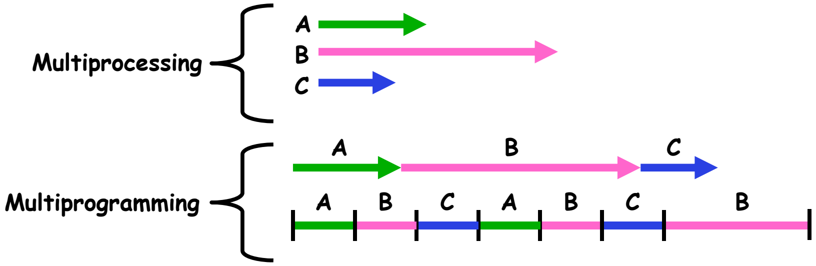
Many-to-One



Many-to-Many

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



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 ⇒ Input state determines results
  - Reproducible ⇒ 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"

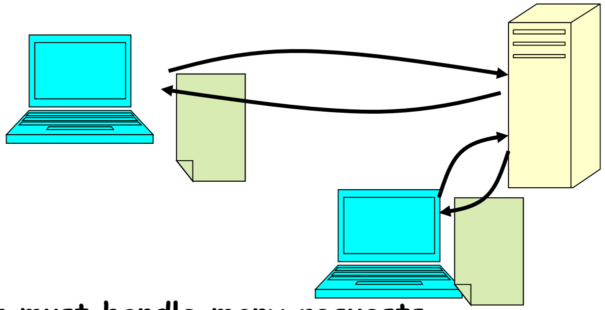
Interactions Complicate Debugging

- 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

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

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?

Threaded Web Server

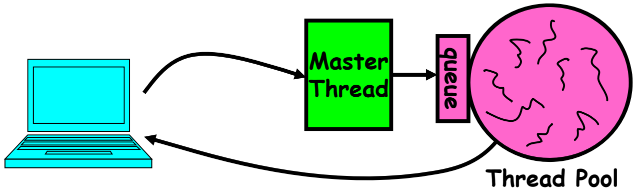
- Now, use a single process
- Multithreaded (cooperating) version:

```
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?

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);
   }
}
```

Summary

- Interrupts: hardware mechanism for returning control to operating system
  - Used for important/high-priority events
  - Can force dispatcher to schedule a different thread (premptive 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 distruction
- 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