

CS162 Operating Systems and Systems Programming Lecture 4

Thread Dispatching

September 12, 2005
Prof. John Kubiawicz
<http://inst.eecs.berkeley.edu/~cs162>

Recall: Modern Process with Multiple Threads

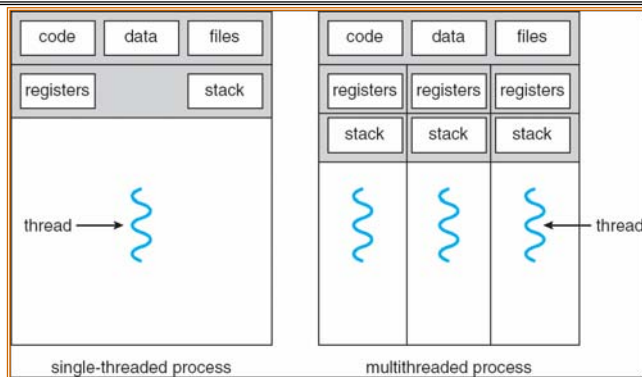
- Process: *Operating system abstraction to represent what is needed to run a single, multithreaded program*
- Two parts:
 - Multiple Threads
 - » Each thread is a *single, sequential stream of execution*
 - Protected Resources:
 - » Main Memory State (contents of Address Space)
 - » I/O state (i.e. file descriptors)
- Why separate the concept of a thread from that of a process?
 - Discuss the "thread" part of a process (concurrency)
 - Separate from the "address space" (Protection)
 - Heavyweight Process \equiv Process with one thread

9/12/05

Kubiawicz CS162 ©UCB Fall 2005

Lec 4.2

Recall: Single and Multithreaded Processes



- Threads encapsulate concurrency
 - "Active" component of a process
- Address spaces encapsulate protection
 - Keeps buggy program from trashing the system
 - "Passive" component of a process

9/12/05

Kubiawicz CS162 ©UCB Fall 2005

Lec 4.3

Recall: Classification

# threads Per AS:	# of addr spaces:	One	Many
One	One	MS/DOS, early Macintosh	Traditional UNIX
Many	One	Embedded systems (Geoworks, VxWorks, JavaOS, etc) JavaOS, Pilot(PC)	Mach, OS/2, Linux Windows 95??? Win NT to XP, Solaris, HP-UX, OS X

- Real operating systems have either
 - One or many address spaces
 - One or many threads per address space
- Did Windows 95/98/ME have real memory protection?
 - No: Users could overwrite process tables/System DLLs

9/12/05

Kubiawicz CS162 ©UCB Fall 2005

Lec 4.4

Goals for Today

- Further Understanding Threads
- Thread Dispatching
- Beginnings of Thread Scheduling

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

9/12/05

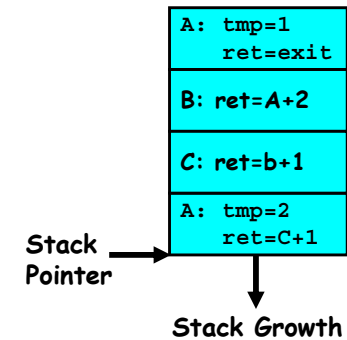
Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.5

Recall: Execution Stack Example

```

A(int tmp) {
    if (tmp<2)
        B();
    printf(tmp);
}
B() {
    C();
}
C() {
    A(2);
}
A(1);
    
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.6

MIPS: Software conventions for Registers

0	zero	constant 0	16	s0	callee saves
1	at	reserved for assembler	...		(callee must save)
2	v0	expression evaluation &	23	s7	
3	v1	function results	24	t8	temporary (cont'd)
4	a0	arguments	25	t9	
5	a1		26	k0	reserved for OS kernel
6	a2		27	k1	
7	a3		28	gp	Pointer to global area
8	t0	temporary: caller saves	29	sp	Stack pointer
...		(callee can clobber)	30	fp	frame pointer
15	t7		31	ra	Return Address (HW)

- Before calling procedure:
 - Save caller-saves regs
 - Save v0, v1
 - Save ra
- After return, assume
 - Callee-saves reg OK
 - gp,sp,fp OK (restored!)
 - Other things trashed

9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.7

Single-Threaded Example

- Imagine the following C program:

```

main() {
    ComputePI("pi.txt");
    PrintClassList("clist.text");
}
    
```

- What is the behavior here?
 - Program would never print out class list
 - Why? ComputePI would never finish

9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

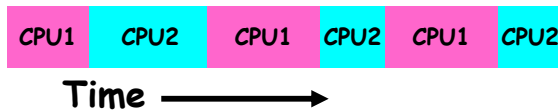
Lec 4.8

Use of Threads

Version of program with Threads:

```
main() {
    CreateThread(ComputePI("pi.txt"));
    CreateThread(PrintClassList("clist.text"));
}
```

- What does "CreateThread" do?
 - Start independent thread running given procedure
- What is the behavior here?
 - Now, you would actually see the class list
 - This *should* behave as if there are two separate CPUs



9/12/05

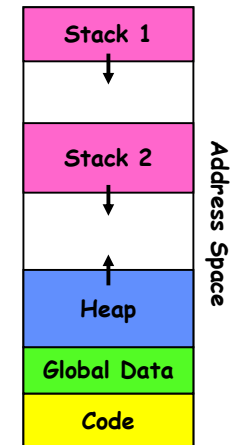
Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.9

Memory Footprint of Two-Thread Example

If we stopped this program and examined it with a debugger, we would see

- Two sets of CPU registers
- Two sets of Stacks
- Questions:
 - How do we position stacks relative to each other?
 - What maximum size should we choose for the stacks?
 - What happens if threads violate this?
 - How might you catch violations?



9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.10

Per Thread State

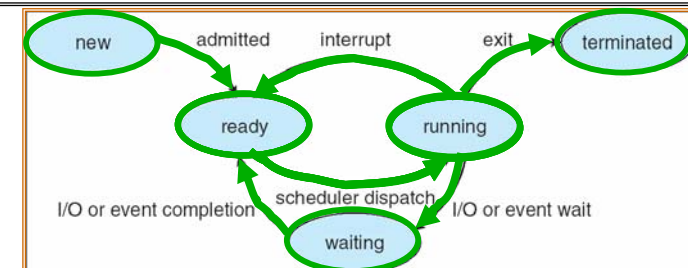
- Each Thread has a *Thread Control Block (TCB)*
 - Execution State: CPU registers, program counter, pointer to stack
 - Scheduling info: State (more later), priority, CPU time
 - Accounting Info
 - Various Pointers (for implementing scheduling queues)
 - Pointer to enclosing process? (PCB)?
 - Etc (add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

9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.11

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

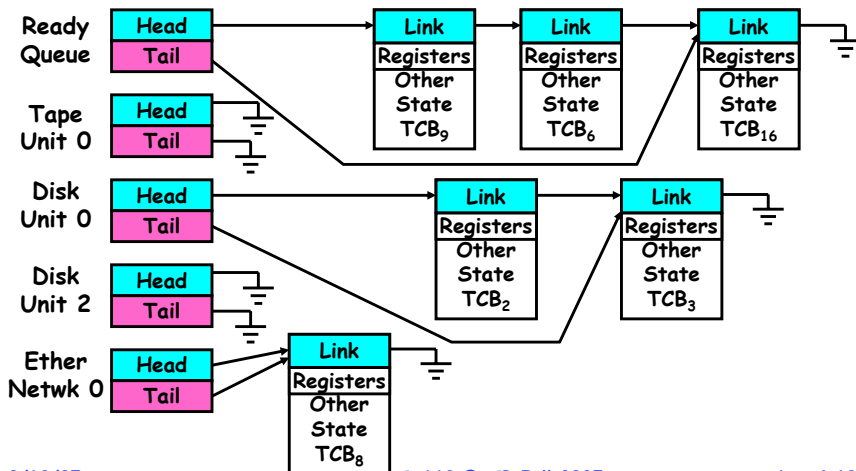
9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.12

Ready Queue And Various I/O Device Queues

- Thread not running \Rightarrow TCB is in some scheduler queue
 - Separate queue for each device/signal/condition
 - Each queue can have a different scheduler policy



9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.13

Administrivia

- Group assignments now posted on website
 - Check out the "Group/Section Assignment" link
 - Please attend your newly assigned section
- Nachos readers:
 - Available from Northside Copy Central
 - Includes printouts of all of the code
- Warning: you will be prompted for a passphrase
 - We need to autogenerate ssh keys for you
 - When prompted for a pass phrase, don't forget it!
 - This is needed for group collaboration tools
- Not everyone has run the register program!
 - This should happen automatically when you login, but you need to avoid hitting control-C
- Time to start Project 1
 - Go to Nachos page and start reading up
 - Start reading through the Nachos code (reader)

9/12/05

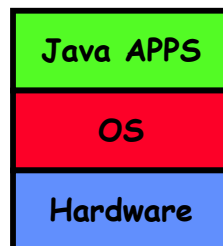
Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.14

Asside: Implementation Java OS

- Many threads, one Address Space
- Why another OS?
 - Recommended Minimum memory sizes:
 - » UNIX + X Windows: 32MB
 - » Windows 98: 16-32MB
 - » Windows NT: 32-64MB
 - » Windows 2000/XP: 64-128MB
 - What if want a cheap network point-of-sale computer?
 - » Say need 1000 terminals
 - » Want < 8MB
- What language to write this OS in?
 - C/C++/ASM? Not terribly high-level. Hard to debug.
 - Java/Lisp? Not quite sufficient - need direct access to HW/memory management

Java OS Structure



9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.15

Dispatch Loop

- Conceptually, the dispatching loop of the operating system looks as follows:

```

Loop {
    RunThread();
    ChooseNextThread();
    SaveStateOfCPU(curTCB);
    LoadStateOfCPU(newTCB);
}
    
```

- This is an *infinite* loop
 - One could argue that this is all that the OS does
- Should we ever exit this loop???
- When would that be?
- Emergency crash of operating system called "panic()"

9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.16

Running a thread

Consider first portion: `RunThread()`

- How do I run a thread?
 - Load its state (registers, PC, stack pointer) into CPU
 - Load environment (virtual memory space, etc)
 - Jump to the PC
- How does the dispatcher get control back?
 - Internal events: thread returns control voluntarily
 - External events: thread gets *preempted*

9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.17

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

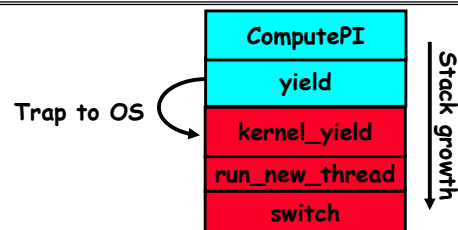
```
computePI() {
    while(TRUE) {
        ComputeNextDigit();
        yield();
    }
}
```

9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.18

Stack for Yielding Thread



- How do we run a new thread?

```
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* next Lecture */
}
```

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

9/12/05

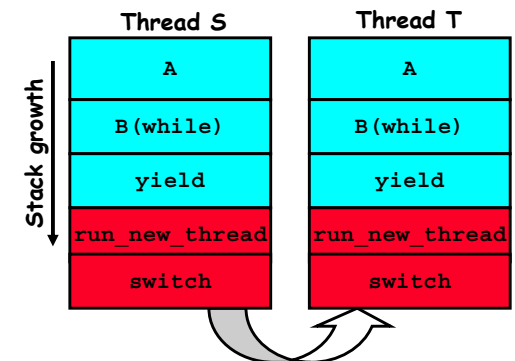
Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.19

What do the stacks look like?

- Consider the following code blocks:

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



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

9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.20

Saving/Restoring state (often called "Context Switch")

```
Switch(tCur,tNew) {
  /* Unload old thread */
  TCB[tCur].regs.r7 = CPU.r7;
  ...
  TCB[tCur].regs.r0 = CPU.r0;
  TCB[tCur].regs.sp = CPU.sp;
  TCB[tCur].regs.retpc = CPU.retpc; /*return addr*/

  /* Load and execute new thread */
  CPU.r7 = TCB[tNew].regs.r7;
  ...
  CPU.r0 = TCB[tNew].regs.r0;
  CPU.sp = TCB[tNew].regs.sp;
  CPU.retpc = TCB[tNew].regs.retpc;
  return; /* Return to CPU.retpc */
}
```

9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.21

Switch Details

- How many registers need to be saved/restored?
 - MIPS 4k: 32 Int(32b), 32 Float(32b)
 - Pentium: 14 Int(32b), 8 Float(80b), 8 SSE(128b),...
 - Sparc(v7): 8 Regs(32b), 16 Int regs (32b) * 8 windows = 136 (32b)+32 Float (32b)
 - Itanium: 128 Int (64b), 128 Float (82b), 19Other(64b)
- `retpc` is where the return should jump to.
 - In reality, this is implemented as a jump
- There is a real implementation of `switch` in Nachos.
 - See `switch.s`
 - » Normally, `switch` is implemented as assembly!
 - Of course, it's magical!
 - But you should be able to follow it!

9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.22

Switch Details (continued)

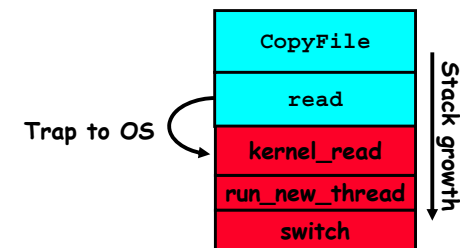
- What if you make a mistake in implementing `switch`?
 - Suppose you forget to save/restore reg 4
 - Get intermittent failures depending on when context switch occurred and whether new thread uses reg 4
 - System will give wrong result without warning
- Can you devise an exhaustive test to test `switch` code?
 - No! Too many combinations and inter-leavings
- Cautionary tail:
 - For speed, Topaz kernel saved one instruction in `switch()`
 - Carefully documented!
 - » Only works As long as kernel size < 1MB
 - What happened?
 - » Time passed, People forgot
 - » Later, they added features to kernel (noone removes features!)
 - » Very weird behavior started happening
 - Moral of story: Design for simplicity

9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.23

What happens when thread blocks on I/O?



- What happens when a thread requests a block of data from the file system?
 - User code invokes a system call
 - Read operation is initiated
 - Run new thread/switch
- Thread communication similar
 - Wait for Signal/Join
 - Networking

9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.24

External Events

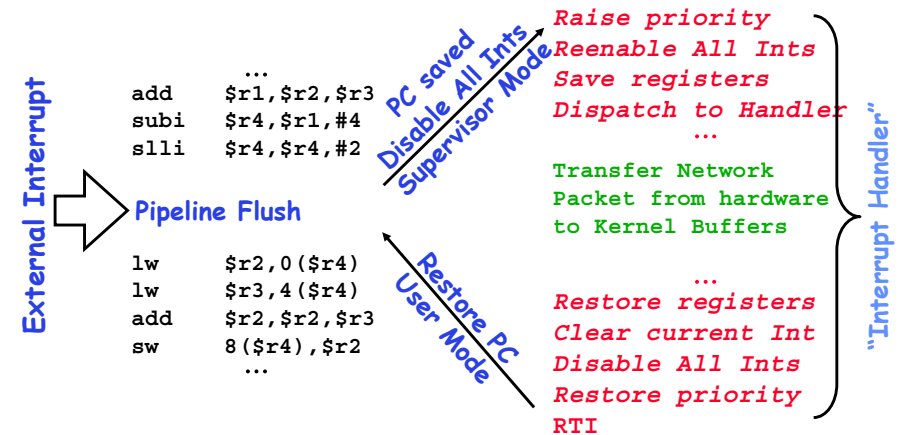
- What happens if thread never does any I/O, never waits, and never yields control?
 - Could the ComputePI program grab all resources and never release the processor?
 - » What if it didn't print to console?
 - Must find way that dispatcher can regain control!
- Answer: Utilize External Events
 - Interrupts: signals from hardware or software that stop the running code and jump to kernel
 - Timer: like an alarm clock that goes off every some many milliseconds
- If we make sure that external events occur frequently enough, can ensure dispatcher runs

9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.25

Example: Network Interrupt



- An interrupt is a hardware-invoked context switch
 - No separate step to choose what to run next
 - Always run the interrupt handler immediately

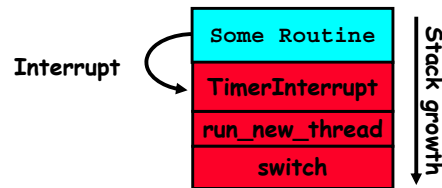
9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.26

Use of Timer Interrupt to Return Control

- Solution to our dispatcher problem
 - Use the timer interrupt to force scheduling decisions



- Timer Interrupt routine:


```

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

```
- I/O interrupt: same as timer interrupt except that DoHousekeeping() replaced by ServiceIO().

9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.27

Choosing a Thread to Run

- How does Dispatcher decide what to run?
 - Zero ready threads - dispatcher loops
 - » Alternative is to create an "idle thread"
 - » Can put machine into low-power mode
 - Exactly one ready thread - easy
 - More than one ready thread: use scheduling priorities
- Possible priorities:
 - LIFO (last in, first out):
 - » put ready threads on front of list, remove from front
 - Pick one at random
 - FIFO (first in, first out):
 - » Put ready threads on back of list, pull them from front
 - » This is fair and is what Nachos does
 - Priority queue:
 - » keep ready list sorted by TCB priority field

9/12/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 4.28

Summary

- **The state of a thread is contained in the TCB**
 - Registers, PC, stack pointer
 - States: New, Ready, Running, Waiting, or Terminated
- **Multithreading provides simple illusion of multiple CPUs**
 - Switch registers and stack to dispatch new thread
 - Provide mechanism to ensure dispatcher regains control
- **Switch routine**
 - Can be very expensive if many registers
 - Must be very carefully constructed!
- **Many scheduling options**
 - Decision of which thread to run complex enough for complete lecture