CS162
Operating Systems and Systems Programming
Lecture 3

Concurrency and Thread Dispatching

September 5, 2012

Ion Stoica

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

Goals for Today

- Review: Processes and Threads
- Thread Dispatching
- Cooperating Threads
- Concurrency examples

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

Why Processes & Threads?

Goals:

- **Multiprogramming:** Run multiple applications concurrently
- **Protection:** Don't want a bad application to crash system!

Solution:

Process: unit of execution and allocation

- Virtual Machine abstraction: give process illusion it owns machine (i.e., CPU, Memory, and IO device multiplexing)

Challenge:

- Process creation & switching expensive
- Need concurrency within same app (e.g., web server)

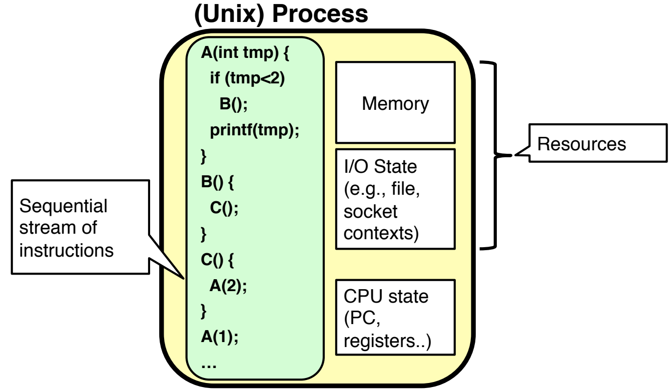
Solution:

Thread: Decouple allocation and execution

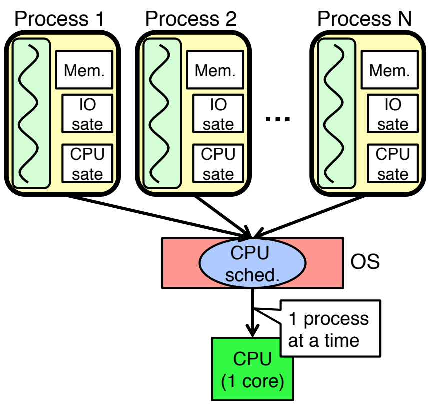
- Run multiple threads within same process

Putting it together: Process

(Unix) Process

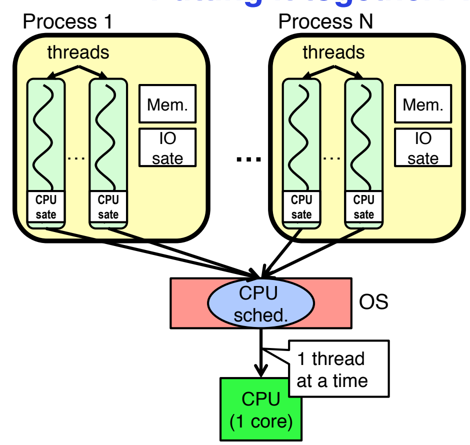


Putting it together: Processes



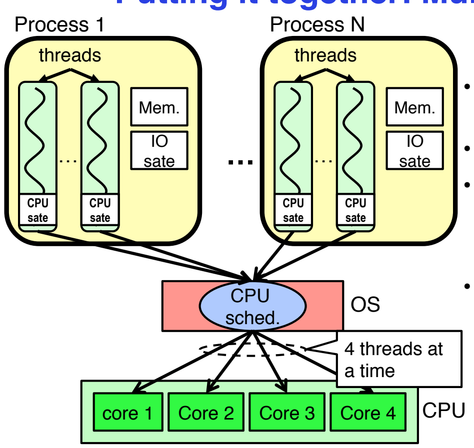
- Switch overhead: high
  - CPU state: low
  - Memory/IO state: high
- Process creation: high
- Protection
  - CPU: yes
  - Memory/IO: yes
- Sharing overhead: high (involves at least a context switch)

Putting it together: Threads



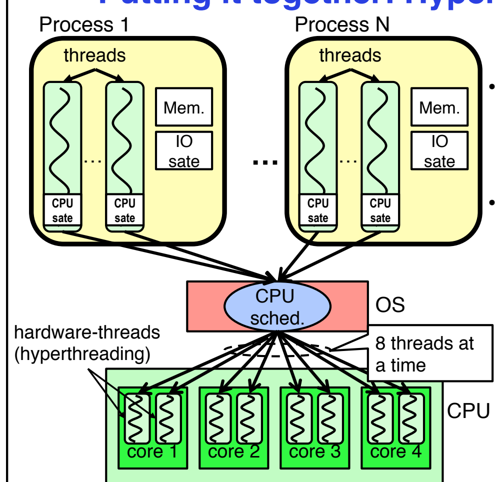
- Switch overhead: low (only CPU state)
- Thread creation: low
- Protection
  - CPU: yes
  - Memory/IO: No
- Sharing overhead: low (thread switch overhead low)

Putting it together: Multi-Cores



- Switch overhead: low (only CPU state)
- Thread creation: low
- Protection
  - CPU: yes
  - Memory/IO: No
- Sharing overhead: low (thread switch overhead low)

Putting it together: Hyper-Threading



- Switch overhead between hardware-threads: very-low (done in hardware)
- Contention to cache may hurt performance

Classification

# threads per AS: / # of addr spaces:	One	Many
One	MS/DOS, early Macintosh	Traditional UNIX
Many	Embedded systems (Geoworks, VxWorks, JavaOS,etc) JavaOS, Pilot(PC)	Mach, OS/2, Linux Win NT to 7, Solaris, HP-UX, OS X

- Real operating systems have either
  – One or many address spaces
  – One or many threads per address space

Thread State

- State shared by all threads in process/addr space
  – Content of memory (global variables, heap)
  – I/O state (file system, network connections, etc)
- State "private" to each thread
  – Kept in TCB ≡ Thread Control Block
  – CPU registers (including, program counter)
  – Execution stack – what is this?
- Execution Stack
  – Parameters, temporary variables
  – Return PCs are kept while called procedures are executing

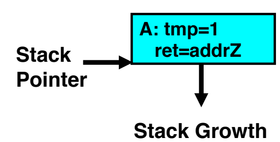
Review: Execution Stack Example

```
addrX:  A(int tmp) {
  .       if (tmp<2)
  .         B();
  .
addrY:    printf(tmp);
  .     }
  .     B() {
  .       C();
addrU:  }
  .     C() {
  .       A(2);
  .
addrV:  }
  .     A(1);
  .
addrZ:  exit;
```

- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

Review: Execution Stack Example

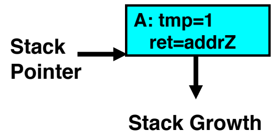
```
addrX:  A(int tmp) {
  .       if (tmp<2)
  .         B();
  .
addrY:    printf(tmp);
  .     }
  .     B() {
  .       C();
addrU:  }
  .     C() {
  .       A(2);
  .
addrV:  }
  .     A(1);
  .
addrZ:  exit;
```



- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

Review: Execution Stack Example

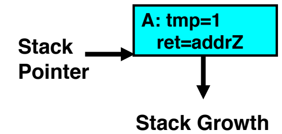
```
addrX: A(int tmp) {
  .      if (tmp<2)
  .        B();
addrY:   printf(tmp);
       }
  .    B() {
  .      C();
addrU: }
  .    C() {
  .      A(2);
addrV: }
  .    A(1);
addrZ: exit;
```



- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

Review: Execution Stack Example

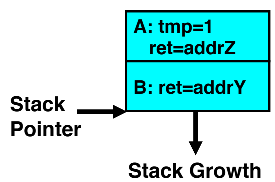
```
addrX: A(int tmp) {
  .      if (tmp<2)
  .        B();
addrY:   printf(tmp);
       }
  .    B() {
  .      C();
addrU: }
  .    C() {
  .      A(2);
addrV: }
  .    A(1);
addrZ: exit;
```



- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

Review: Execution Stack Example

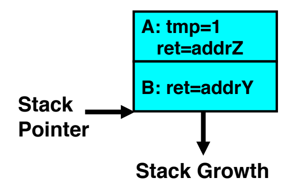
```
addrX: A(int tmp) {
  .      if (tmp<2)
  .        B();
addrY:   printf(tmp);
       }
  .    B() {
  .      C();
addrU: }
  .    C() {
  .      A(2);
addrV: }
  .    A(1);
addrZ: exit;
```



- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

Review: Execution Stack Example

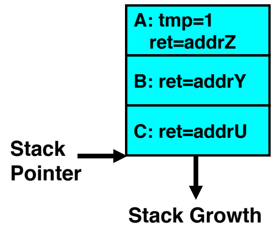
```
addrX: A(int tmp) {
  .      if (tmp<2)
  .        B();
addrY:   printf(tmp);
       }
  .    B() {
  .      C();
addrU: }
  .    C() {
  .      A(2);
addrV: }
  .    A(1);
addrZ: exit;
```



- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

Review: Execution Stack Example

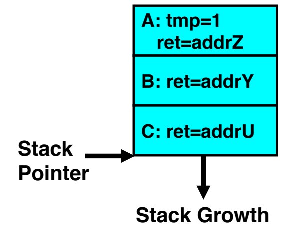
```
addrX:  A(int tmp) {
   .      if (tmp<2)
   .        B();
   .
addrY:    printf(tmp);
   .    }
   .    B() {
   .      C();
addrU:  }
   .    C() {
   .      A(2);
addrV:  }
   .    A(1);
addrZ:  exit;
```



- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

Review: Execution Stack Example

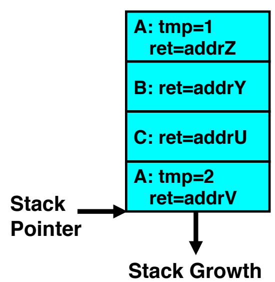
```
addrX:  A(int tmp) {
   .      if (tmp<2)
   .        B();
   .
addrY:    printf(tmp);
   .    }
   .    B() {
   .      C();
addrU:  }
   .    C() {
   .      A(2);
addrV:  }
   .    A(1);
addrZ:  exit;
```



- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

Review: Execution Stack Example

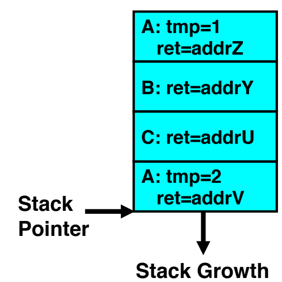
```
addrX:  A(int tmp) {
   .      if (tmp<2)
   .        B();
   .
addrY:    printf(tmp);
   .    }
   .    B() {
   .      C();
addrU:  }
   .    C() {
   .      A(2);
addrV:  }
   .    A(1);
addrZ:  exit;
```



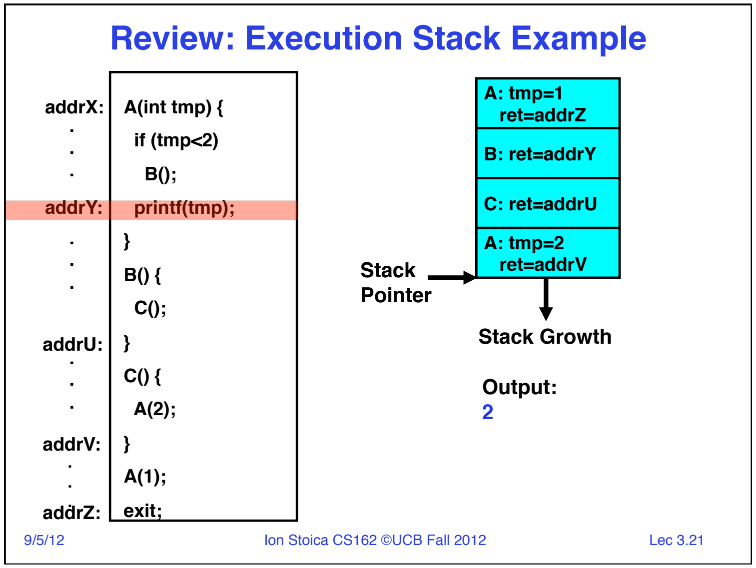
- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

Review: Execution Stack Example

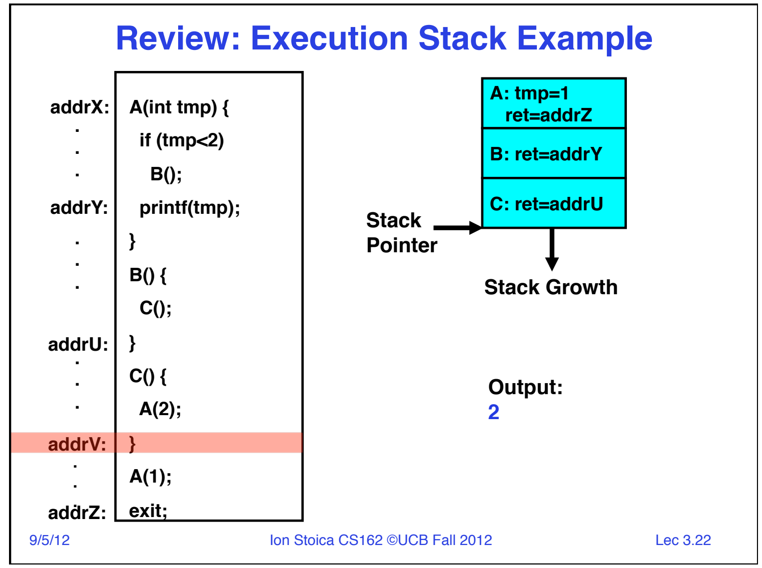
```
addrX:  A(int tmp) {
   .      if (tmp<2)
   .        B();
   .
addrY:    printf(tmp);
   .    }
   .    B() {
   .      C();
addrU:  }
   .    C() {
   .      A(2);
addrV:  }
   .    A(1);
addrZ:  exit;
```



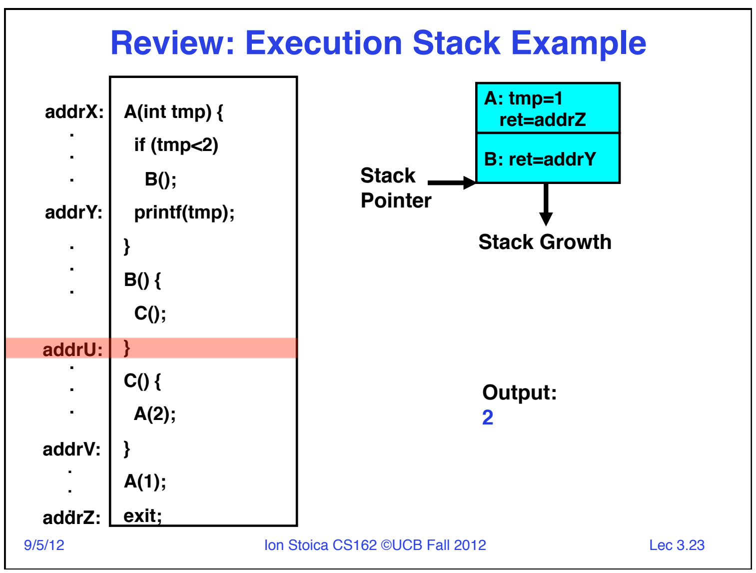
- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages



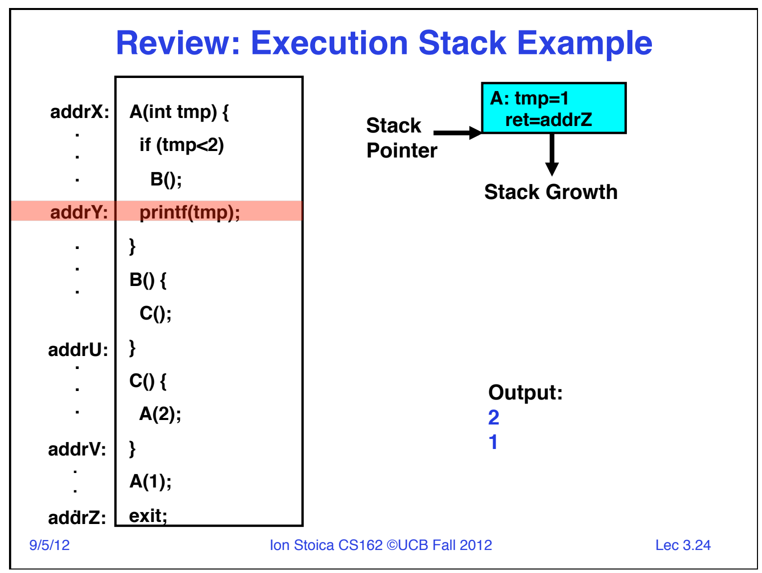
Review: Execution Stack Example
addrX: A(int tmp) {
if (tmp<2)
B();
addrY: printf(tmp);
}
B() {
C();
addrU: }
C() {
A(2);
addrV: }
A(1);
addrZ: exit;
A: tmp=1 ret=addrZ
B: ret=addrY
C: ret=addrU
A: tmp=2 ret=addrV
Stack Pointer
Stack Growth
Output:
2
9/5/12
Ion Stoica CS162 ©UCB Fall 2012
Lec 3.21



Review: Execution Stack Example
addrX: A(int tmp) {
if (tmp<2)
B();
addrY: printf(tmp);
}
B() {
C();
addrU: }
C() {
A(2);
addrV: }
A(1);
addrZ: exit;
A: tmp=1 ret=addrZ
B: ret=addrY
C: ret=addrU
Stack Pointer
Stack Growth
Output:
2
9/5/12
Ion Stoica CS162 ©UCB Fall 2012
Lec 3.22



Review: Execution Stack Example
addrX: A(int tmp) {
if (tmp<2)
B();
addrY: printf(tmp);
}
B() {
C();
addrU: }
C() {
A(2);
addrV: }
A(1);
addrZ: exit;
A: tmp=1 ret=addrZ
B: ret=addrY
Stack Pointer
Stack Growth
Output:
2
9/5/12
Ion Stoica CS162 ©UCB Fall 2012
Lec 3.23



Review: Execution Stack Example
addrX: A(int tmp) {
if (tmp<2)
B();
addrY: printf(tmp);
}
B() {
C();
addrU: }
C() {
A(2);
addrV: }
A(1);
addrZ: exit;
A: tmp=1 ret=addrZ
Stack Pointer
Stack Growth
Output:
2
1
9/5/12
Ion Stoica CS162 ©UCB Fall 2012
Lec 3.24

Review: Execution Stack Example

```
addrX:  A(int tmp) {
   .      if (tmp<2)
   .        B();
   .
addrY:    printf(tmp);
   .    }
   .    B() {
   .      C();
addrU:  }
   .    C() {
   .      A(2);
   .
addrV:  }
   .    A(1);
   .
addrZ:  exit;
```

Output:
2
1

Single-Threaded Example

- Imagine the following C program:

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

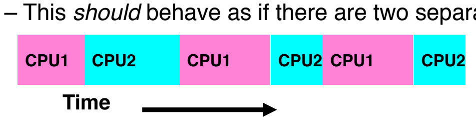
- What is the behavior here?
  - Program would never print out class list
  - Why? ComputePI would never finish

Use of Threads

- Version of program with Threads:

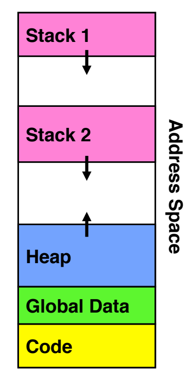
```
main() {
    CreateThread(ComputePI("pi.txt"));
    CreateThread(PrintClassList("clist.text"));
}
```

- 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



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?



Announcements

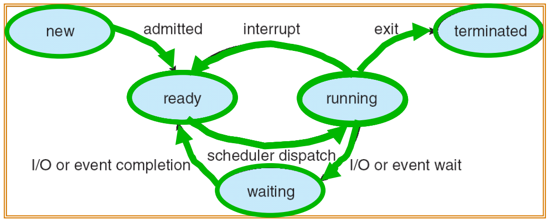
- Section assignments posted on Piazza
  - Attend new sections THIS week
  - Email cs162@cory if you don't have a group!

5min Break

Per Thread State

- Each Thread has a *Thread Control Block* (TCB)
  - Execution State: CPU registers, program counter (PC), pointer to stack (SP)
  - Scheduling info: state, priority, CPU time
  - 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 Array, or Linked List, or …

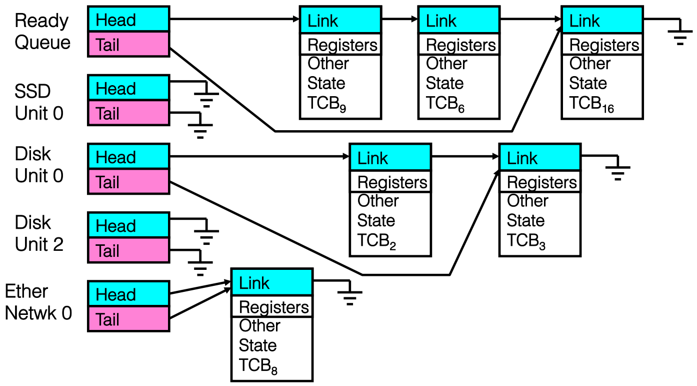
Lifecycle of a Thread (or Process)



- 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

Ready Queue And Various I/O Device Queues

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



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

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*

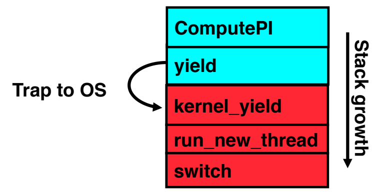
Yielding through Internal Events

- 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

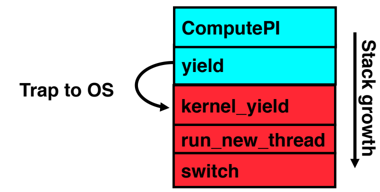


- How do we run a new thread?

```
run_new_thread() {
  newThread = PickNewThread();
  switch(curThread, newThread);
  ThreadHouseKeeping(); /* deallocates finished threads */
}
```

- Finished thread not killed right away. Why?
  – Move them in "exit/terminated" state
  – ThreadHouseKeeping() deallocates finished threads

Review: Stack for Yielding Thread



- How do we run a new thread?

```
run_new_thread() {
  newThread = PickNewThread();
  switch(curThread, newThread);
  ThreadHouseKeeping(); /* deallocates finished threads */
}
```

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

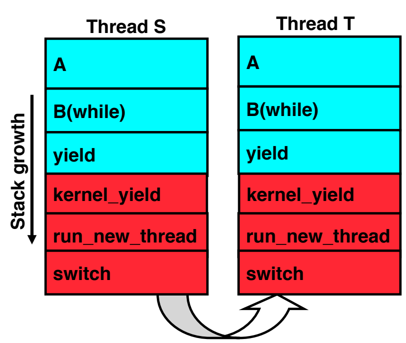
Review: Two Thread Yield Example

- Consider the following code blocks:

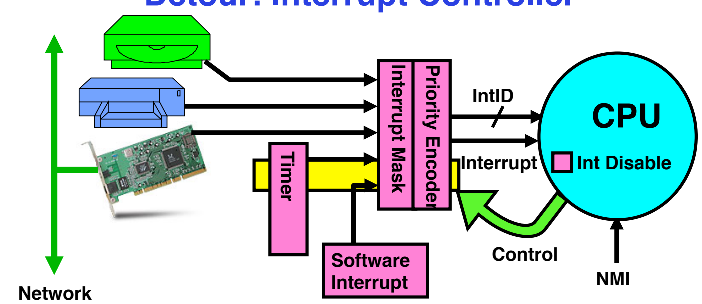
```
proc A() {
   B();

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

- Suppose we have two threads:
  – Threads S and T



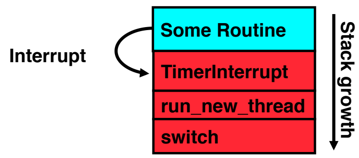
Detour: Interrupt Controller



- 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

Review: Preemptive Multithreading

- Use the timer interrupt to force scheduling decisions



- Timer Interrupt routine:

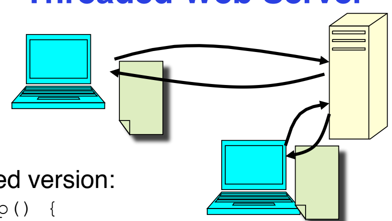
```
TimerInterrupt() {
   DoPeriodicHouseKeeping();
   run_new_thread();
}
```

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

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
  - Multiprocessors – chop up program into parallel pieces
- Advantage 3: Modularity
  - Chop large problem up into simpler pieces
    - » To compile, for instance, gcc calls cpp | cc1 | cc2 | as | ld
    - » Makes system easier to extend

Threaded Web Server



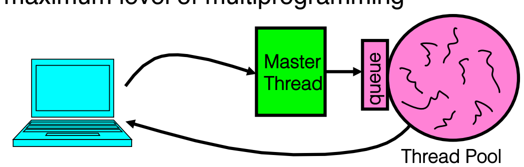
- Multithreaded version:

```
serverLoop() {
      connection = AcceptCon();
      ThreadCreate(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?

Thread Pools

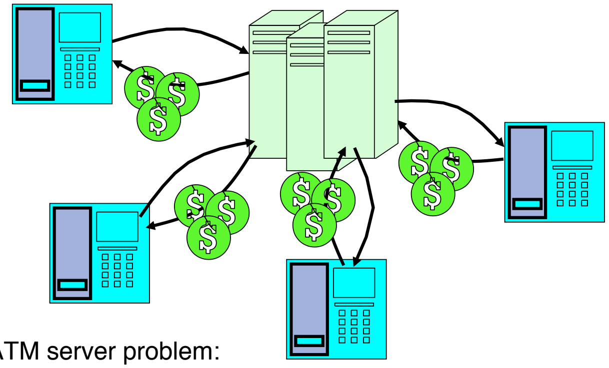
- Problem with previous version: Unbounded Threads
  - When web-site becomes too popular – throughput sinks
- Instead, allocate a bounded "pool" of threads, representing the maximum level of multiprogramming




```
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);
   }
}
```

ATM Bank Server



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

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
  - Multiple threads (multi-proc, or overlap comp and I/O)

Can Threads Help?

- One thread per request!

- Requests proceeds to completion, blocking as required:

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

- Unfortunately, shared state can get corrupted:

```
       Thread 1                          Thread 2
load r1, acct->balance
                                  load r1, acct->balance
                                  add r1, amount2
                                  store r1, acct->balance
add r1, amount1
store r1, acct->balance
```


Problem is at the lowest level

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

Thread A	Thread B
x = 1;	y = 2;

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

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

  - What are the possible values of x?

Thread A	Thread B
x = 1;	
x = y+1;	
	y = 2;
	y = y*2

x=13

Problem is at the lowest level

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

Thread A	Thread B
x = 1;	y = 2;

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

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

  – What are the possible values of x?

Thread A	Thread B
	y = 2;
	y = y*2;
x = 1;	
x = y+1;	

x=5

Problem is at the lowest level

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

Thread A	Thread B
x = 1;	y = 2;

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

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

  – What are the possible values of x?

Thread A	Thread B
	y = 2;
x = 1;	
x = y+1;	
	y= y*2;

x=3

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
- Next lecture: deal with concurrency problems