CS162
Operating Systems and Systems Programming
Lecture 4

Processes (con't), Threads, Concurrency

January 30th, 2020

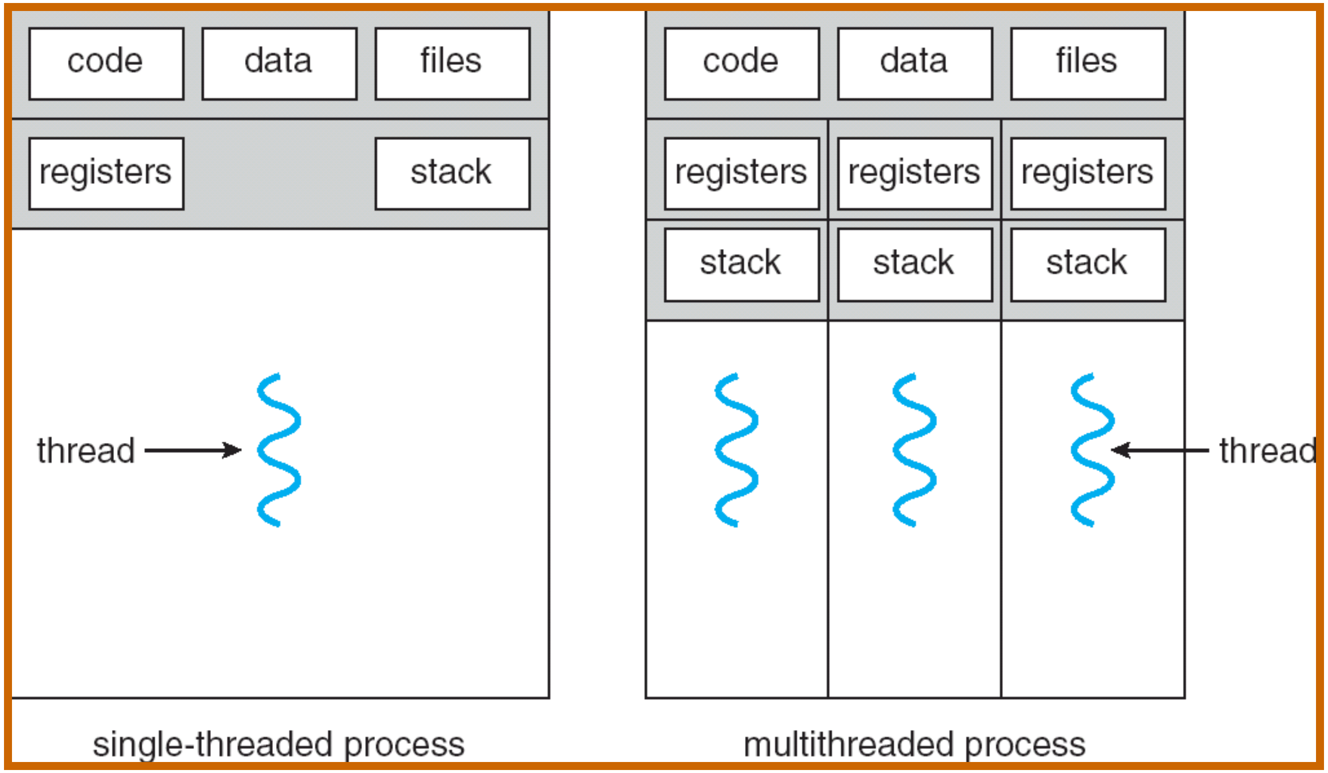
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiatowicz at Berkeley, with few minor updates/changes*. When slides are obtained from other sources, a a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.*

Recall: Modern Process with Threads

- **Process:** execution environment with restricted rights
  - **Address Space with One or More Threads**
    - » *One Page table per process!*
  - Owns memory (mapped pages)
  - Owns file descriptors, file system context, …
  - Encapsulates one or more threads sharing process resources
- Thread: *a sequential execution stream within process* (Sometimes called a "Lightweight process")
  - Process still contains a single Address Space
  - No protection between threads
- Multithreading: *a single program made up of a number of different concurrent activities*
  - Sometimes called multitasking, as in Ada …
- 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 ≡ Process with one thread

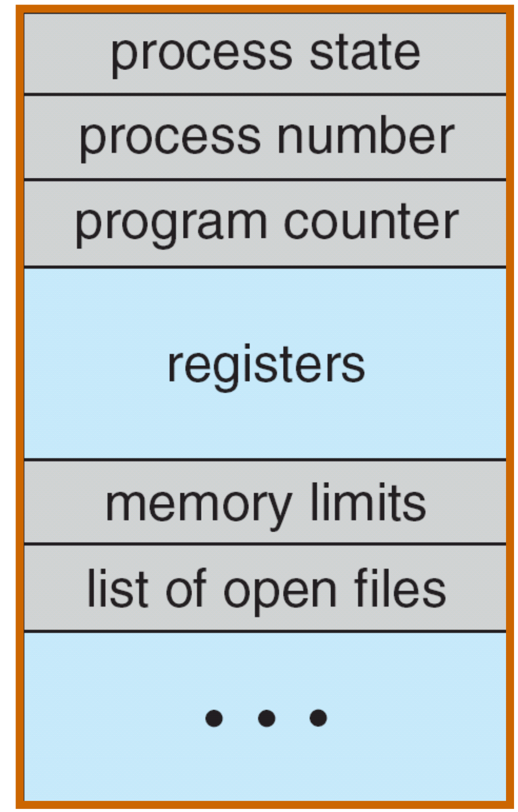
Recall: Single and Multithreaded Processes



- Threads encapsulate concurrency: "Active" component
- Address spaces encapsulate protection: "Passive" part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?

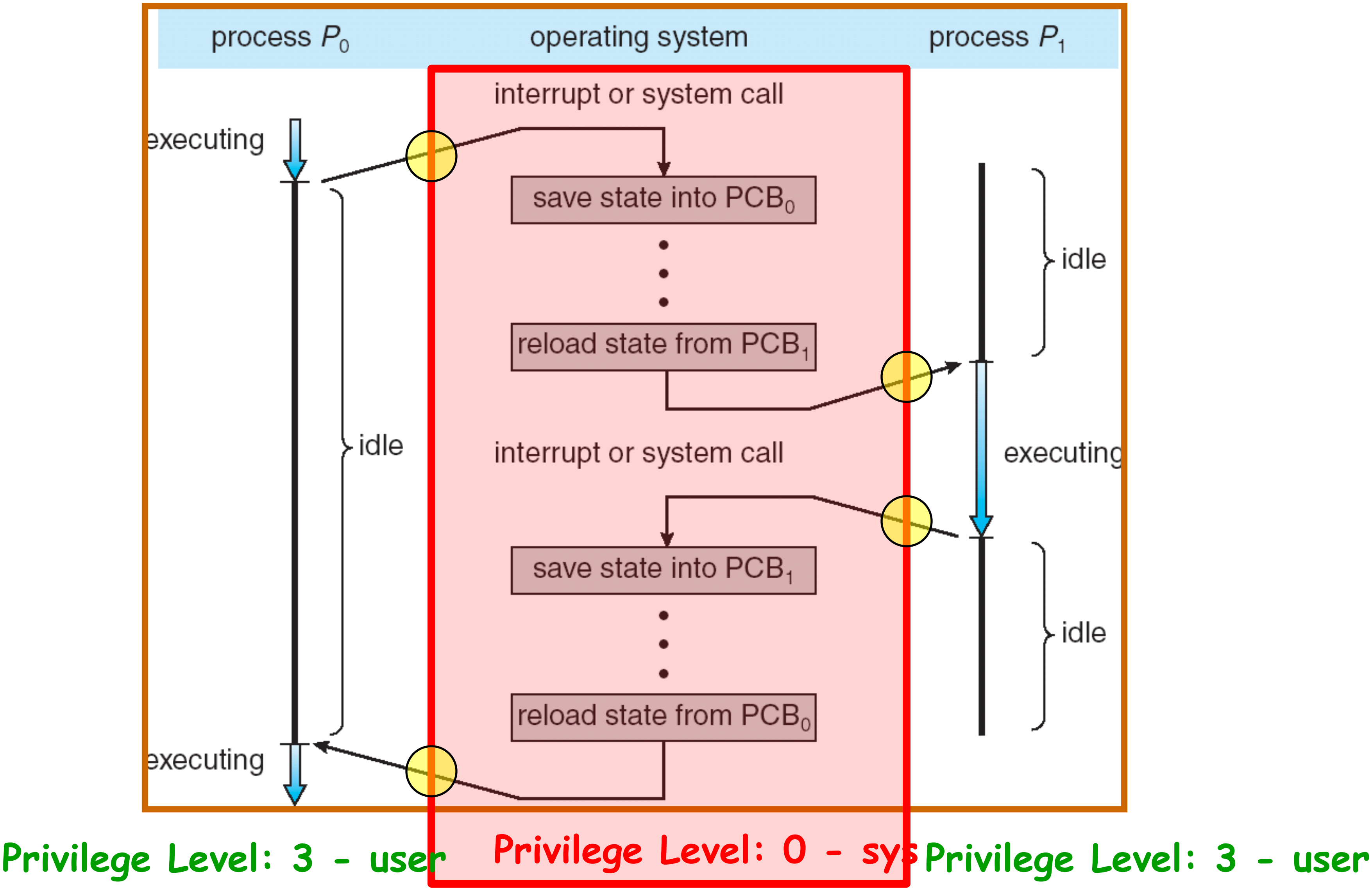
Recall: How do we Multiplex Processes?

- The current state of process held in a process control block (PCB):
  - This is a "snapshot" of the execution and protection environment
  - Only one PCB active at a time
- Give out CPU time to different processes (Scheduling):
  - Only one process "running" at a time
  - Give more time to important processes
- Give pieces of resources to different processes (Protection):
  - Controlled access to non-CPU resources
  - Example mechanisms:
    - » Memory Translation: Give each process their own (protected) address space
    - » Kernel/User duality: Arbitrary multiplexing of I/O through system calls

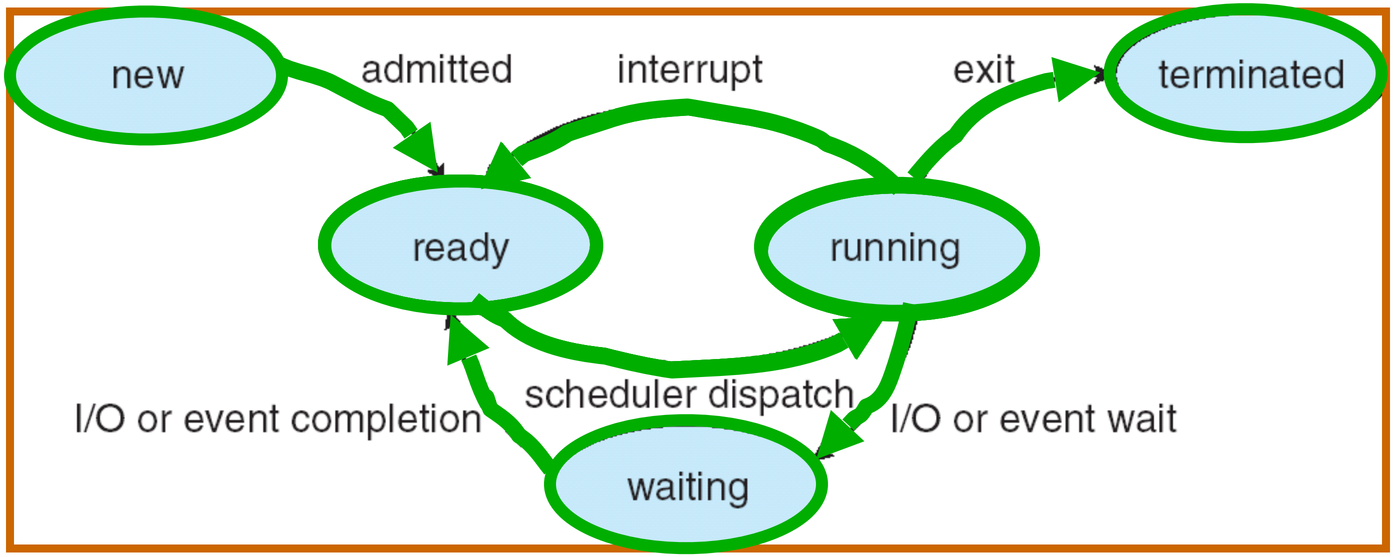


Process Control Block

Recall: Context Switch



Recall: Lifecycle of a Process

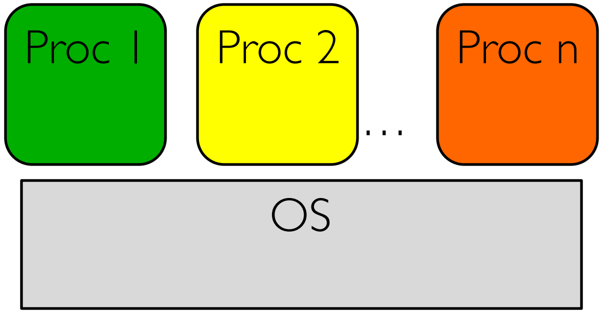


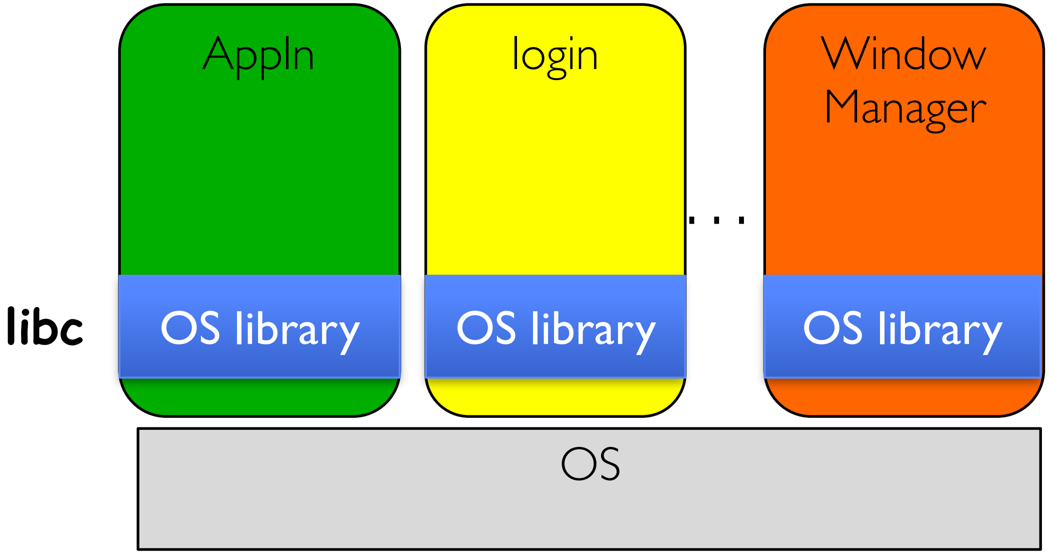
- As a process executes, it changes state:
  - new: The process is being created
  - ready: The process is waiting to run
  - running: Instructions are being executed
  - waiting: Process waiting for some event to occur
  - terminated: The process has finished execution

Discussion

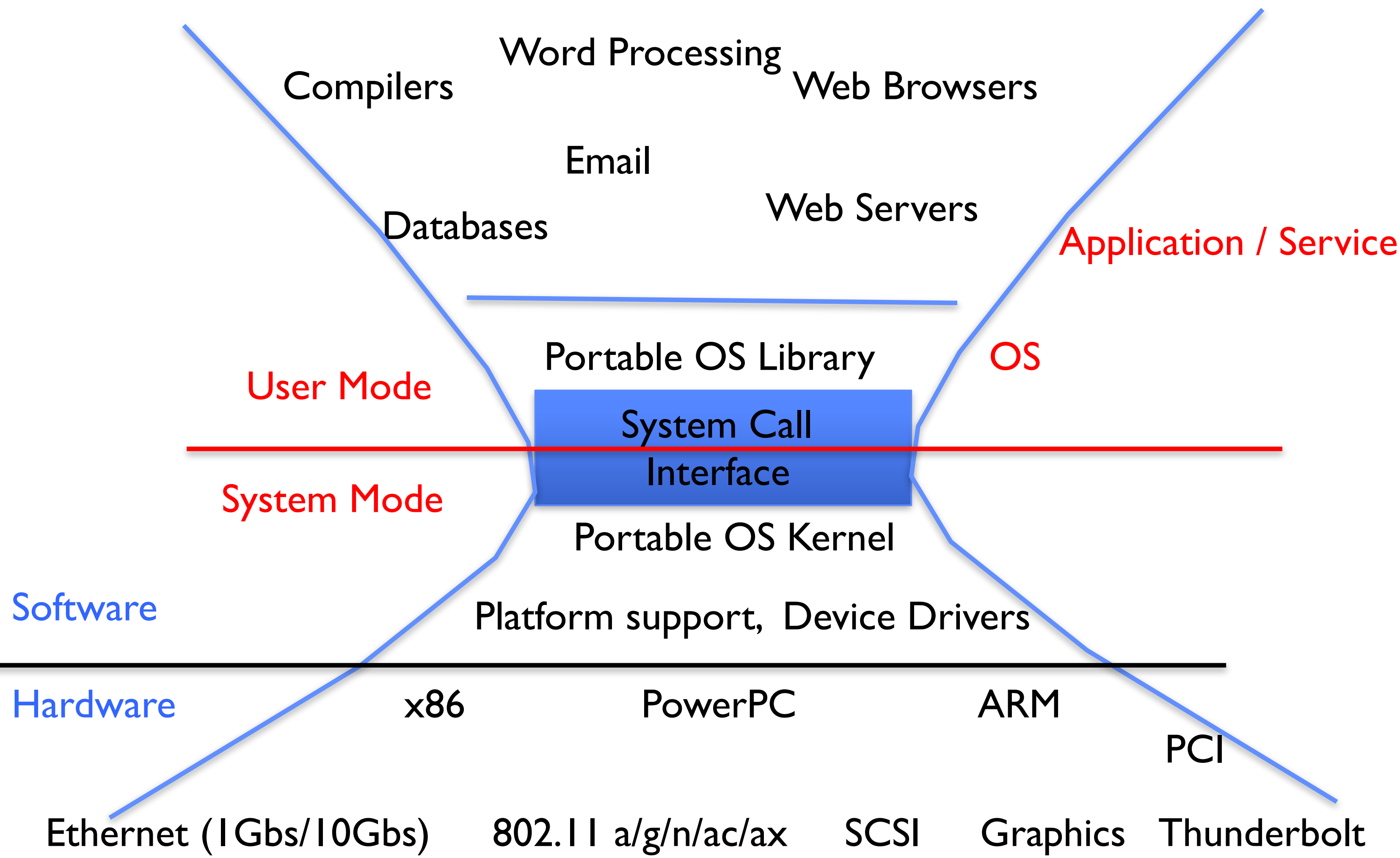
- Process is an *instance* of an *executing* program
  - The fundamental OS responsibility
  - Each instance has an identity (Process ID) or PID
- Processes do their work by processing and calling file system operations
  - This involves interacting with the Kernel!
  - How do we do that?
- Are their any operations on processes themselves?
  - create (fork) ?
  - terminate (exit) ?
  - sleep (sleep) ?
  - communicate with (e.g. signal)?

OS Run-Time Library





A Narrow Waist



pid.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, char *argv[])
{
  pid_t pid = getpid();   /* get current processes PID */

  printf("My pid: %d\n", pid);

  exit(0);
}
```

ps anyone?

POSIX/Unix

- Portable Operating System Interface [X?]
- Defines "Unix", derived from AT&T Unix
  - Created to bring order to many Unix-derived OSs
- Interface for application programmers (mostly)

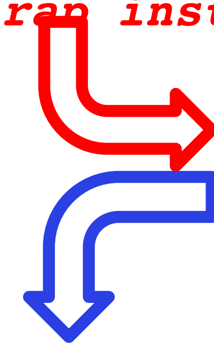
System Calls

Application:

```
fd = open(pathname);
```

Library:

```
File *open(pathname) {
    asm code … syscall # into ax
    put args into registers bx, …
    special trap instruction
```



Operating System:

```
get args from regs
dispatch to system func
process, schedule, …
complete, resume process
```

```
    get results from regs
};
```

```
Continue with results
```

Pintos: userprog/syscall.c, lib/user/syscall.c

SYSCALLs (of over 300)

%eax	Name	Source	%ebx	%ecx	%edx	%esi	%edi
1	sys_exit	kernel/exit.c	int	-	-	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t	-	-
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t	-	-
5	sys_open	fs/open.c	const char *	int	int	-	-
6	sys_close	fs/open.c	unsigned int	-	-	-	-
7	sys_waitpid	kernel/exit.c	pid_t	unsigned int *	int	-	-
8	sys_creat	fs/open.c	const char *	int	-	-	-
9	sys_link	fs/namei.c	const char *	const char *	-	-	-
10	sys_unlink	fs/namei.c	const char *	-	-	-	-
11	sys_execve	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
12	sys_chdir	fs/open.c	const char *	-	-	-	-
13	sys_time	kernel/time.c	int *	-	-	-	-
14	sys_mknod	fs/namei.c	const char *	int	dev_t	-	-
15	sys_chmod	fs/open.c	const char *	mode_t	-	-	-
16	sys_lchown	fs/open.c	const char *	uid_t	gid_t	-	-
18	sys_stat	fs/stat.c	char *	struct __old_kernel_stat *	-	-	-
19	sys_lseek	fs/read_write.c	unsigned int	off_t	unsigned int	-	-
20	sys_getpid	kernel/sched.c	-	-	-	-	-
21	sys_mount	fs/super.c	char *	char *	char *	-	-
22	sys_oldumount	fs/super.c	char *	-	-	-	-
23	sys_setuid	kernel/sys.c	uid_t	-	-	-	-
24	sys_getuid	kernel/sched.c	-	-	-	-	-
25	sys_stime	kernel/time.c	int *	-	-	-	-
26	sys_ptrace	arch/i386/kernel/ptrace.c	long	long	long	long	-
27	sys_alarm	kernel/sched.c	unsigned int	-	-	-	-
28	sys_fstat	fs/stat.c	unsigned int	struct __old_kernel_stat *	-	-	-
29	sys_pause	arch/i386/kernel/sys_i386.c	-	-	-	-	-
30	sys_utime	fs/open.c	char *	struct utimbuf *	-	-	-

Pintos: syscall-nr.h

Recall: Kernel System Call Handler

- Locate arguments
  - In registers or on user(!) stack
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back
  - into user memory

Process Management

- `exit – terminate a process`
- `fork` – copy the current process
- `exec` – change the *program* being run by the current process
- `wait` – wait for a process to finish
- `kill` – send a *signal* (interrupt-like notification) to another process
- `sigaction` – set handlers for signals

Process Management

- **`exit` – terminate a process**
- **`fork`** – copy the current process
- **`exec`** – change the *program* being run by the current process
- **`wait`** – wait for a process to finish
- **`kill`** – send a *signal* (interrupt-like notification) to another process
- **`sigaction`** – set handlers for signals

Creating Processes

- pid_t fork(); -- copy the current process
  - This means everything!
  - New process has different pid
- Return value from fork(): pid (like an integer)
  - When > 0:
    - » Running in (original) Parent process
    - » return value is pid of new child
  - When = 0:
    - » Running in new Child process
  - When < 0:
    - » Error! Must handle somehow
    - » Running in original process
- If no error: State of original process duplicated in *both* Parent and Child!
  - Address Space (Memory), File Descriptors (covered later), etc…
  - Not as bad as it seems – really only copy page table [more later]

fork1.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
  pid_t cpid, mypid;
  pid_t pid = getpid();            /* get current processes PID
*/
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {                    /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  } else if (cpid == 0) {            /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
  }
}
```

fork1.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
  pid_t cpid, mypid;
  pid_t pid = getpid();            /* get current processes PID
*/
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {                  /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  } else if (cpid == 0) {          /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
  }
}
```

fork1.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
  pid_t cpid, mypid;
  pid_t pid = getpid();            /* get current processes PID
*/
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {                  /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  } else if (cpid == 0) {          /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
  }
}
```

fork_race.c

```c
int i;
cpid = fork();
if (cpid > 0) {
  for (i = 0; i < 10; i++) {
    printf("Parent: %d\n", i);
    // sleep(1);
  }
} else if (cpid == 0) {
  for (i = 0; i > -10; i--) {
    printf("Child: %d\n", i);
    // sleep(1);
  }
}
```

- What does this print?
- Would adding the calls to `sleep` matter?

Fork "race"

```
int i;
cpid = fork();
if (cpid > 0) {
  for (i = 0; i < 10; i++) {
    printf("Parent: %d\n", i);
    // sleep(1);
  }
} else if (cpid == 0) {
  for (i = 0; i > -10; i--) {
    printf("Child: %d\n", i);
    // sleep(1);
  }
}
```

Process Management

- **`fork`** – copy the current process
- **`exec`** – change the *program* being run by the current process
- **`wait`** – wait for a process to finish
- **`kill`** – send a *signal* (interrupt-like notification) to another process
- **`sigaction`** – set handlers for signals

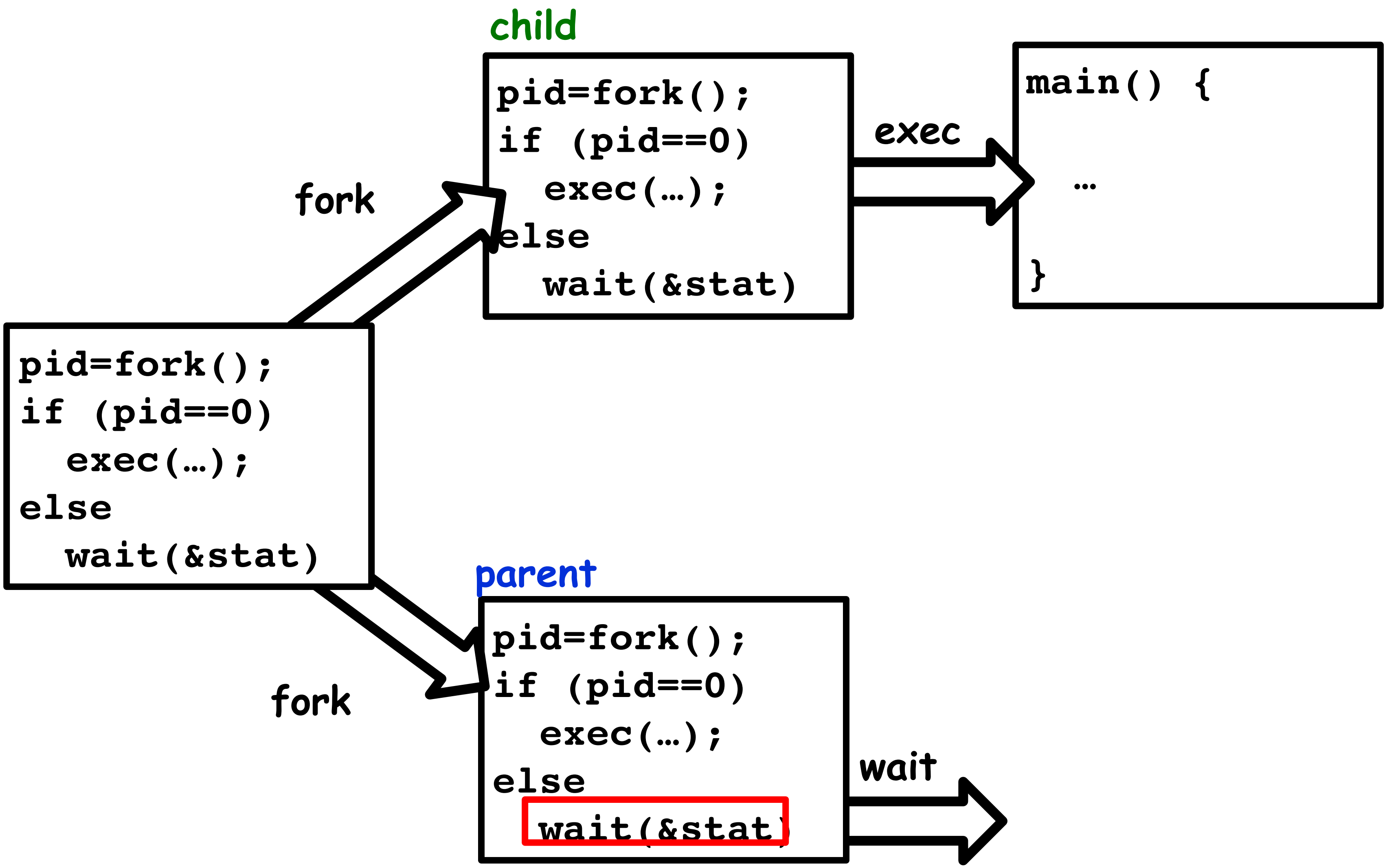
fork2.c – parent waits for child to finish

```
int status;
pid_t tcpid;
…
cpid = fork();
if (cpid > 0) {                /* Parent Process */
  mypid = getpid();
  printf("[%d] parent of [%d]\n", mypid, cpid);
  tcpid = wait(&status);
  printf("[%d] bye %d(%d)\n", mypid, tcpid,
status);
} else if (cpid == 0) {        /* Child Process */
  mypid = getpid();
  printf("[%d] child\n", mypid);
  exit(42);
}
…
```

Process Management

- **`fork`** – copy the current process
- **`exec`** – change the *program* being run by the current process
- **`wait`** – wait for a process to finish
- **`kill`** – send a *signal* (interrupt-like notification) to another process
- **`sigaction`** – set handlers for signals

Process Management



fork3.c

```
…
cpid = fork();
if (cpid > 0) {                         /* Parent Process */
  tcpid = wait(&status);
} else if (cpid == 0) {             /* Child Process */
  char *args[] = {“ls”, “-l”, NULL};
  execv(“/bin/ls”, args);
  /* execv doesn’t return when it works.
     So, if we got here, it failed! */
  perror(“execv”);
  exit(1);
}
…
```

Shell

- A shell is a job control system
  – Allows programmer to create and manage a set of programs to do some task
  – Windows, MacOS, Linux all have shells

- Example: to compile a C program

  cc –c sourcefile1.c

  cc –c sourcefile2.c

  ln –o program sourcefile1.o sourcefile2.o

  ./program

Process Management

- **`fork`** – copy the current process
- **`exec`** – change the *program* being run by the current process
- **`wait`** – wait for a process to finish
- **`kill`** – send a *signal* (interrupt-like notification) to another process
- **`sigaction`** – set handlers for signals

inf_loop.c

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

void signal_callback_handler(int signum) {
  printf("Caught signal!\n");
  exit(1);
}
int main() {
  struct sigaction sa;
  sa.sa_flags = 0;
  sigemptyset(&sa.sa_mask);
  sa.sa_handler = signal_callback_handler;

  sigaction(SIGINT, &sa, NULL);
  while (1) {}
}
```

Common POSIX Signals

- `SIGINT` – control-C
- `SIGTERM` – default for `kill` shell command
- `SIGSTP` – control-Z (default action: stop process)

- `SIGKILL`, `SIGSTOP` – terminate/stop process
  – Can't be changed or disabled with `sigaction`
  – Why?

Administrivia

- HW0 due today!
- HW1 started?
- Groups assignment
- Any issues?

Reminder: Definitions

- A *thread* is a single execution sequence that represents a separately schedulable task
- Protection is an orthogonal concept
  - Can have one or many threads per protection domain
  - Single threaded user program: one thread, one protection domain
  - Multi-threaded user program: multiple threads, sharing same data structures, isolated from other user programs
  - Multi-threaded kernel: multiple threads, sharing kernel data structures, capable of using privileged instructions

Threads Motivation

- Operating systems need to be able to handle *multiple things at once* (MTAO)
  – processes, interrupts, background system maintenance
- Servers need to handle MTAO
  – Multiple connections handled simultaneously
- Parallel programs need to handle MTAO
  – To achieve better performance
- Programs with user interfaces often need to handle MTAO
  – To achieve user responsiveness while doing computation
- Network and disk bound programs need to handle MTAO
  – To hide network/disk latency
  – Sequence steps in access or communication

Silly Example for Threads

Imagine the following program:

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

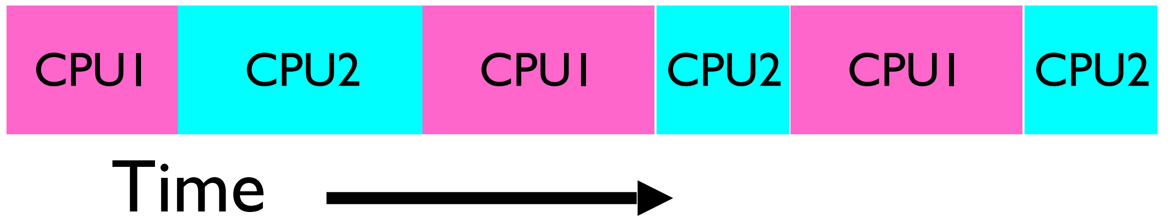
- What is the behavior here?
  - Program would never print out class list
  - Why? `ComputePI` would never finish

Adding Threads

- Version of program with Threads (loose syntax):

```
main() {
    thread_fork(ComputePI, "pi.txt" ));
    thread_fork(PrintClassList, "classlist.txt"));
}
```

- `thread_fork`: 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



More Practical Motivation

Back to Jeff Dean's "Numbers everyone should know":

Handle I/O in separate thread, avoid blocking other progress

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Little Better Example for Threads?

Imagine the following program:

```
main() {
    …
    ReadLargeFile("pi.txt");
    RenderUserInterface();
}
```

- What is the behavior here?
  - Still respond to user input
  - While reading file in the background

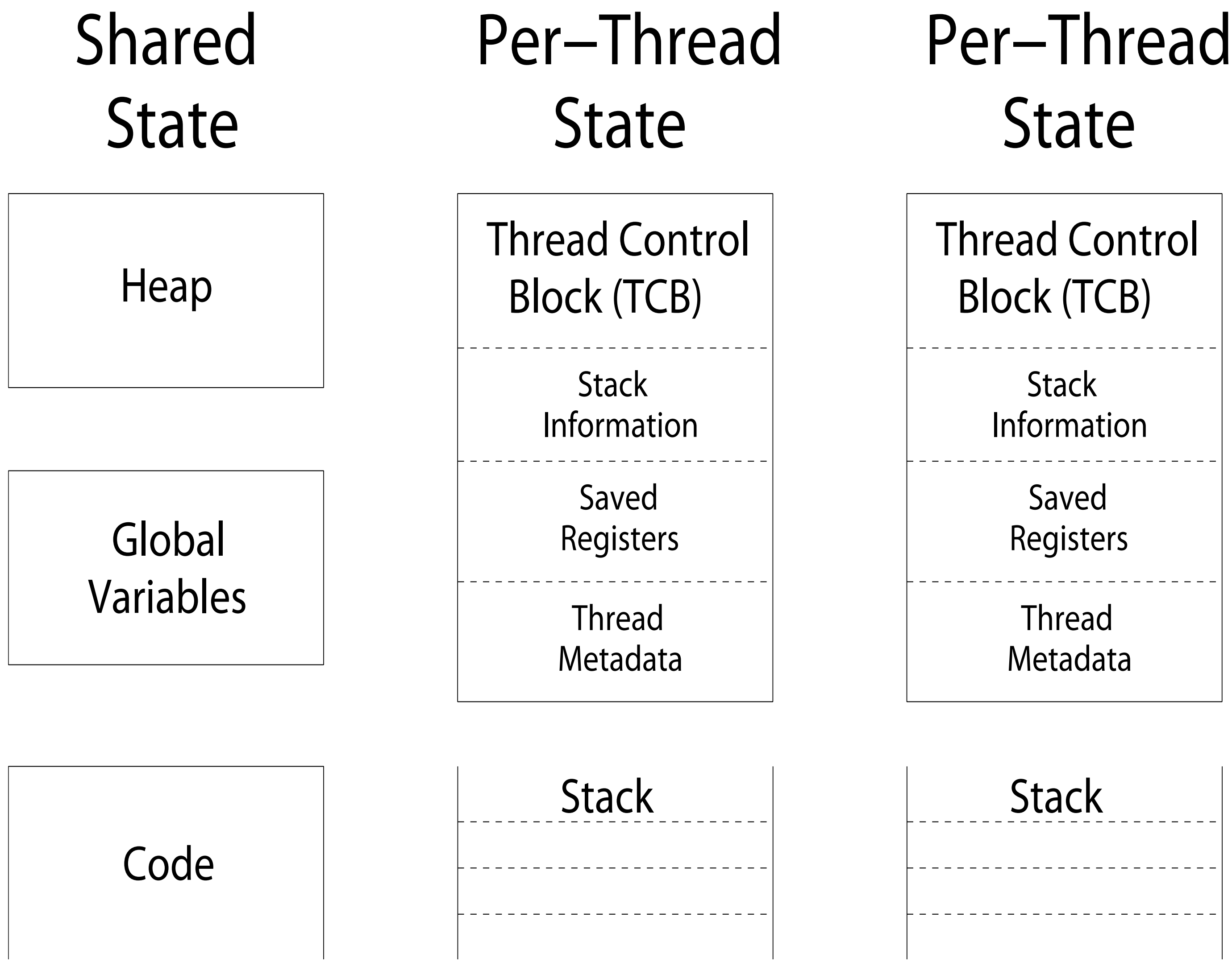
Voluntarily Giving Up Control

- I/O – e.g. keypress
- Waiting for a signal from another thread
  – Thread makes system call to *wait*
- Thread executes **`thread_yield()`**
  – Relinquishes CPU but puts calling thread back on ready queue

Thread State

- State shared by all threads in process/address space
  – Content of memory (global variables, heap)
  – I/O state (file descriptors, 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

Shared vs. Per-Thread State



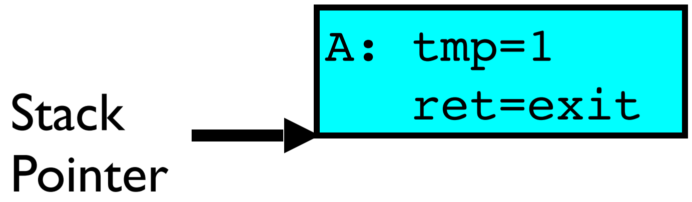
Execution Stack Example

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

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

Execution Stack Example

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



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

Execution Stack Example

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

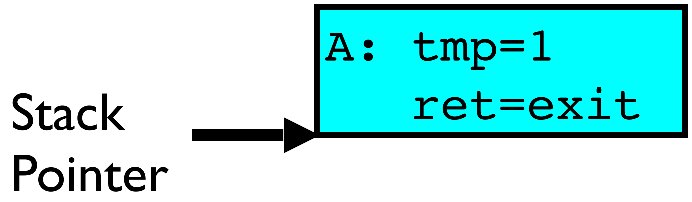
Stack Pointer →

```
A: tmp=1
   ret=exit
```

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

Execution Stack Example

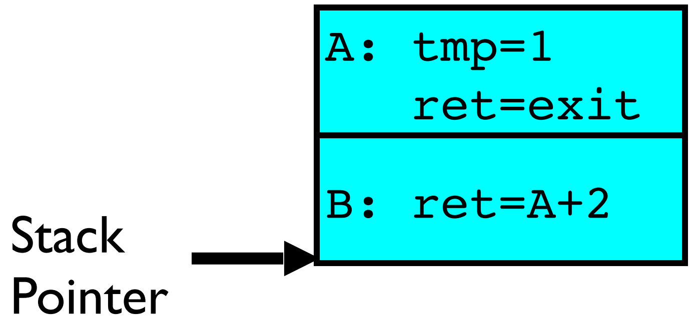
```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



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

Execution Stack Example

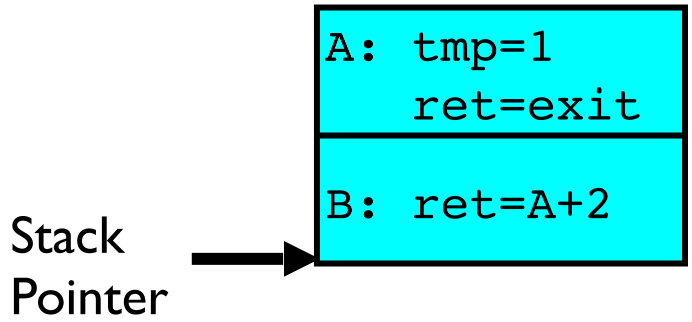
```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



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

Execution Stack Example

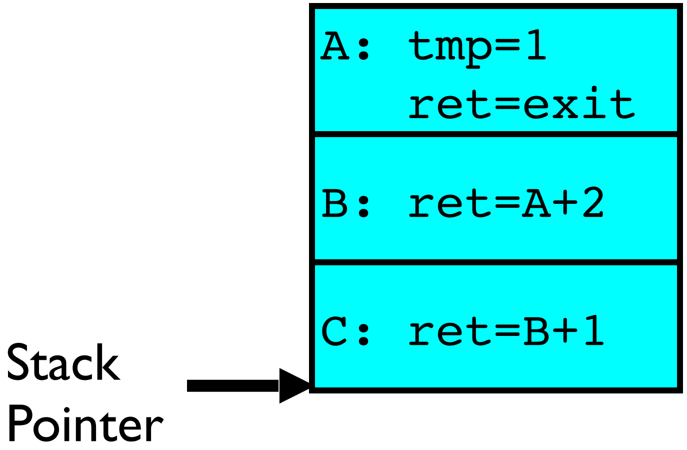
```
      A(int tmp) {
A:      if (tmp<2)
A+1:      B();
A+2:    printf(tmp);
      }
      B() {
B:      C();
B+1:  }
      C() {
C:      A(2);
C+1:  }
      A(1);
exit:
```



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

Execution Stack Example

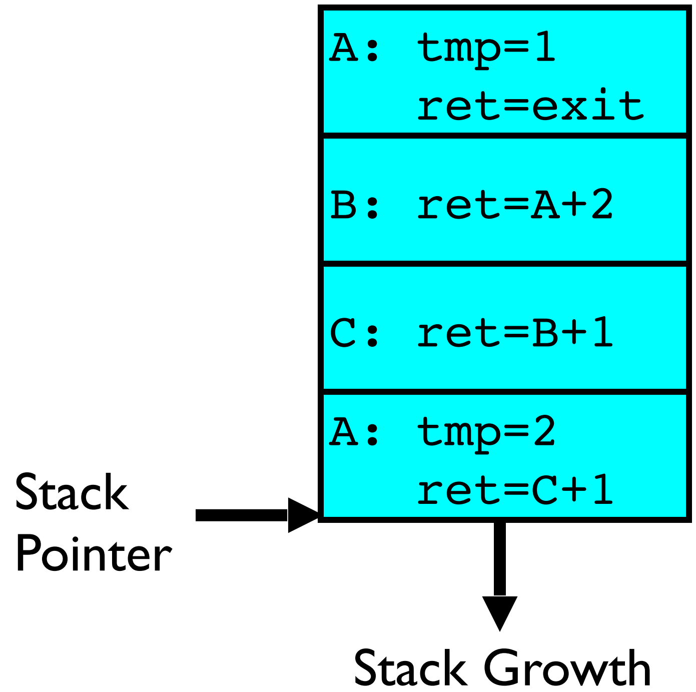
```
     A(int tmp) {
A:     if (tmp<2)
A+1:     B();
A+2:   printf(tmp);
     }
     B() {
B:     C();
B+1: }
     C() {
C:     A(2);
C+1: }
     A(1);
exit:
```



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

Execution Stack Example

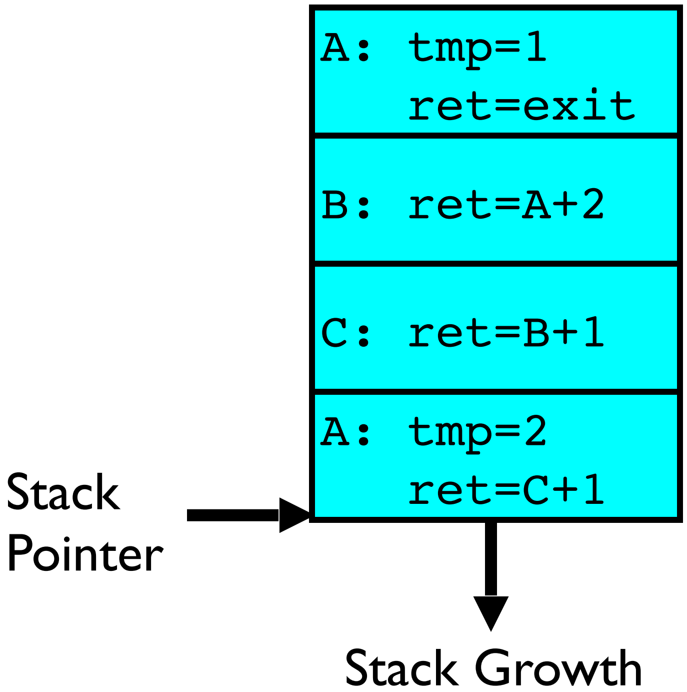
```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



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

Execution Stack Example

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

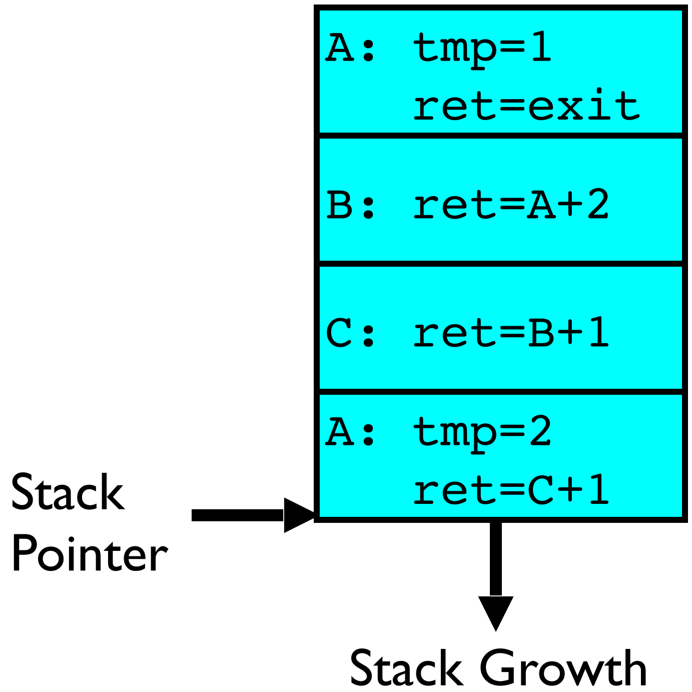


Output >2

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

Execution Stack Example

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

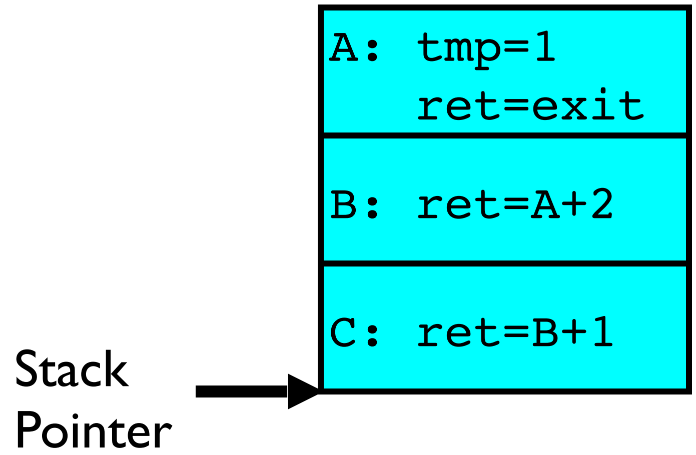


Output: >2

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

Execution Stack Example

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

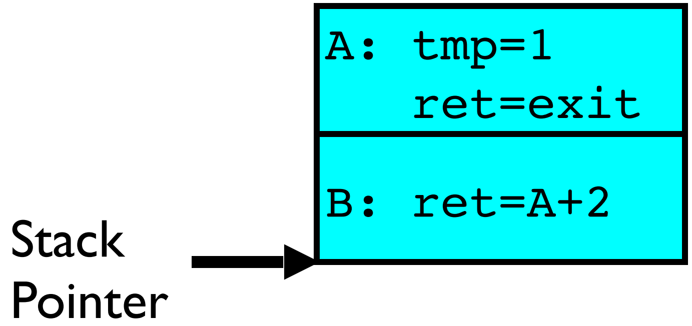


Output: >2

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

Execution Stack Example

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



Output: >2

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

Execution Stack Example

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

Stack Pointer →

```
A: tmp=1
   ret=exit
```

Output: >2 1

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

Execution Stack Example

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

Stack Pointer →

```
A:  tmp=1
    ret=exit
```

Output: `>2 1`

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

Execution Stack Example

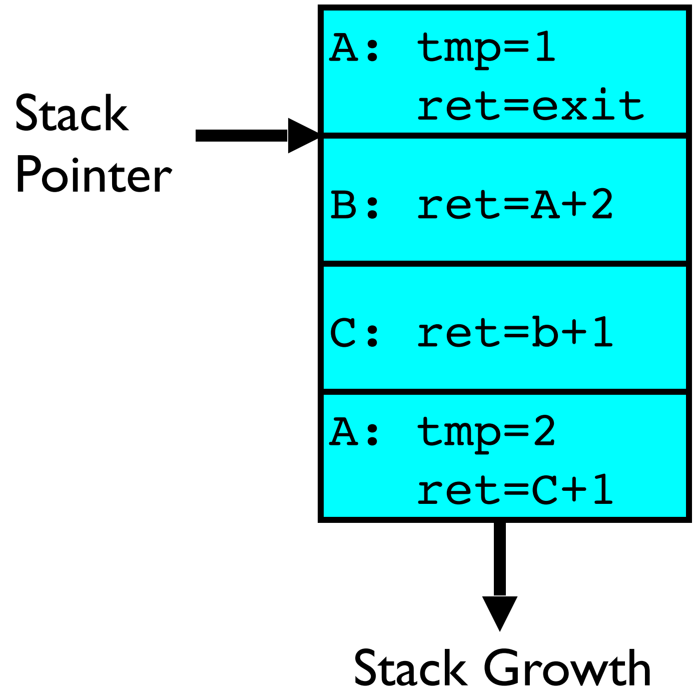
```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```

Output: `>2 1`

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

Execution Stack Example

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



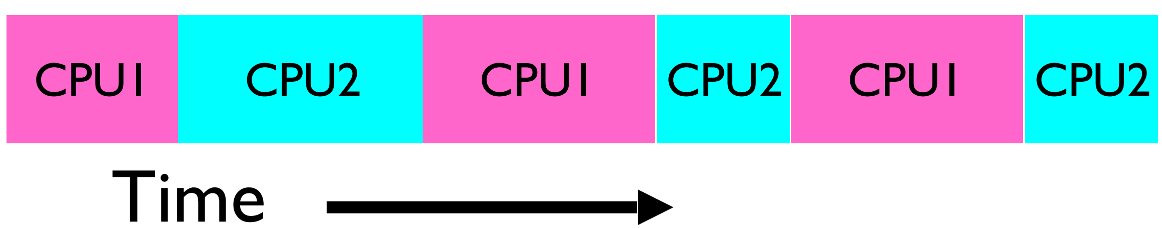
- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Adding Threads

- Version of program with Threads (loose syntax):

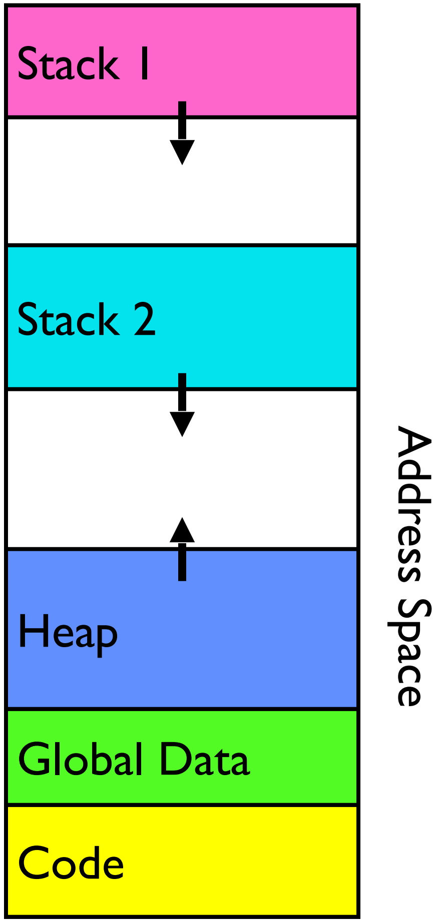
```
main() {
    thread_fork(ReadLargeFile, "pi.txt" );
    thread_fork(RenderUserInterface, "classlist.txt");
}
```

- `thread_fork`: 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: Two-Threads

- 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?



Actual Thread Operations

- **`thread_fork(func, args)`**
  - Create a new thread to run func(args)
  - Pintos: **`thread_create`**
- **`thread_yield()`**
  - Relinquish processor voluntarily
  - Pintos: **`thread_yield`**
- **`thread_join(thread)`**
  - In parent, wait for forked thread to exit, then return
  - Pintos: **`thread_join`**
- **`thread_exit`**
  - Quit thread and clean up, wake up joiner if any
  - Pintos: **`thread_exit`**

- **`pThreads`**: POSIX standard for thread programming [POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)]

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?

Running a thread

Consider first portion: **`RunThread()`**

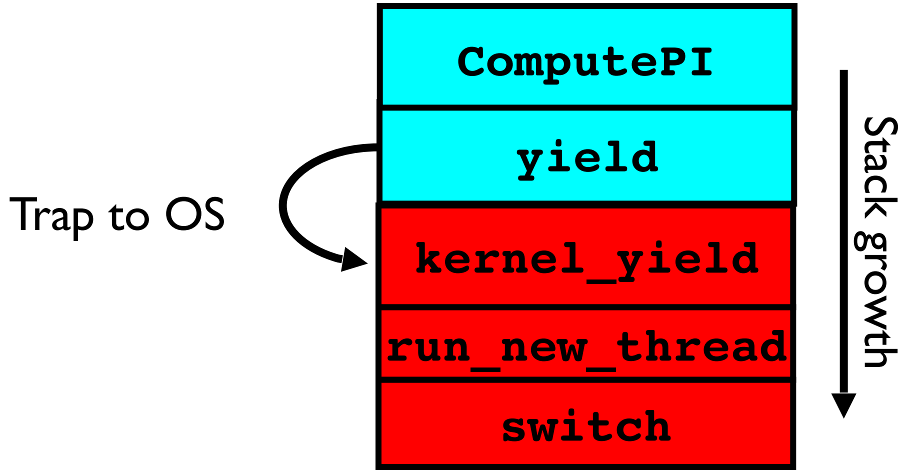
- How do I run a thread?
  - Load its state (registers, PC, stack pointer) into CPU
  - Load environment (virtual memory space, etc)
  - Jump to the PC
- How does the dispatcher get control back?
  - Internal events: thread returns control voluntarily
  - External events: thread gets *preempted*

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();
  }
}
```

Stack for Yielding Thread



- How do we run a new thread?

```
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* Do any cleanup */
}
```

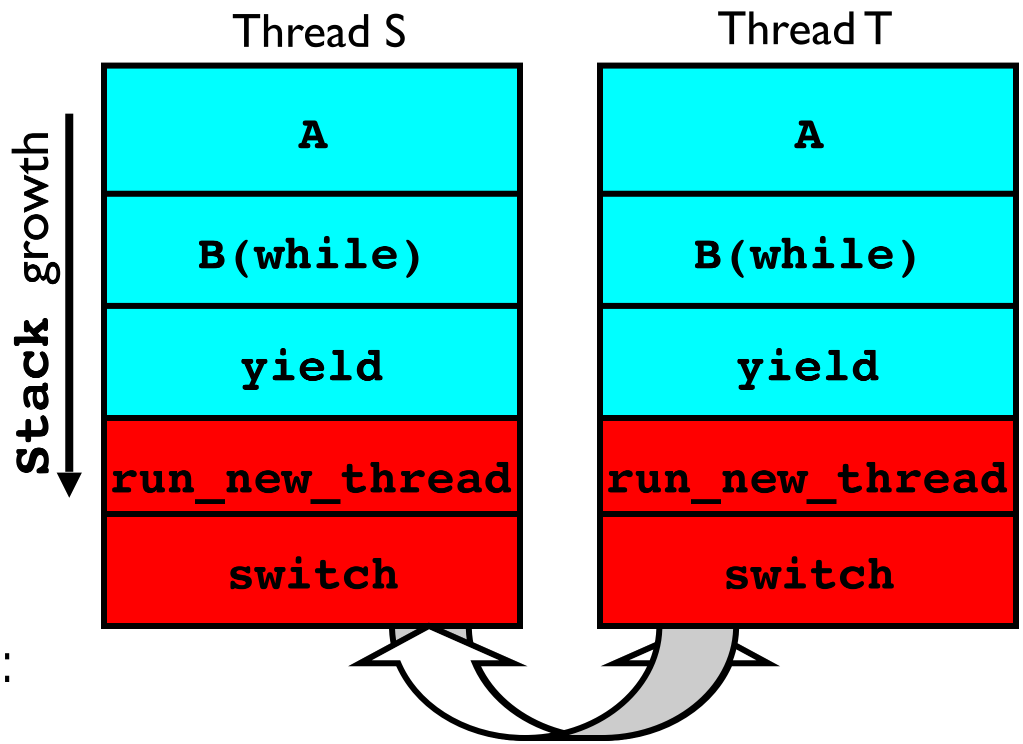
- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, stack pointer
  - Maintain isolation for each thread

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



Thread S's switch returns to Thread T's (and vice versa)

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 */
}
```

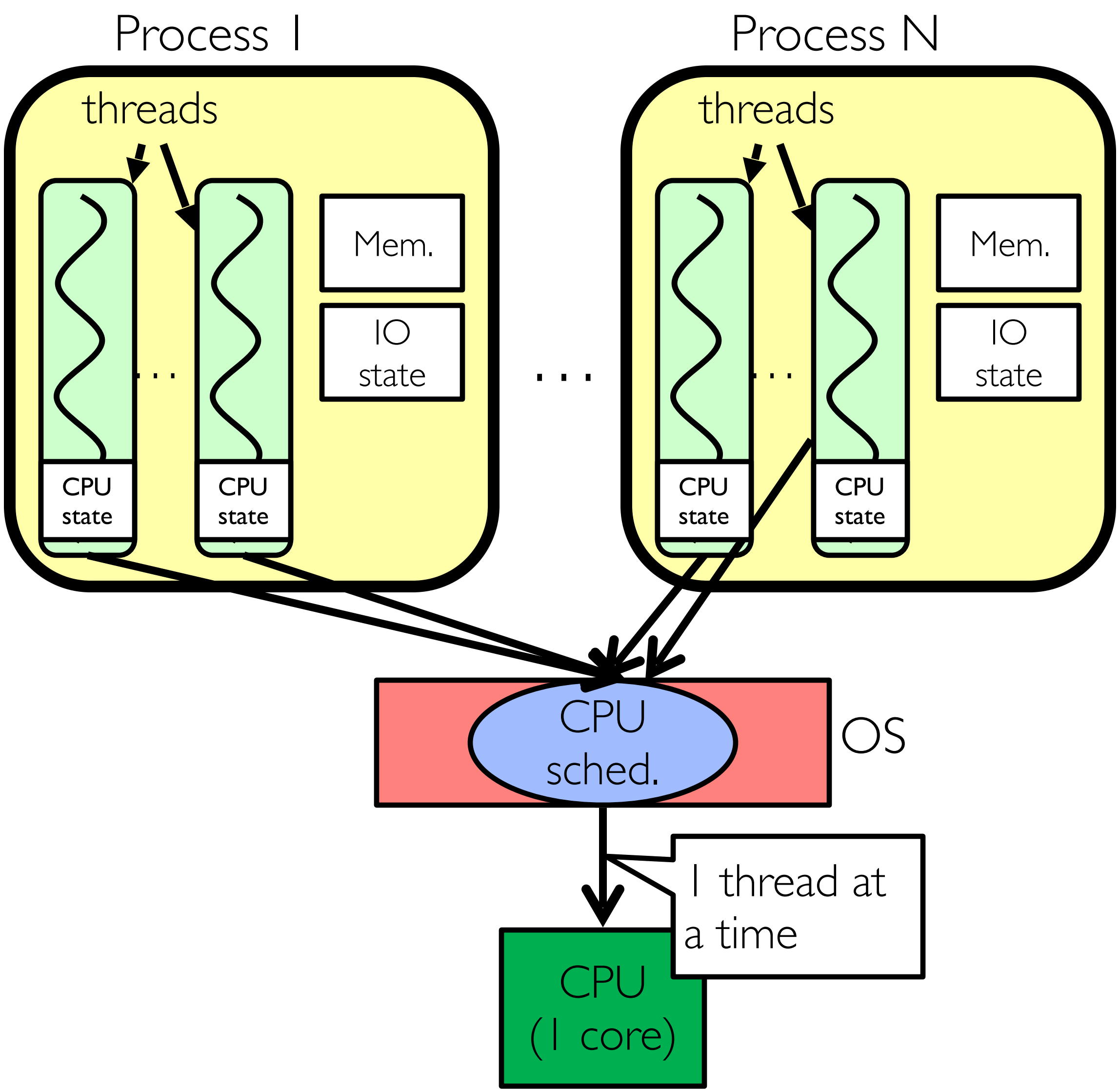
Switch Details (continued)

- What if you make a mistake in implementing switch?
  - Suppose you forget to save/restore register 32
  - Get intermittent failures depending on when context switch occurred and whether new thread uses register 32
  - 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 tale:
  - 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 (no one removes features!)
    - » Very weird behavior started happening
  - Moral of story: Design for simplicity

Aren't we still switching contexts?

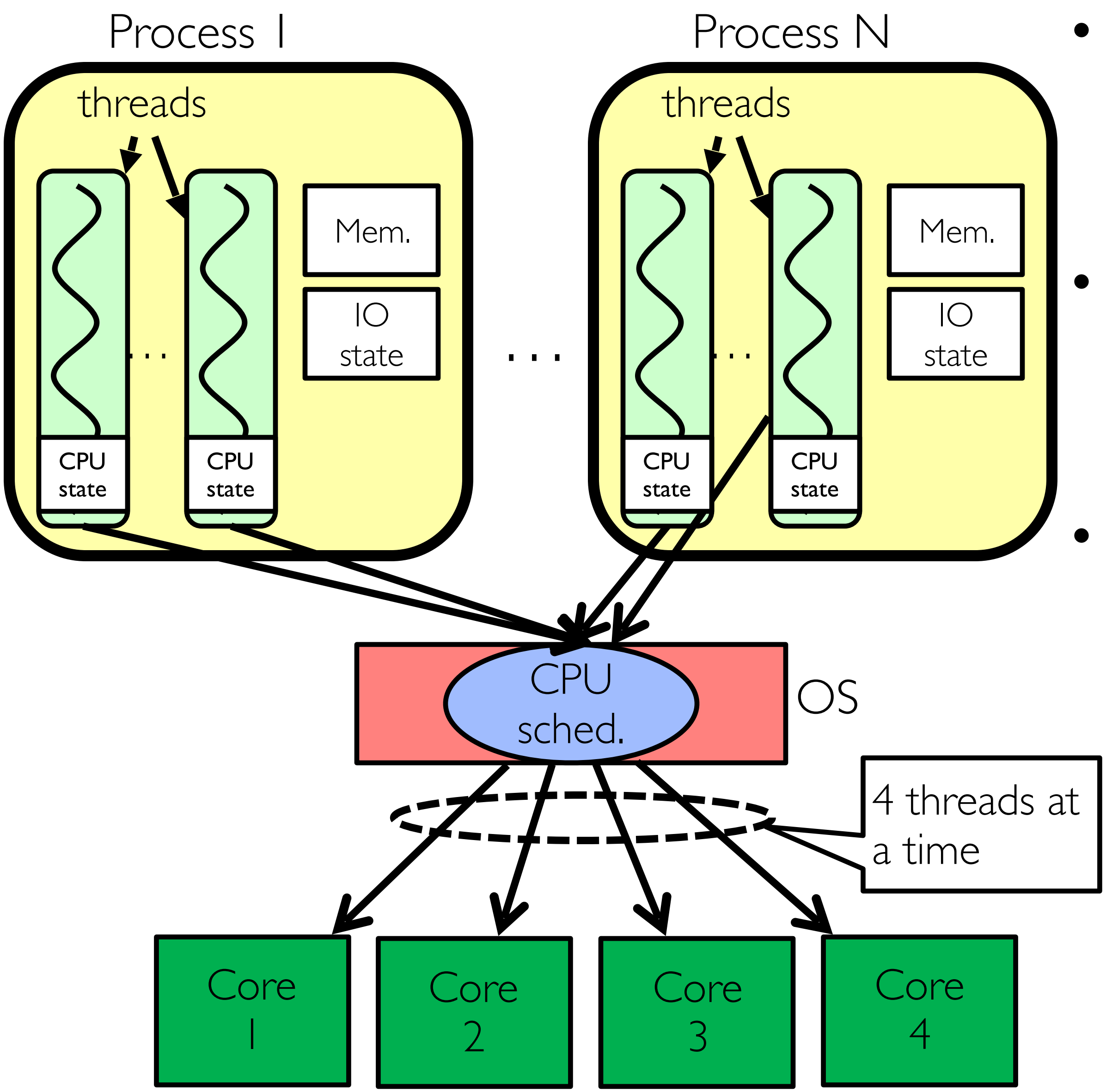
- Yes, but much cheaper than switching processes
  - No need to change address space
- Some numbers from Linux:
  - Frequency of context switch: 10-100ms
  - Switching between processes: 3-4 μsec.
  - Switching between threads: 100 ns

Processes vs. Threads



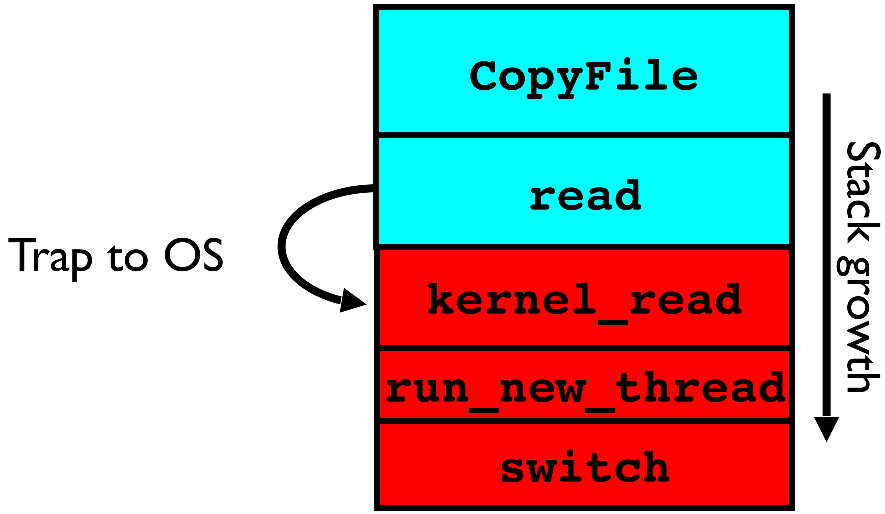
- Switch overhead:
  - Same process: low
  - Different proc.: high
- Protection
  - Same proc: low
  - Different proc: high
- Sharing overhead
  - Same proc: low
  - Different proc: high

Processes vs. Threads



- Switch overhead:
  - Same process: low
  - Different proc.: high
- Protection
  - Same proc: low
  - Different proc: high
- Sharing overhead
  - Same proc: low
  - Different proc: high

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

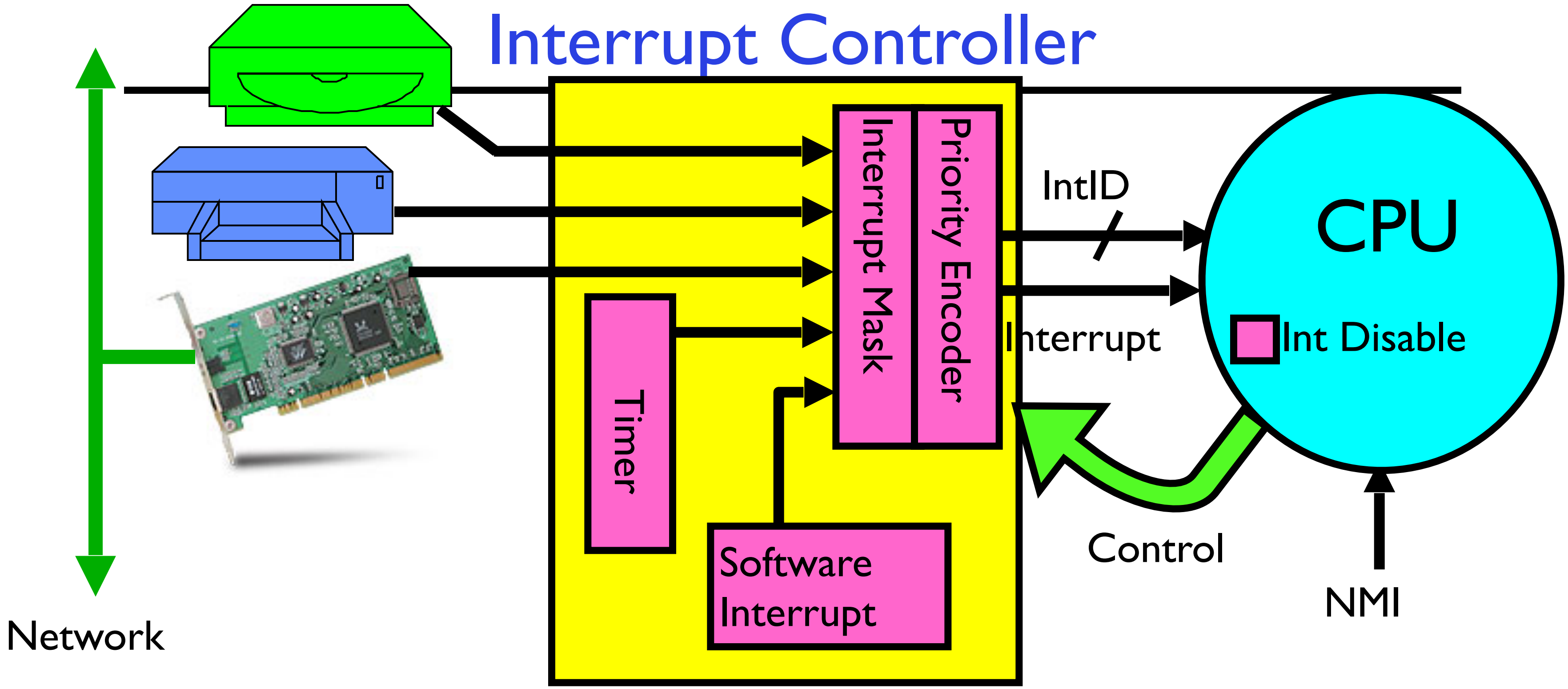
External Events

- 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 milliseconds

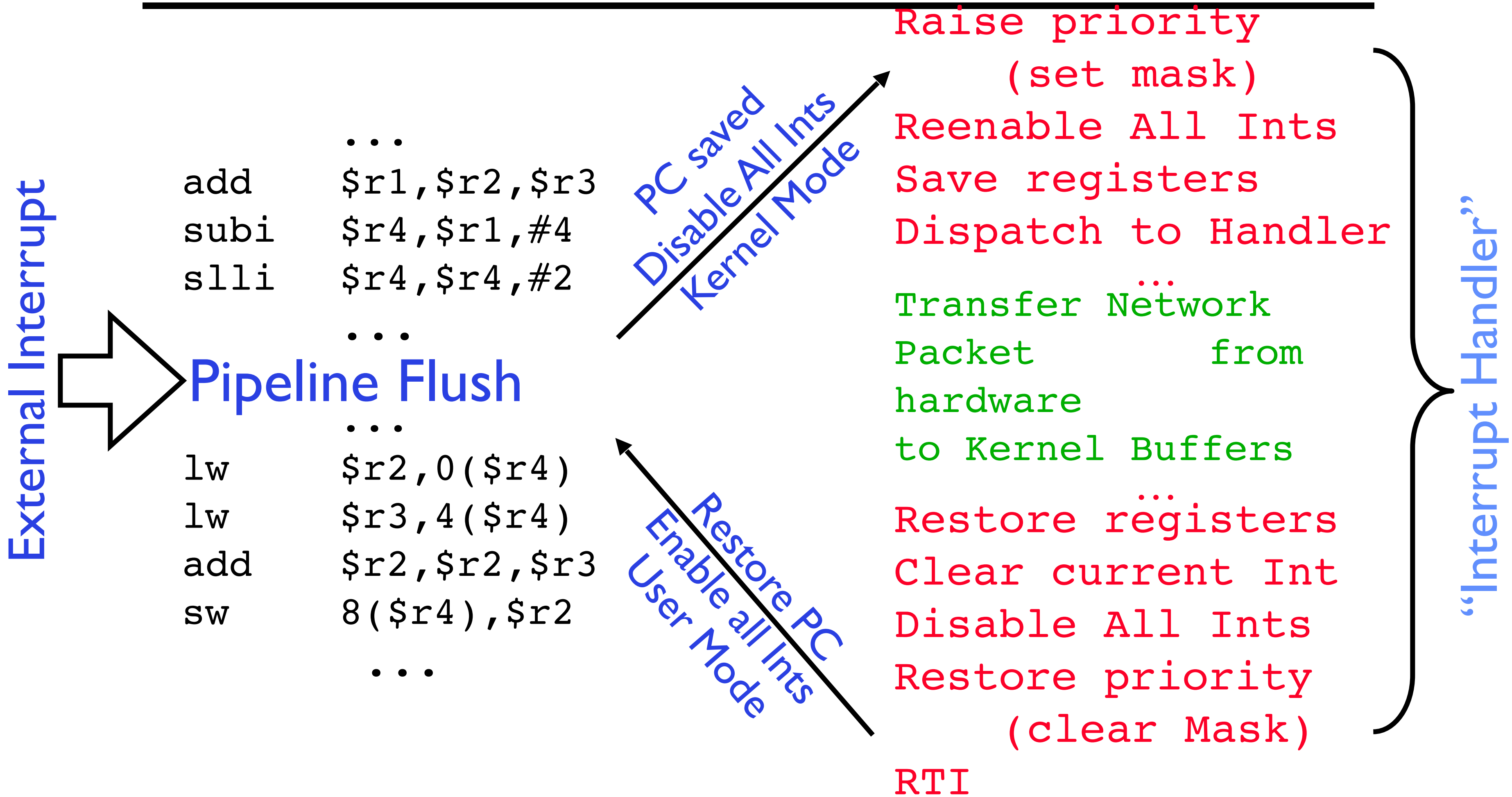
- If we make sure that external events occur frequently enough, can ensure dispatcher runs

Interrupt Controller



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

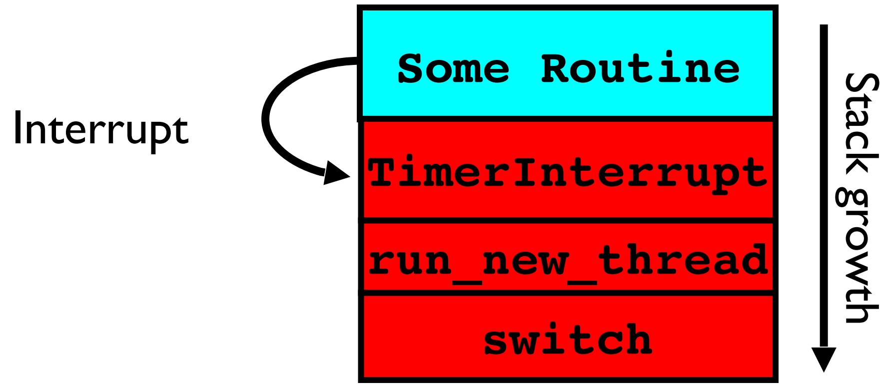
Example: Network Interrupt



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

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();
}
```

Hardware context switch support in x86

- Syscall/Intr (U → K)
  - PL 3 → 0;
  - TSS ← EFLAGS, CS:EIP;
  - SS:SP ← k-thread stack (TSS PL 0);
  - push (old) SS:ESP onto (new) k-stack
  - push (old) eflags, cs:eip, <err>
  - CS:EIP ← <k target handler>
- Then
  - *Handler then saves other regs, etc*
  - *Does all its works, possibly choosing other threads, changing PTBR (CR3)*

  - kernel thread has set up user GPRs
- iret (K → U)
  - PL 0 → 3;
  - Eflags, CS:EIP ← popped off k-stack
  - SS:SP ← user thread stack (TSS PL 3);

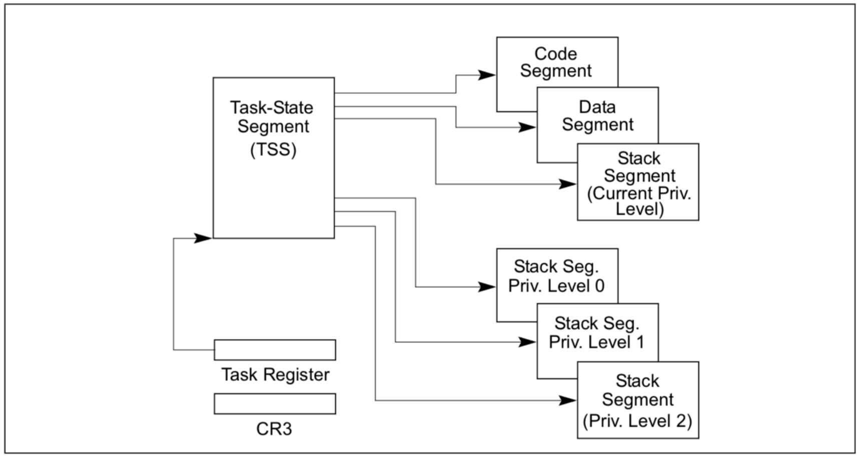


Figure 7-1. Structure of a Task

pg 2,942 of 4,922 of x86 reference manual

Pintos: tss.c, intr-stubs.S

Summary

- Processes have two parts
  - One or more Threads (Concurrency)
  - Address Spaces (Protection)
- **Threads: unit of concurrent execution**
  - Useful for parallelism, overlapping computation and IO, organizing sequences of interactions (protocols)
  - **Require: multiple stacks per address space**
  - **Thread switch:**
    - » Save/Restore registers, "return" from new thread's `switch` routine
- Concurrency accomplished by multiplexing CPU Time:
  - Unloading current thread (PC, registers)
  - Loading new thread (PC, registers)
  - Such context switching may be voluntary (**`yield()`**, I/O operations) or involuntary (timer, other interrupts)
- Concurrent threads introduce problems when accessing shared data
  - Programs must be insensitive to arbitrary interleavings
  - Without careful design, shared variables can become completely inconsistent