Lecturer: Dr. Michael J. May
Kinneret College

# Processes, Threads, Concurrency

Photo by Bozhin Karaivanov on Unsplash

12 December 2024
Lecture 6

Slides adapted from John Kubiatowicz (UC Berkeley)

# Concept Review

fork()

wait()

shell

Signal handler

kill()

FILE*

Stream

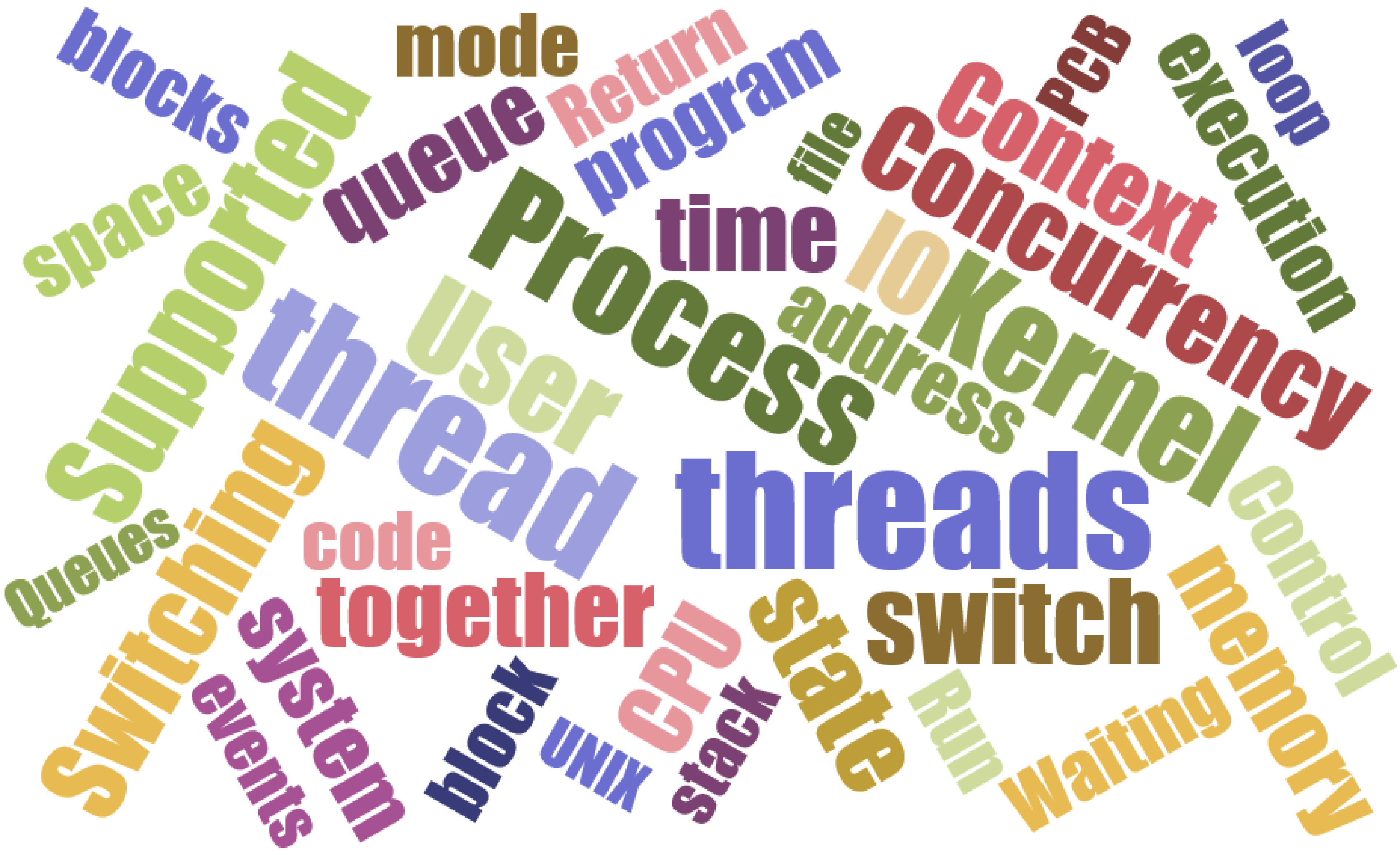
File Descriptor

File permissions
- Read
- Write
- Execute

Signals
- SIGINT
- SIGQUIT
- SIGKILL

# Topics for Today

- Concurrency: Processes
- Threads
  - Kernel Supported Threads vs. User Supported Threads
  - Switching between threads
- Concurrency: Putting it together
- Cooperating threads
- Concurrency challenge


blocks
mode
Return
program
PCB
loop
queue
supported
space
file
Context
execution
Concurrency
Process
time
IO
Kernel
User
thread
address
Supported
threads
control
code
Queues
Switching
together
switch
memory
system
state
CPU
events
block
UNIX
stack
Run
Waiting

# Recall: Traditional UNIX Process

- Process: Operating system abstraction to represent what is needed to run a single program
  - Often called a "HeavyWeight Process"
  - No concurrency in a "HeavyWeight Process"

Two parts:

1. Sequential program execution stream
   - Code executed as a sequential stream of execution (i.e., thread)
   - Includes state of CPU registers
2. Protected resources:
   - Main memory state (contents of address space)
   - I/O state (i.e., file descriptors)

# How do we Multiplex Processes?

## Current state of process held in a Process Control Block (PCB):

- 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):

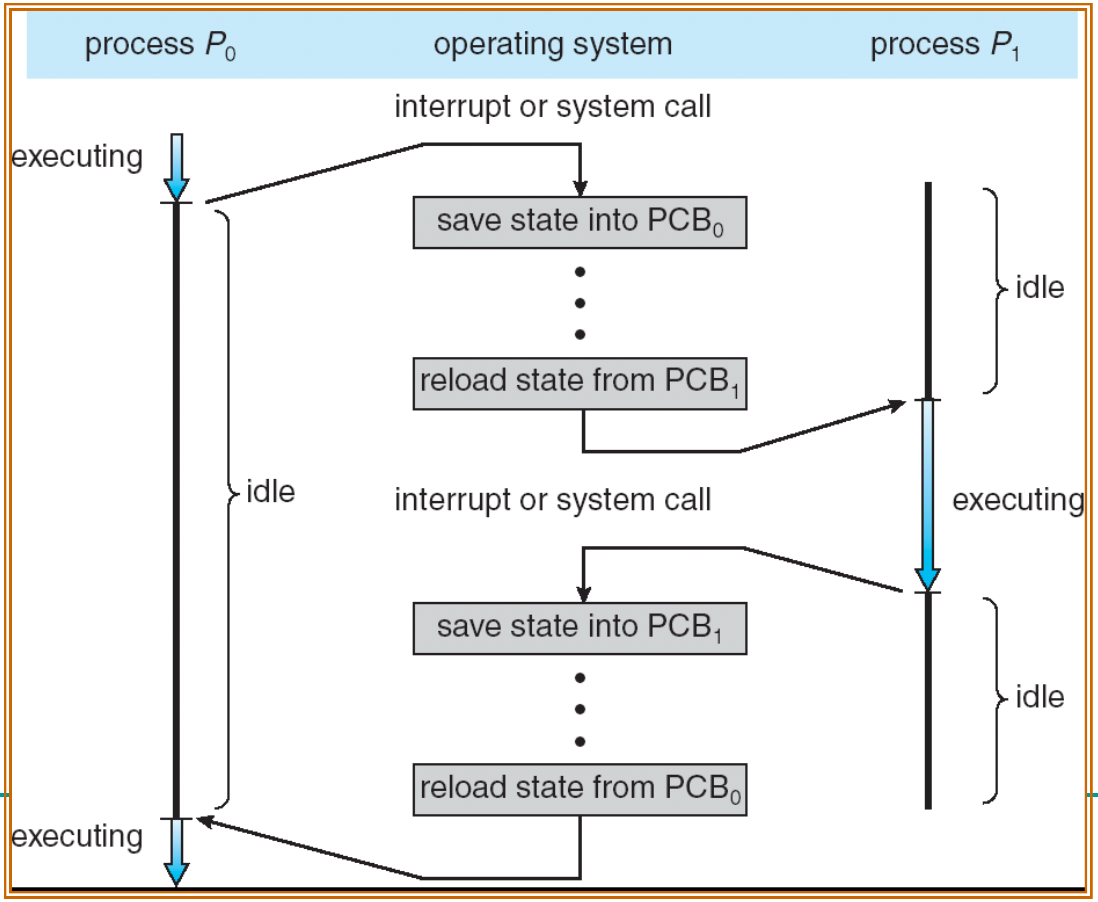
- Controlled access to non-CPU resources
- Example mechanisms:
  - Memory Mapping: Give each process their own address space
  - Kernel/User duality: Arbitrary multiplexing of I/O through system calls

| |
|---|
| Process state |
| Process number |
| Process counter |
| Registers |
| Memory limits |
| List of open files |
| … |

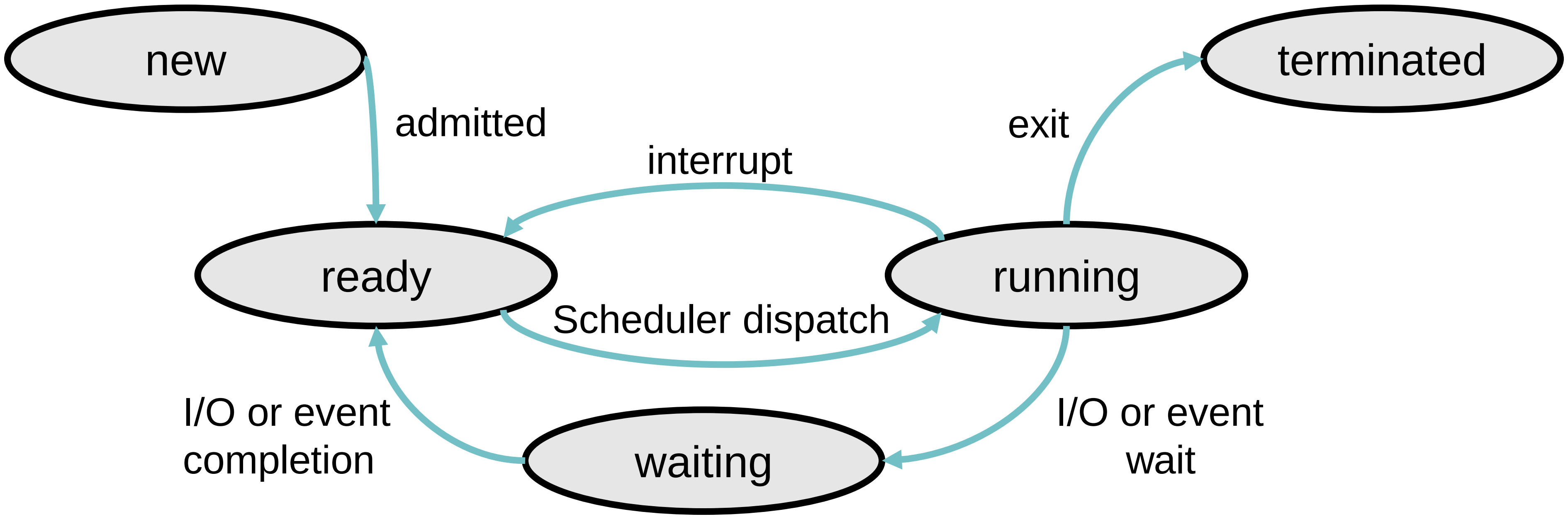
Process Control Block

# CPU Switch From Process to Process

- This is also called a “context switch”
- Code executed in kernel above is overhead
  - Overhead sets minimum practical switching time
  - Less overhead with SMT/hyperthreading, but… contention for resources instead

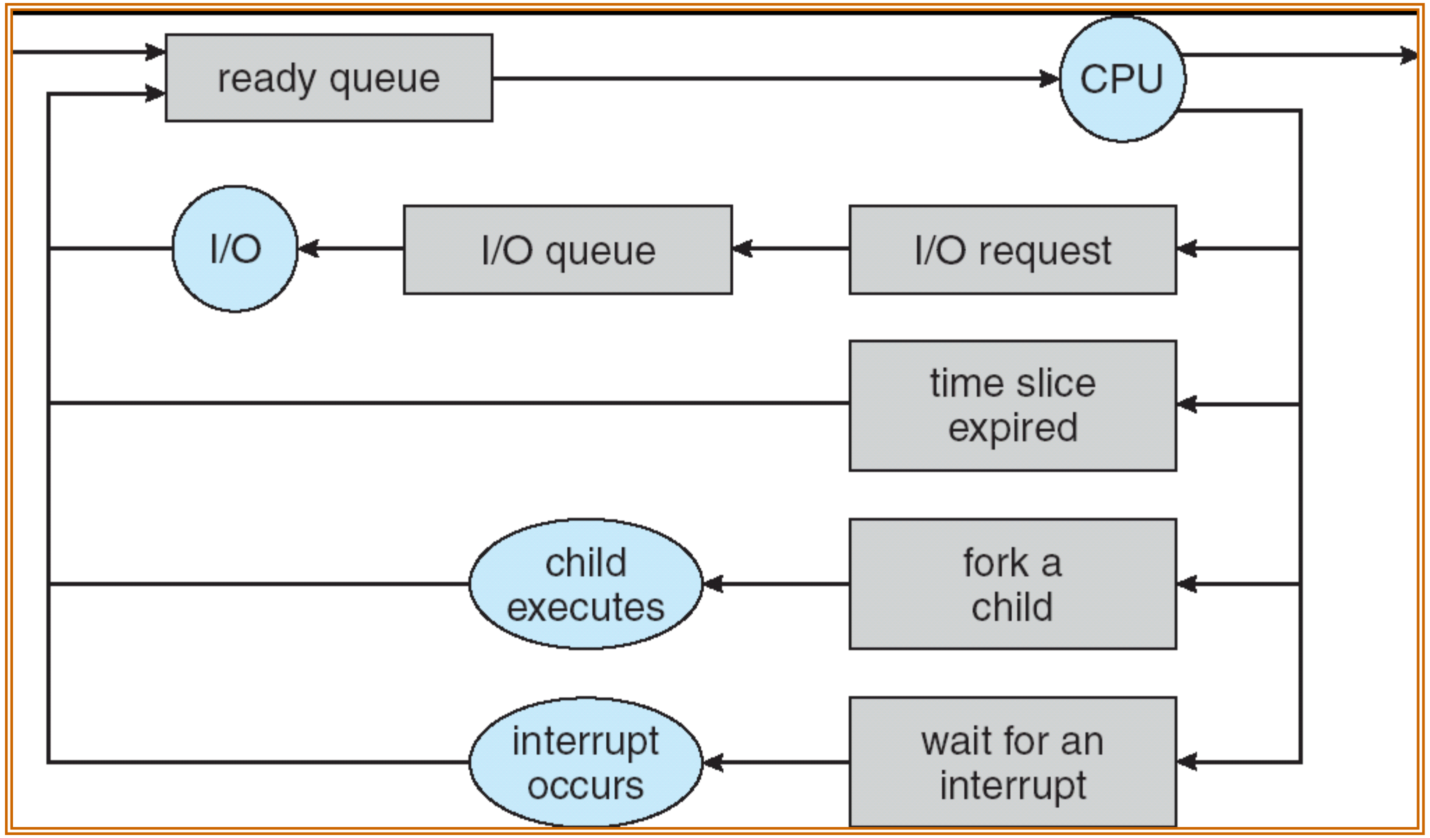


# Lifecycle of a Process




- 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

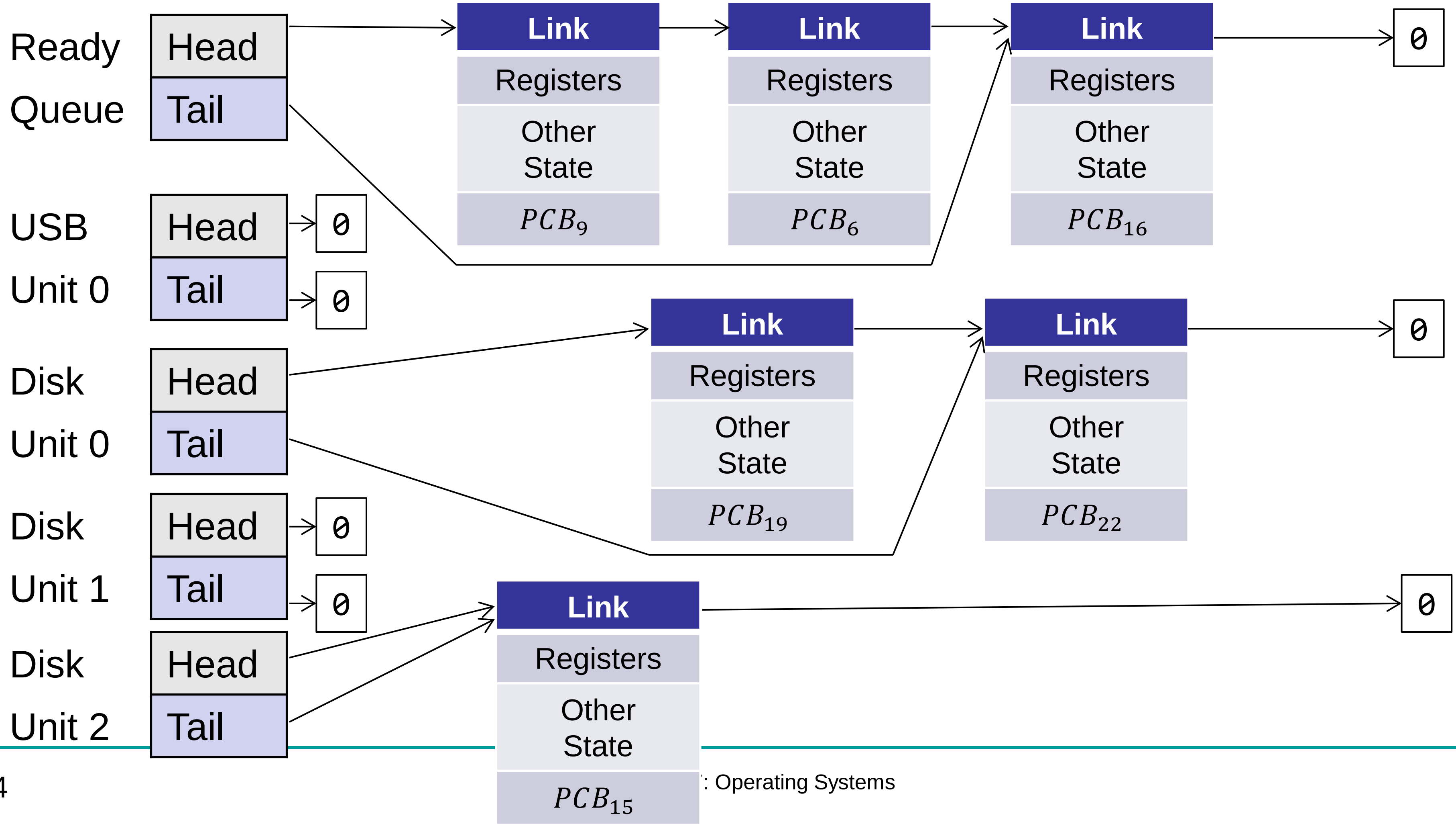
# Process Scheduling




- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are Scheduling decisions
  - Many algorithms possible (later)

# Ready Queue And Various I/O Device Queues

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

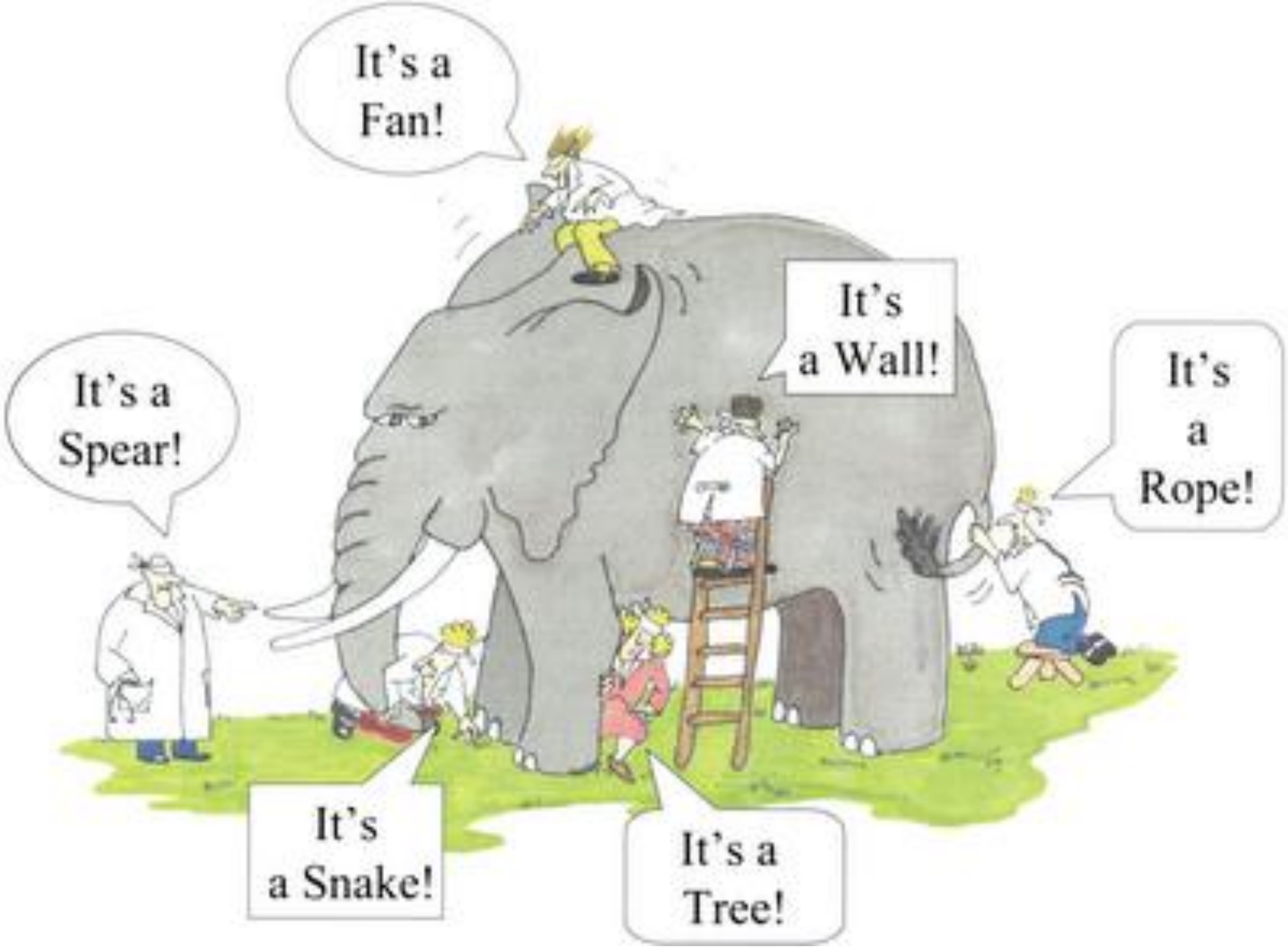


# So Far

- Concurrency: Processes
- Threads
  - Kernel Supported Threads vs. User Supported Threads
  - Switching between threads
- Concurrency: Putting it together
- Cooperating threads
- Concurrency challenge

# About Threads

Image source: https://maddy06.blogspot.com/2012/10/six-blind-men-and-elephant.html



# Modern Process with Threads

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*

Why separate the concept of a thread from that of a process?

- The "thread" part of a process (concurrency)
- Separate from the "address space" (protection)

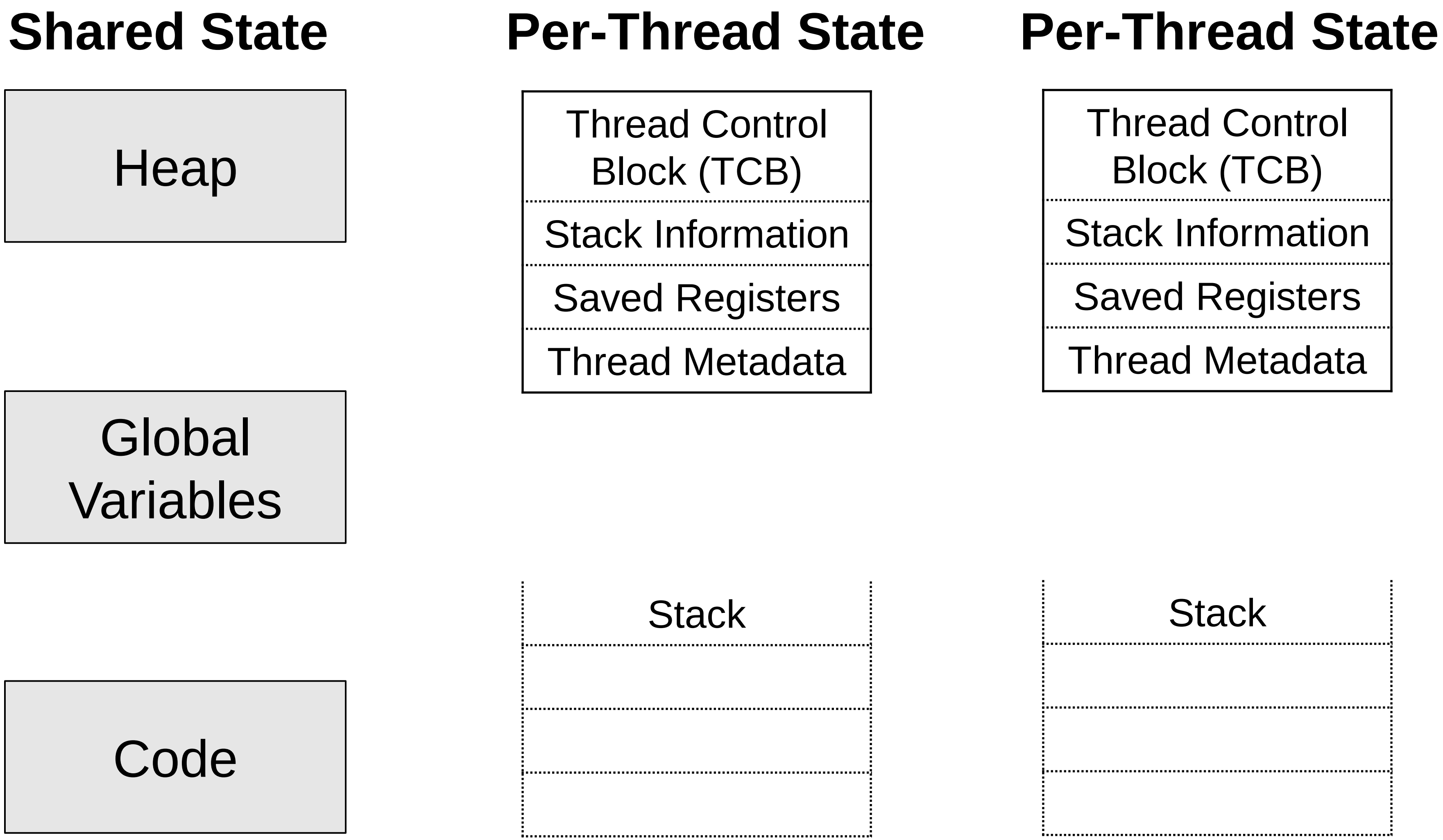
# Per Thread Descriptor (Kernel Supported Threads)

- 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) – user threads

- OS Keeps track of TCBs in "kernel memory"
  - In array, or linked list, or …
  - I/O state (file descriptors, network connections, etc)

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

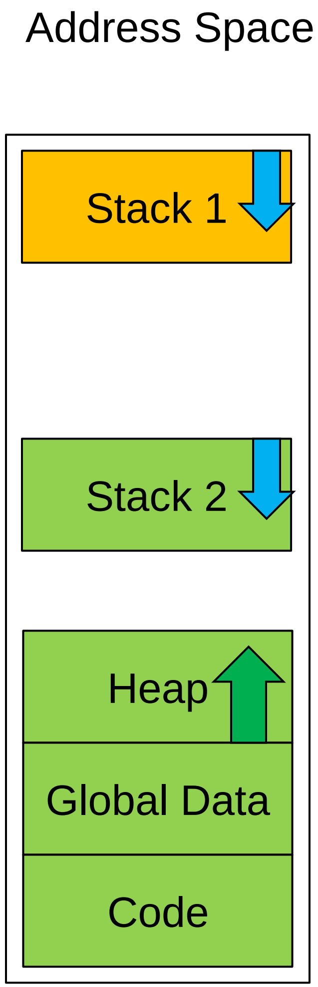
# Shared vs. Per-Thread State



# Memory Footprint: Two Threads

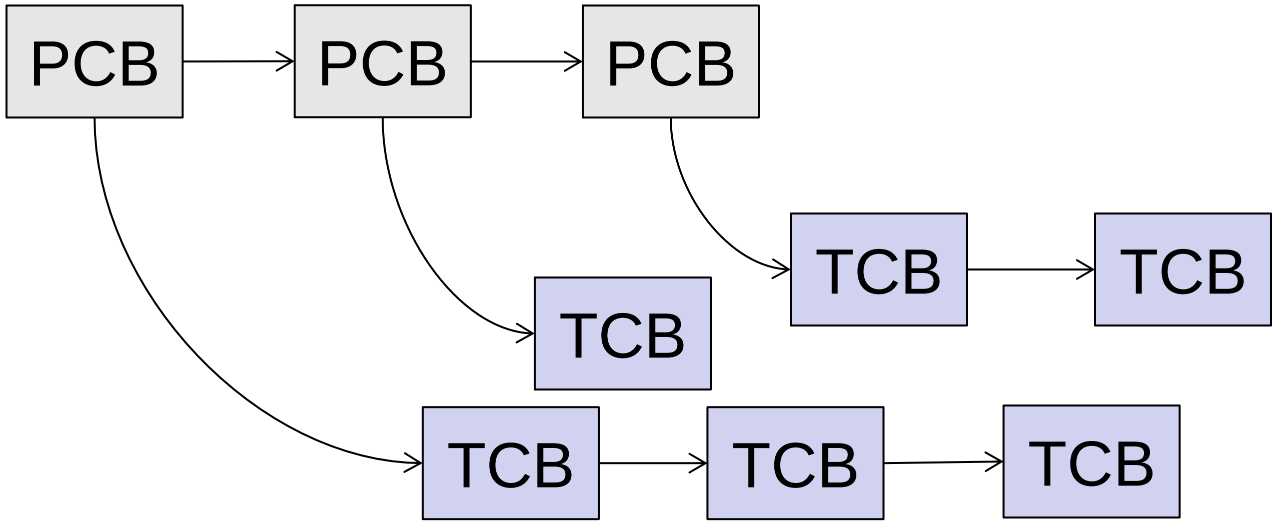
- If we stopped a program and examined with a debugger, we would see
  - Two sets of CPU registers
  - Two stacks

- 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?



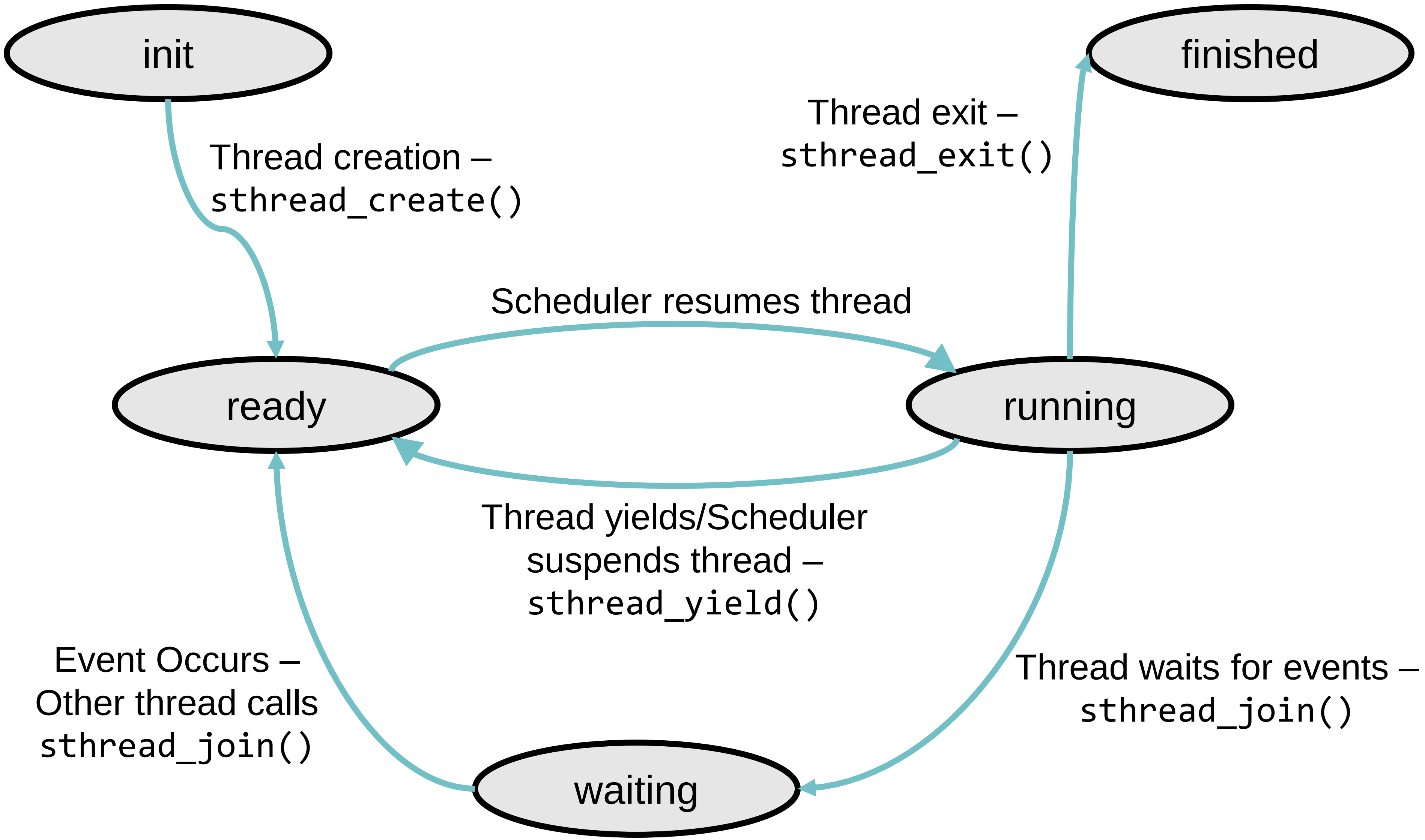
# Multithreaded Processes

- PCB points to multiple TCBs:

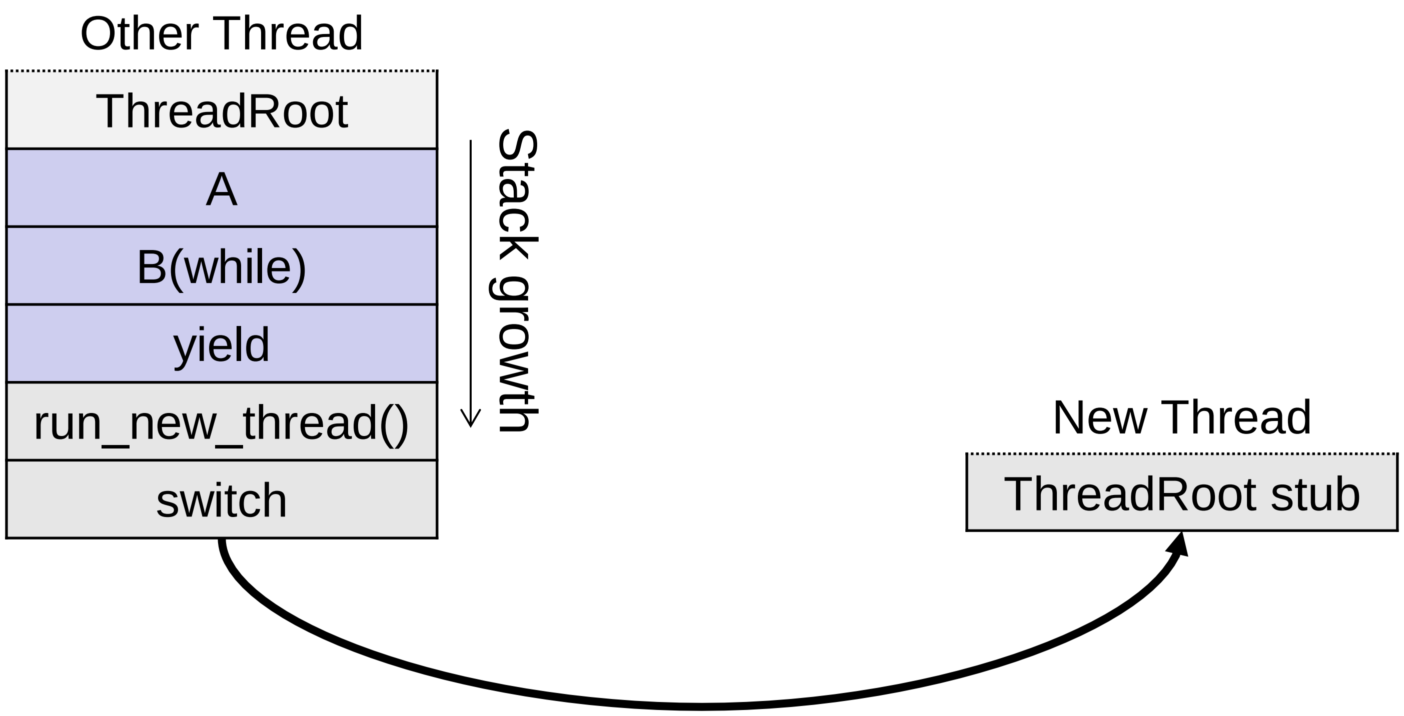



- Switching threads within a block is a simple thread switch
- Switching threads across blocks requires changes to memory and I/O address tables.

# Thread Lifecycle



# How does a Thread get started?

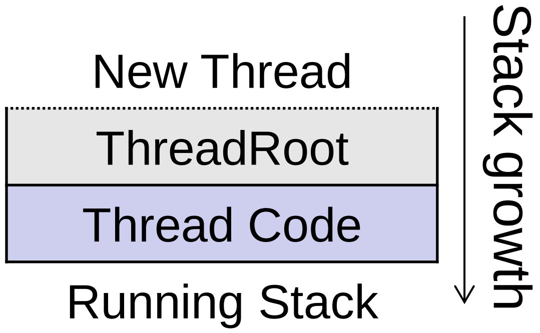



- Eventually, `run_new_thread()` will select this TCB and return into beginning of `ThreadRoot()`
  - This really starts the new thread

# What does `ThreadRoot()` look like?

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

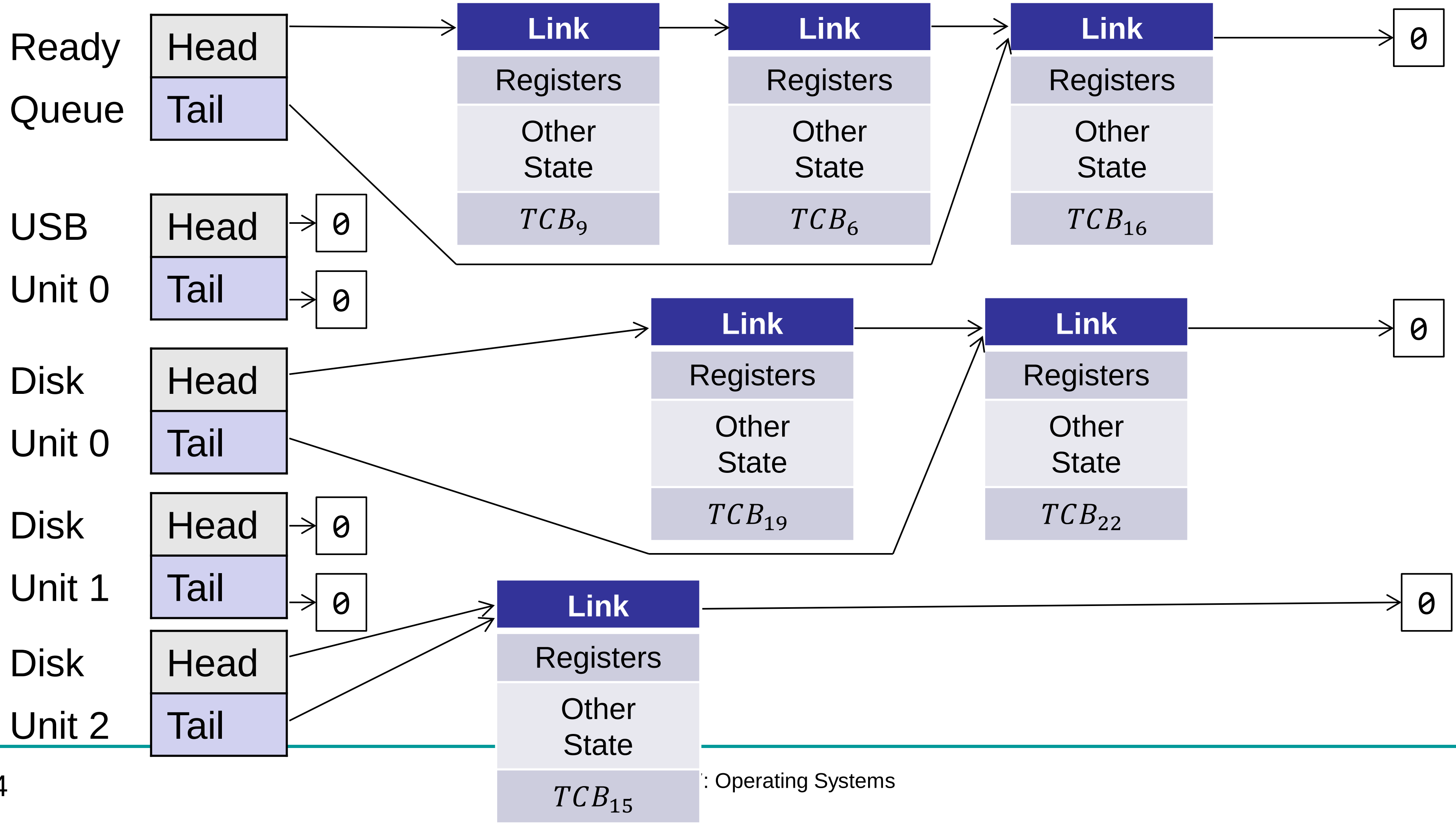
```
ThreadRoot() {
   DoStartupHousekeeping();
   UserModeSwitch(); /* enter user mode*/
   Call fcnPtr(fcnArgPtr);
   ThreadFinish();
}
```




- Startup Housekeeping
  - Includes things like recording start time of thread and other statistics
- Stack will grow and shrink with execution of thread

# Ready Queue And Various I/O Device Queues

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



# Common Thread Operations

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

- Note: These aren't really the names of the functions.

- pThreads: POSIX standard for thread programming

# 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?

# Running a thread

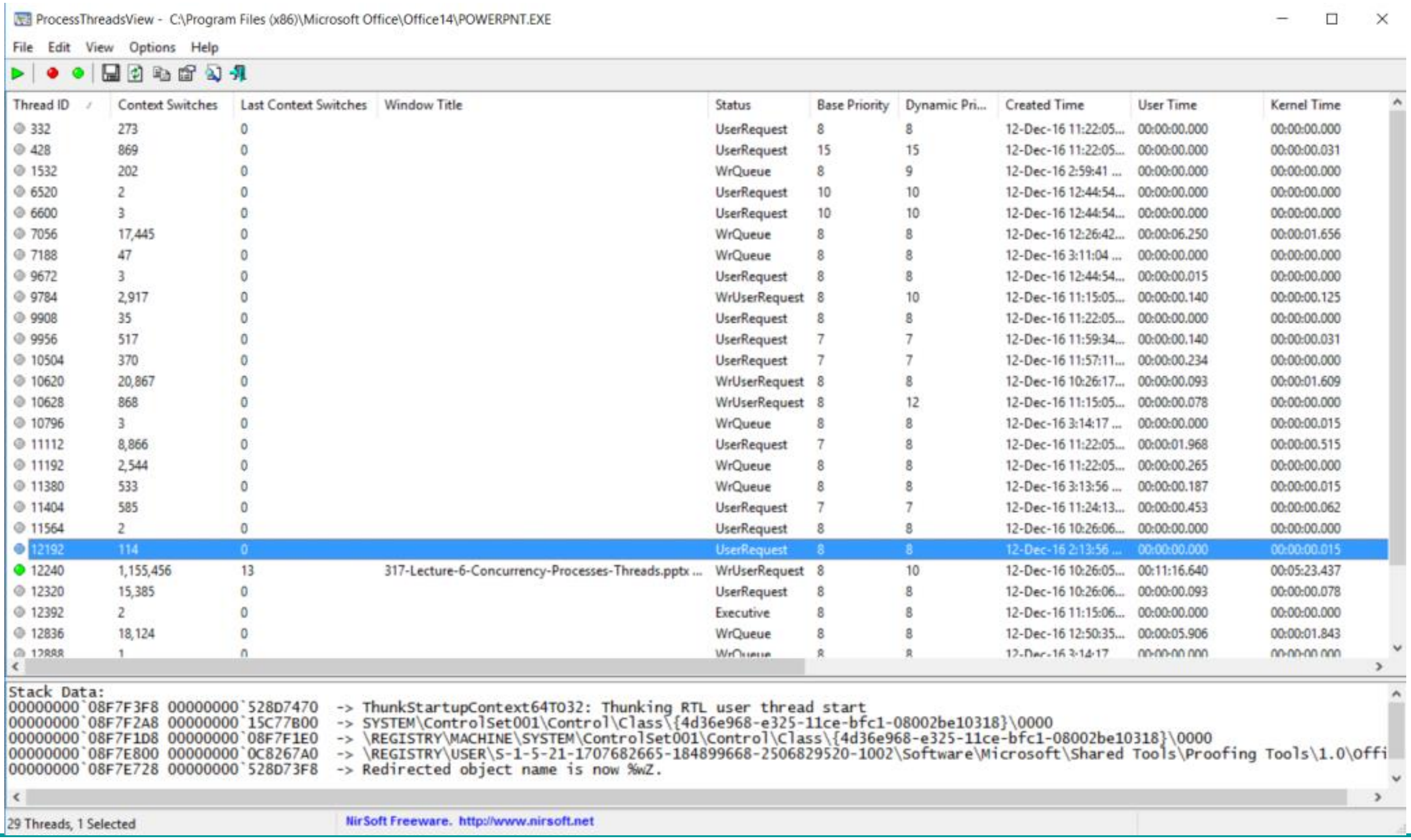
Consider first portion: `RunThread()`

- 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

# Some Actual Numbers

- Many process are multi-threaded, so thread context switches may be either **within-process** or **across-processes**.



ProcessThreadsView - C:\Program Files (x86)\Microsoft Office\Office14\POWERPNT.EXE

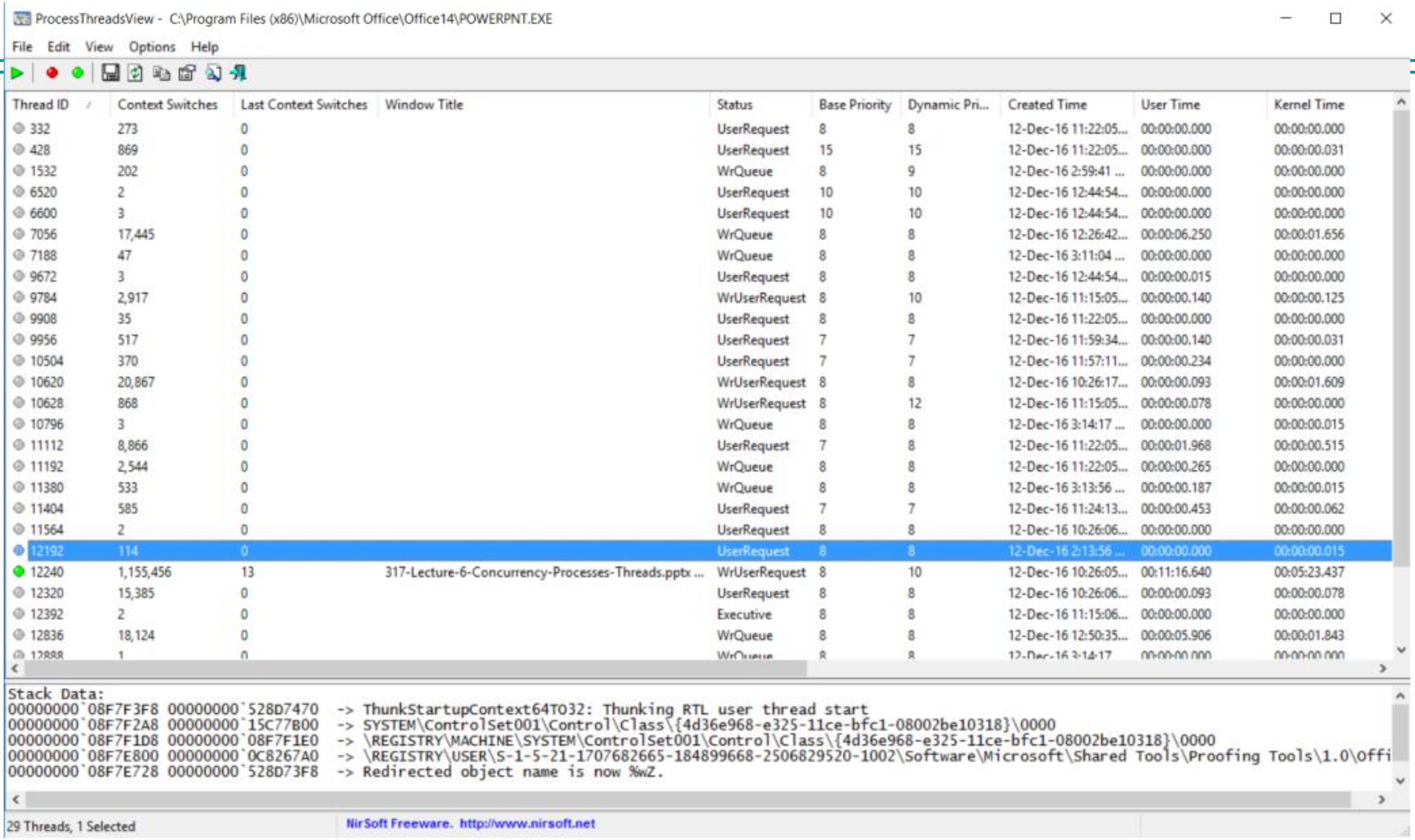
File Edit View Options Help

| Thread ID | Context Switches | Last Context Switches | Window Title | Status | Base Priority | Dynamic Pri... | Created Time | User Time | Kernel Time |
|---|---|---|---|---|---|---|---|---|---|
| 332 | 273 | 0 | | UserRequest | 8 | 8 | 12-Dec-16 11:22:05... | 00:00:00.000 | 00:00:00.000 |
| 428 | 869 | 0 | | UserRequest | 15 | 15 | 12-Dec-16 11:22:05... | 00:00:00.000 | 00:00:00.031 |
| 1532 | 202 | 0 | | WrQueue | 8 | 9 | 12-Dec-16 2:59:41 ... | 00:00:00.000 | 00:00:00.000 |
| 6520 | 2 | 0 | | UserRequest | 10 | 10 | 12-Dec-16 12:44:54... | 00:00:00.000 | 00:00:00.000 |
| 6600 | 3 | 0 | | UserRequest | 10 | 10 | 12-Dec-16 12:44:54... | 00:00:00.000 | 00:00:00.000 |
| 7056 | 17,445 | 0 | | WrQueue | 8 | 8 | 12-Dec-16 12:26:42... | 00:00:06.250 | 00:00:01.656 |
| 7188 | 47 | 0 | | WrQueue | 8 | 8 | 12-Dec-16 3:11:04 ... | 00:00:00.000 | 00:00:00.000 |
| 9672 | 3 | 0 | | UserRequest | 8 | 8 | 12-Dec-16 12:44:54... | 00:00:00.015 | 00:00:00.000 |
| 9784 | 2,917 | 0 | | WrUserRequest | 8 | 10 | 12-Dec-16 11:15:05... | 00:00:00.140 | 00:00:00.125 |
| 9908 | 35 | 0 | | UserRequest | 8 | 8 | 12-Dec-16 11:22:05... | 00:00:00.000 | 00:00:00.000 |
| 9956 | 517 | 0 | | UserRequest | 7 | 7 | 12-Dec-16 11:59:34... | 00:00:00.140 | 00:00:00.031 |
| 10504 | 370 | 0 | | UserRequest | 7 | 7 | 12-Dec-16 11:57:11... | 00:00:00.234 | 00:00:00.000 |
| 10620 | 20,867 | 0 | | WrUserRequest | 8 | 8 | 12-Dec-16 10:26:17... | 00:00:00.093 | 00:00:01.609 |
| 10628 | 868 | 0 | | WrUserRequest | 8 | 12 | 12-Dec-16 11:15:05... | 00:00:00.078 | 00:00:00.000 |
| 10796 | 3 | 0 | | WrQueue | 8 | 8 | 12-Dec-16 3:14:17 ... | 00:00:00.000 | 00:00:00.015 |
| 11112 | 8,866 | 0 | | UserRequest | 7 | 8 | 12-Dec-16 11:22:05... | 00:00:01.968 | 00:00:00.515 |
| 11192 | 2,544 | 0 | | WrQueue | 8 | 8 | 12-Dec-16 11:22:05... | 00:00:00.265 | 00:00:00.000 |
| 11380 | 533 | 0 | | WrQueue | 8 | 8 | 12-Dec-16 3:13:56 ... | 00:00:00.187 | 00:00:00.015 |
| 11404 | 585 | 0 | | UserRequest | 7 | 7 | 12-Dec-16 11:24:13... | 00:00:00.453 | 00:00:00.062 |
| 11564 | 2 | 0 | | UserRequest | 8 | 8 | 12-Dec-16 10:26:06... | 00:00:00.000 | 00:00:00.000 |
| 12192 | 114 | 0 | | UserRequest | 8 | 8 | 12-Dec-16 2:13:56 ... | 00:00:00.000 | 00:00:00.015 |
| 12240 | 1,155,456 | 13 | 317-Lecture-6-Concurrency-Processes-Threads.pptx ... | WrUserRequest | 8 | 10 | 12-Dec-16 10:26:05... | 00:11:16.640 | 00:05:23.437 |
| 12320 | 15,385 | 0 | | UserRequest | 8 | 8 | 12-Dec-16 10:26:06... | 00:00:00.093 | 00:00:00.078 |
| 12392 | 2 | 0 | | Executive | 8 | 8 | 12-Dec-16 11:15:06... | 00:00:00.000 | 00:00:00.000 |
| 12836 | 18,124 | 0 | | WrQueue | 8 | 8 | 12-Dec-16 12:50:35... | 00:00:05.906 | 00:00:01.843 |
| 12888 | 1 | 0 | | WrQueue | 8 | 8 | 12-Dec-16 3:14:17 | 00:00:00.000 | 00:00:00.000 |

```
Stack Data:
00000000`08F7F3F8 00000000`528D7470  -> ThunkStartupContext64TO32: Thunking RTL user thread start
00000000`08F7F2A8 00000000`15C77B00  -> SYSTEM\ControlSet001\Control\Class\{4d36e968-e325-11ce-bfc1-08002be10318}\0000
00000000`08F7F1D8 00000000`08F7F1E0  -> \REGISTRY\MACHINE\SYSTEM\ControlSet001\Control\Class\{4d36e968-e325-11ce-bfc1-08002be10318}\0000
00000000`08F7E800 00000000`0C8267A0  -> \REGISTRY\USER\S-1-5-21-1707682665-184899668-2506829520-1002\Software\Microsoft\Shared Tools\Proofing Tools\1.0\Offi
00000000`08F7E728 00000000`528D73F8  -> Redirected object name is now %wZ.
```

29 Threads, 1 Selected

NirSoft Freeware. http://www.nirsoft.net



ProcessThreadsView - C:\Program Files (x86)\Microsoft Office\Office14\POWERPNT.EXE

File Edit View Options Help

| Thread ID | Context Switches | Last Context Switches | Window Title | Status | Base Priority | Dynamic Pri... | Created Time | User Time | Kernel Time |
|---|---|---|---|---|---|---|---|---|---|
| 332 | 273 | 0 | | UserRequest | 8 | 8 | 12-Dec-16 11:22:05... | 00:00:00.000 | 00:00:00.000 |
| 428 | 869 | 0 | | UserRequest | 15 | 15 | 12-Dec-16 11:22:05... | 00:00:00.000 | 00:00:00.031 |
| 1532 | 202 | 0 | | WrQueue | 8 | 9 | 12-Dec-16 2:59:41 ... | 00:00:00.000 | 00:00:00.000 |
| 6520 | 2 | 0 | | UserRequest | 10 | 10 | 12-Dec-16 12:44:54... | 00:00:00.000 | 00:00:00.000 |
| 6600 | 3 | 0 | | UserRequest | 10 | 10 | 12-Dec-16 12:44:54... | 00:00:00.000 | 00:00:00.000 |
| 7056 | 17,445 | 0 | | WrQueue | 8 | 8 | 12-Dec-16 12:26:42... | 00:00:06.250 | 00:00:01.656 |
| 7188 | 47 | 0 | | WrQueue | 8 | 8 | 12-Dec-16 3:11:04 ... | 00:00:00.000 | 00:00:00.000 |
| 9672 | 3 | 0 | | UserRequest | 8 | 8 | 12-Dec-16 12:44:54... | 00:00:00.015 | 00:00:00.000 |
| 9784 | 2,917 | 0 | | WrUserRequest | 8 | 10 | 12-Dec-16 11:15:05... | 00:00:00.140 | 00:00:00.125 |
| 9908 | 35 | 0 | | UserRequest | 8 | 8 | 12-Dec-16 11:22:05... | 00:00:00.000 | 00:00:00.000 |
| 9956 | 517 | 0 | | UserRequest | 7 | 7 | 12-Dec-16 11:59:34... | 00:00:00.140 | 00:00:00.031 |
| 10504 | 370 | 0 | | UserRequest | 7 | 7 | 12-Dec-16 11:57:11... | 00:00:00.234 | 00:00:00.000 |
| 10620 | 20,867 | 0 | | WrUserRequest | 8 | 8 | 12-Dec-16 10:26:17... | 00:00:00.093 | 00:00:01.609 |
| 10628 | 868 | 0 | | WrUserRequest | 8 | 12 | 12-Dec-16 11:15:05... | 00:00:00.078 | 00:00:00.000 |
| 10796 | 3 | 0 | | WrQueue | 8 | 8 | 12-Dec-16 3:14:17 ... | 00:00:00.000 | 00:00:00.015 |
| 11112 | 8,866 | 0 | | UserRequest | 7 | 8 | 12-Dec-16 11:22:05... | 00:00:01.968 | 00:00:00.515 |
| 11192 | 2,544 | 0 | | WrQueue | 8 | 8 | 12-Dec-16 11:22:05... | 00:00:00.265 | 00:00:00.000 |
| 11380 | 533 | 0 | | WrQueue | 8 | 8 | 12-Dec-16 3:13:56 ... | 00:00:00.187 | 00:00:00.015 |
| 11404 | 585 | 0 | | UserRequest | 7 | 7 | 12-Dec-16 11:24:13... | 00:00:00.453 | 00:00:00.062 |
| 11564 | 2 | 0 | | UserRequest | 8 | 8 | 12-Dec-16 10:26:06... | 00:00:00.000 | 00:00:00.000 |
| 12192 | 114 | 0 | | UserRequest | 8 | 8 | 12-Dec-16 2:13:56 ... | 00:00:00.000 | 00:00:00.015 |
| 12240 | 1,155,456 | 13 | 317-Lecture-6-Concurrency-Processes-Threads.pptx ... | WrUserRequest | 8 | 10 | 12-Dec-16 10:26:05... | 00:11:16.640 | 00:05:23.437 |
| 12320 | 15,385 | 0 | | UserRequest | 8 | 8 | 12-Dec-16 10:26:06... | 00:00:00.093 | 00:00:00.078 |
| 12392 | 2 | 0 | | Executive | 8 | 8 | 12-Dec-16 11:15:06... | 00:00:00.000 | 00:00:00.000 |
| 12836 | 18,124 | 0 | | WrQueue | 8 | 8 | 12-Dec-16 12:50:35... | 00:00:05.906 | 00:00:01.843 |
| 12888 | 1 | 0 | | WrQueue | 8 | 8 | 12-Dec-16 3:14:17 | 00:00:00.000 | 00:00:00.000 |

```
Stack Data:
00000000`08F7F3F8 00000000`528D7470  -> ThunkStartupContext64TO32: Thunking RTL user thread start
00000000`08F7F2A8 00000000`15C77B00  -> SYSTEM\ControlSet001\Control\Class\{4d36e968-e325-11ce-bfc1-08002be10318}\0000
00000000`08F7F1D8 00000000`08F7F1E0  -> \REGISTRY\MACHINE\SYSTEM\ControlSet001\Control\Class\{4d36e968-e325-11ce-bfc1-08002be10318}\0000
00000000`08F7E800 00000000`0C8267A0  -> \REGISTRY\USER\S-1-5-21-1707682665-184899668-2506829520-1002\Software\Microsoft\Shared Tools\Proofing Tools\1.0\Offi
00000000`08F7E728 00000000`528D73F8  -> Redirected object name is now %wZ.
```

29 Threads, 1 Selected    NirSoft Freeware. http://www.nirsoft.net

# So Far

- Concurrency: Processes
- Threads
  - Kernel Supported Threads vs. User Supported Threads
  - Switching between threads
- Concurrency: Putting it together
- Cooperating threads
- Concurrency challenge

# Kernel Supported vs User Supported threads

- We have been talking about Kernel supported threads
  - Commonly called “Kernel mode 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 cross into kernel mode to schedule

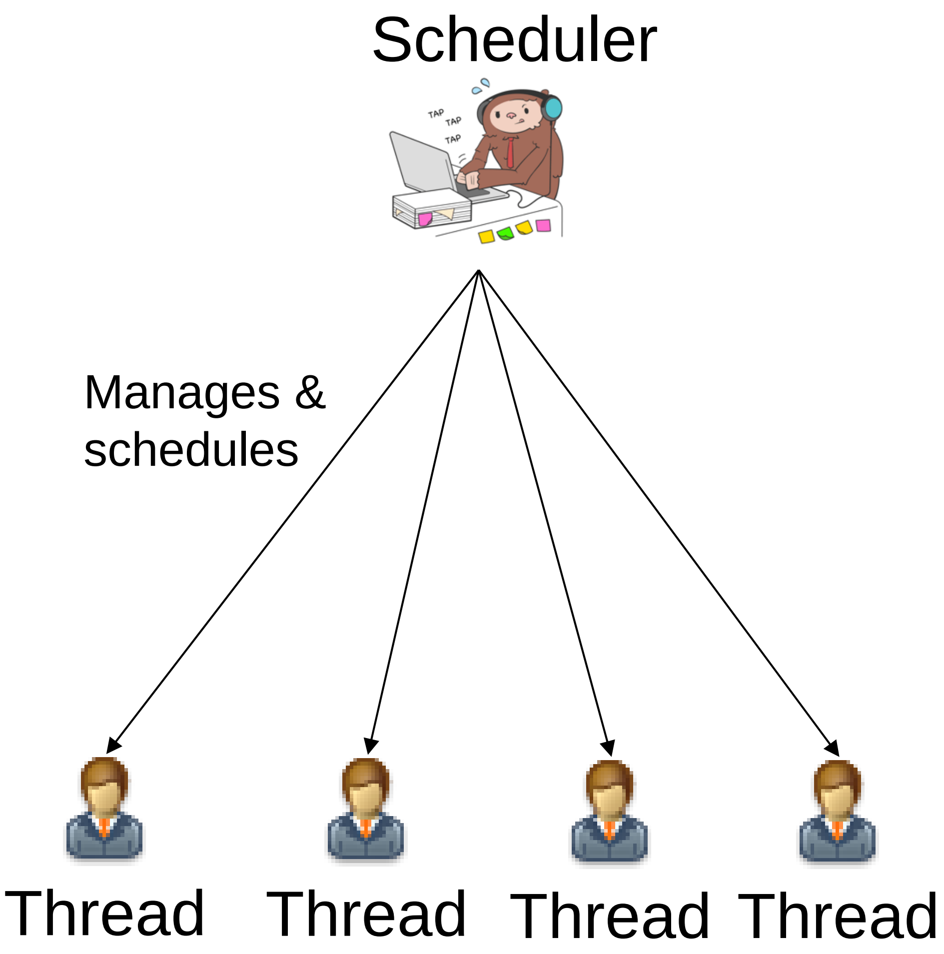
- Even lighter weight option: User Threads
  - User program provides scheduler and thread package

# User Supported Threads

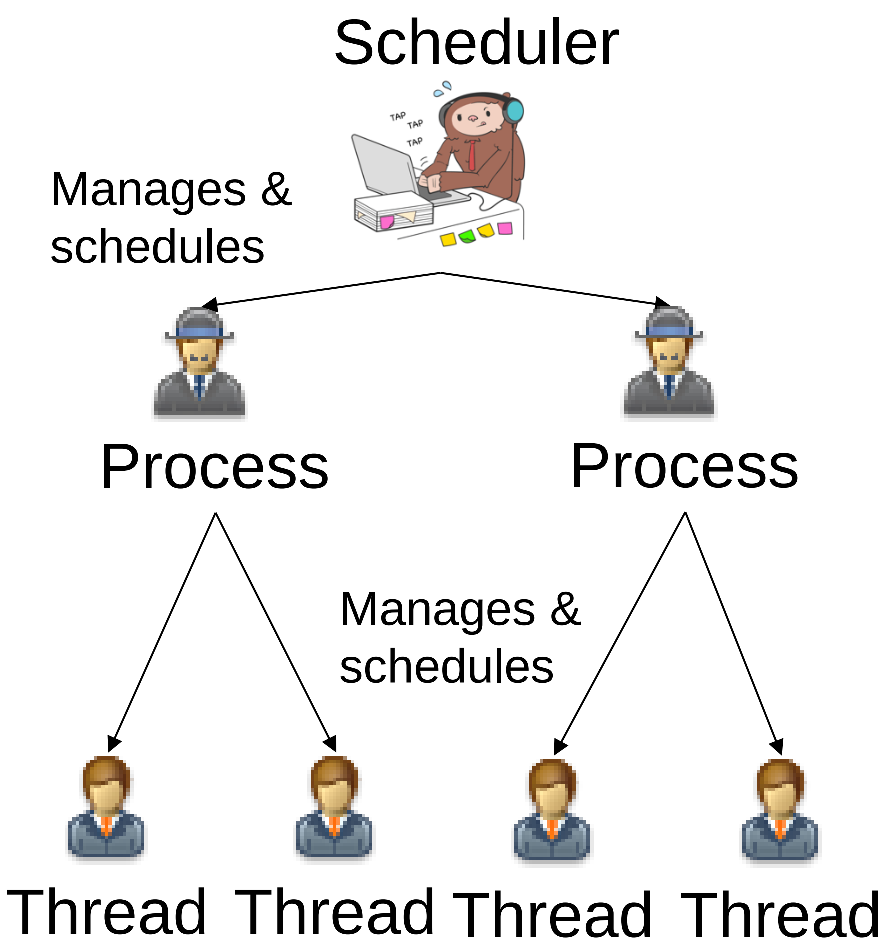
- User Supported Threads:
  - Commonly called “User mode threads”
  - May have several user threads per kernel thread
  - User threads may be scheduled non-preemptively relative to each other (only switch on `yield()`)
- Downsides of user threads:
  - When one thread blocks on I/O, all threads block
  - Kernel cannot adjust scheduling among all threads
- Option: *Scheduler Activations and Upcalls*
  - Have kernel inform user level when thread blocks, e.g. a signal

# Imagining it

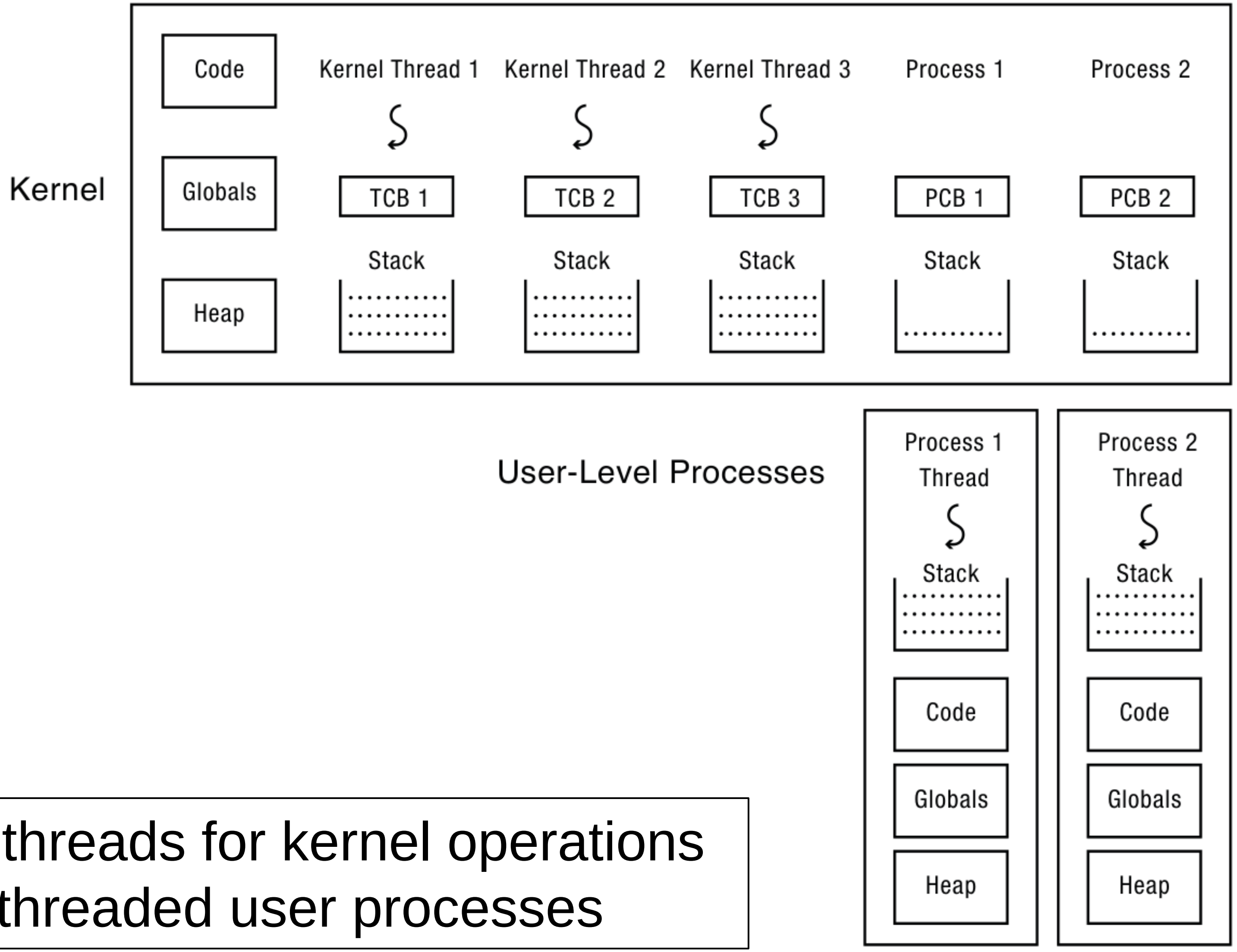
**Kernel Supported**




**User Supported**

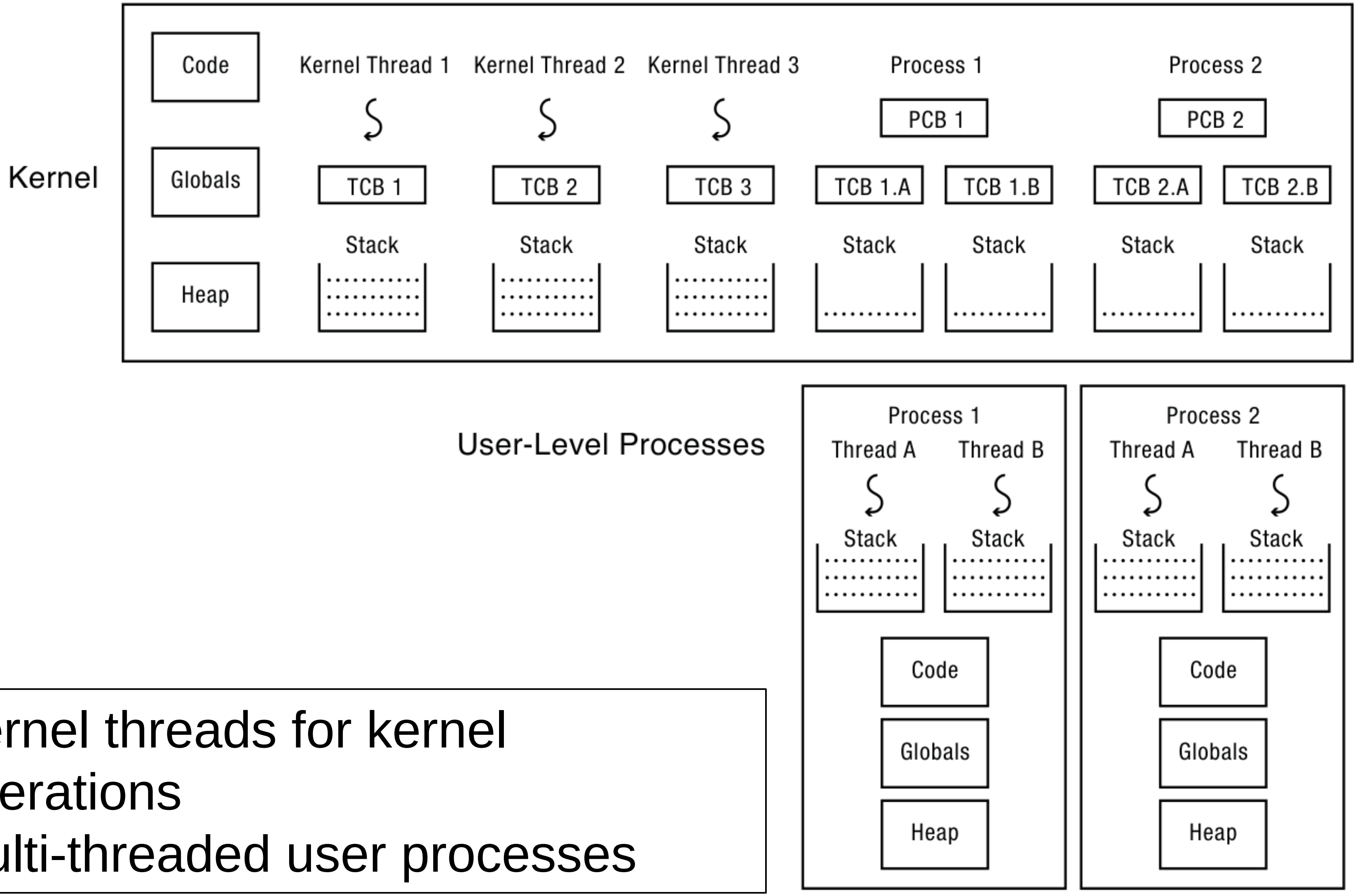


# Simple One-to-One Threading Mode



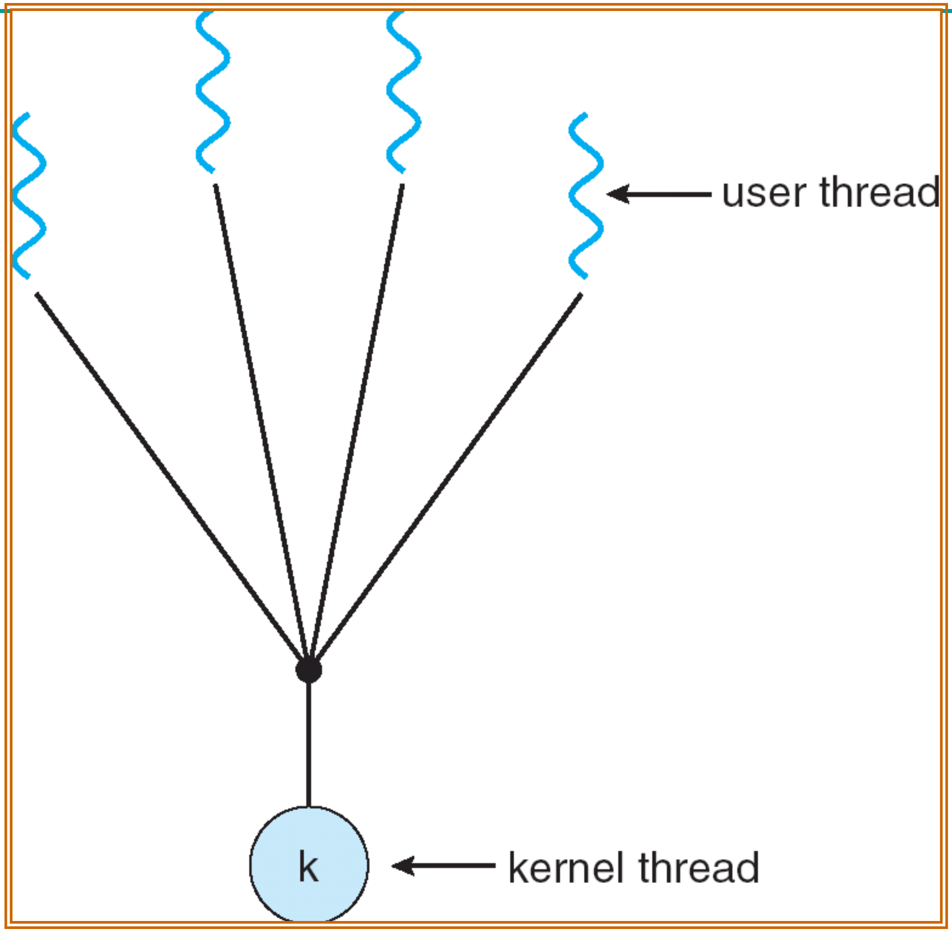

Kernel threads for kernel operations
Single threaded user processes

# Simple One-to-One Threading Mode



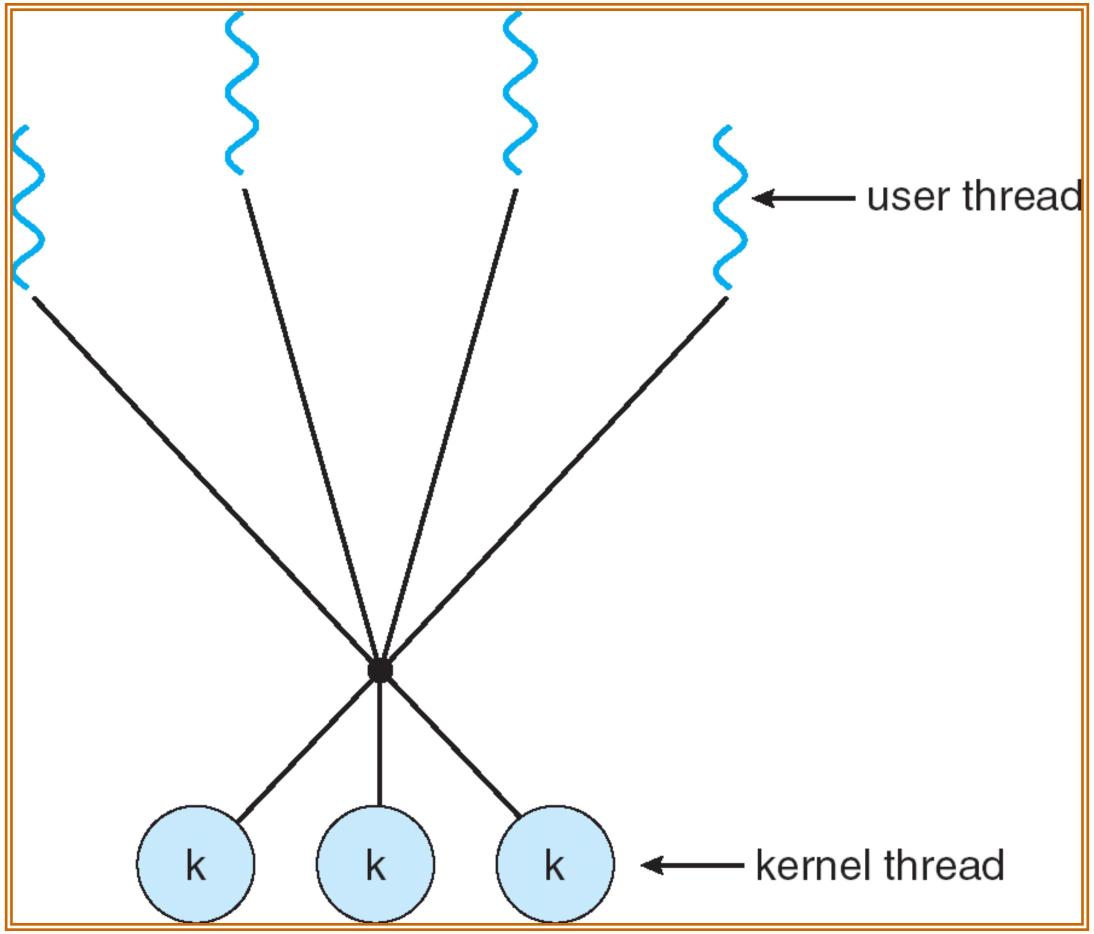

Kernel threads for kernel operations
Multi-threaded user processes

# Alternative Thread Models




Many-to-One

# Alternative Thread Models




Many-to-Many

# (Old) Thread Strategies

Option A (early Java):

- User-level library, within a single-threaded process
- Library does thread context switch
- Kernel time slices between processes, e.g., on system call I/O

Option B (SunOS, Unix vars):

- Called Green Threads
- User-level library does thread multiplexing
- Issues with I/O waiting and multi-core

# (Modern) Thread Strategies

## Option C (Windows):

- Technique: Scheduler Activations
- Kernel allocates processors to user-level library
- Thread library implements context switch
- System call I/O that blocks triggers upcall

## Option D (Linux, MacOS, Windows):

- Full Kernel Supported Threads
- System calls for thread fork, join, exit (and lock, unlock,…)
- Kernel does context switching
- Many user ←→ kernel mode transitions

# So Far

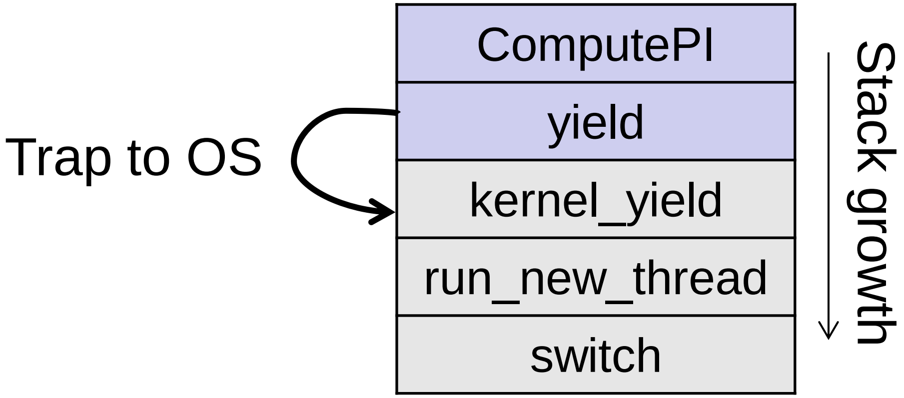
- Concurrency: Processes
- Threads
  - Kernel Supported Threads vs. User Supported Threads
  - Switching between threads
- Concurrency: Putting it together
- Cooperating threads
- Concurrency challenge

# Interrupting a Thread (internal)

- 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 switch to a new thread?

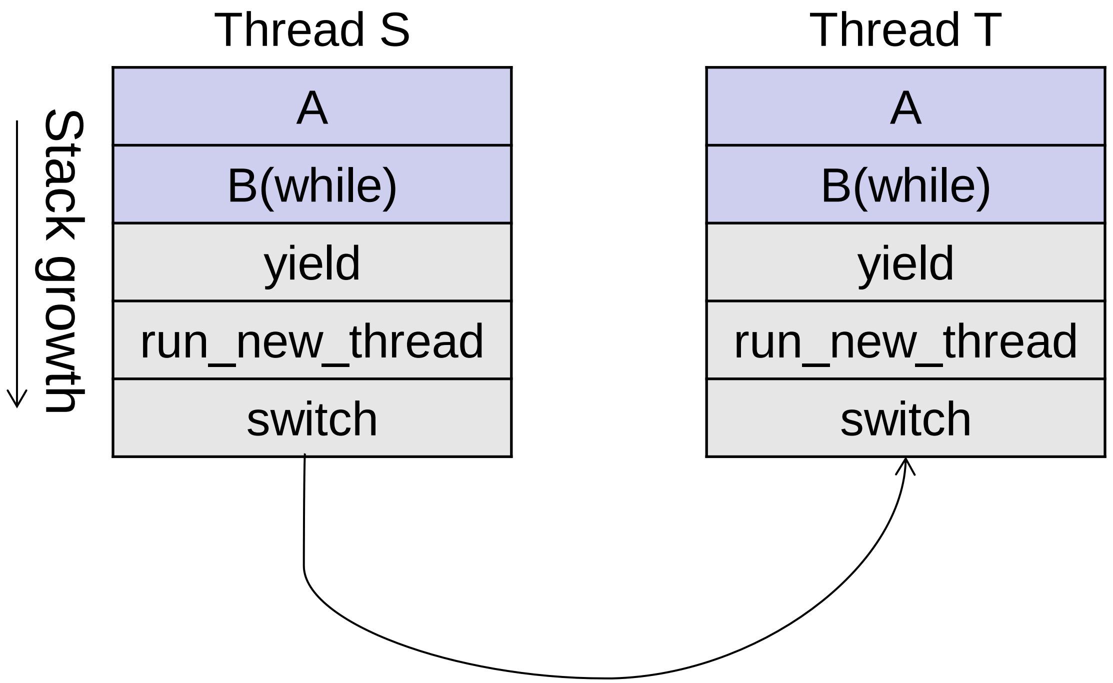
```
run_new_thread() {
  newThread = PickNewThread();
  switch(curThread, newThread);
  ThreadHouseKeeping(); /* Do any cleanup */
}
```

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

# Switch

Image source: https://tenor.com/view/superman-clark-kent-chest-ready-gif-14642505

# 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 */
}
```

# From UNIX V6, slp.c (Dennis Ritchie) on scheduling

```
/*
 * If the new process paused because it was
 * swapped out, set the stack level to the last call
 * to savu(u_ssav).  This means that the return
 * which is executed immediately after the call to aretu
 * actually returns from the last routine which did
 * the savu.
 *
 * You are not expected to understand this.
 */
```

https://github.com/hephaex/unix-v6/blob/master/ken/slp.c

# Switch Details

- What if you make a mistake in implementing switch?
  - Suppose you forget to save/restore register 4
  - Get intermittent failures depending on when context switch occurred and whether new thread uses register 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 tale

- For speed, Topaz kernel (UNIX variant by DEC) 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

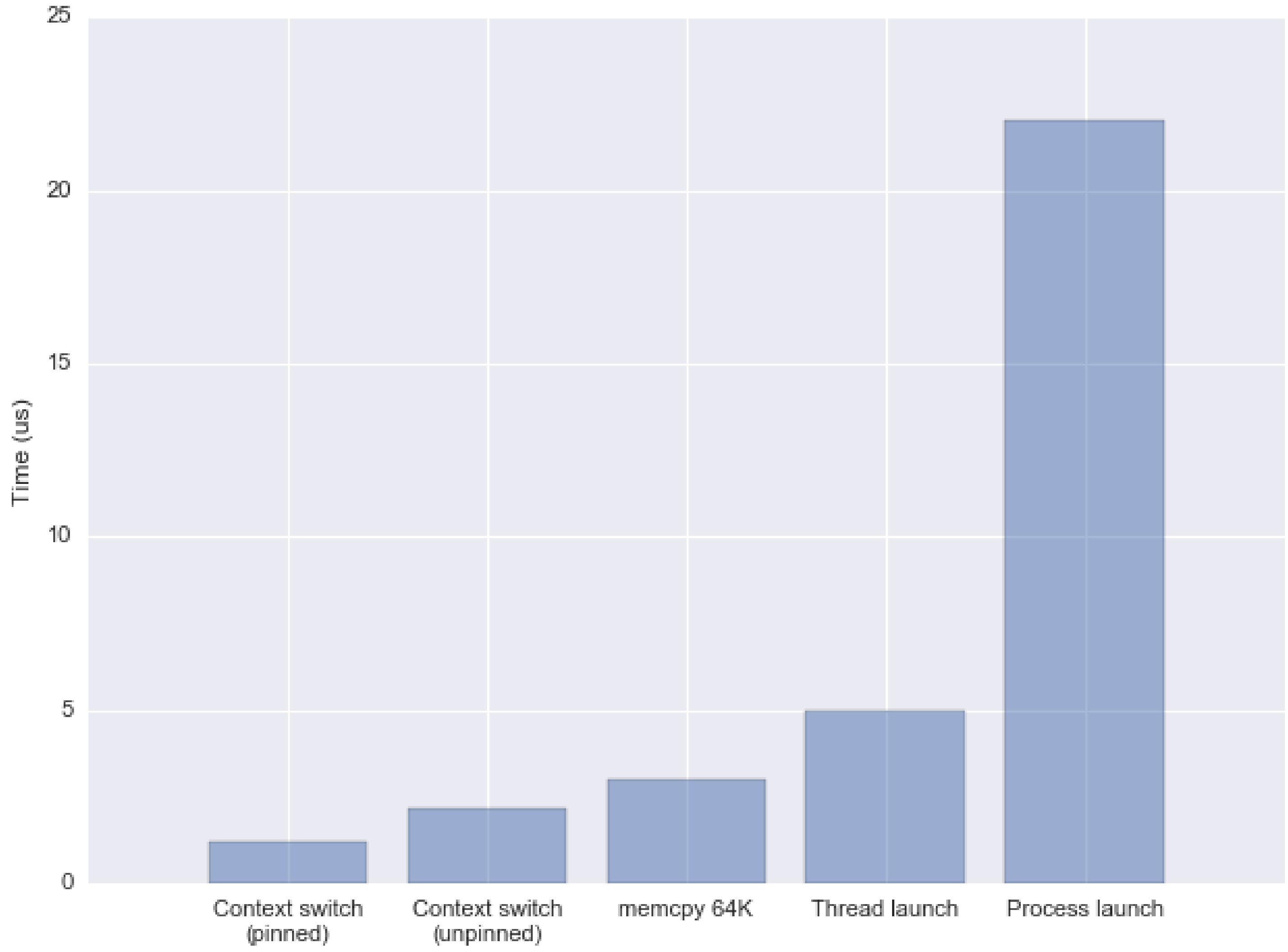
- Moral: Design for simplicity

# Some Numbers

- Frequency of performing context switches: 10-100ms
- Context switch time in Linux: **1.2-1.5 μsecs** **(2018)** (https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/).
  - Thread switching faster than process switching (**100 ns**).
  - But switching across cores about 2x more expensive than within-core switching.
- Context switch time increases sharply with the size of the working set and can increase 100x or more.
  - The *working set* is the subset of memory used by the process in a time window.

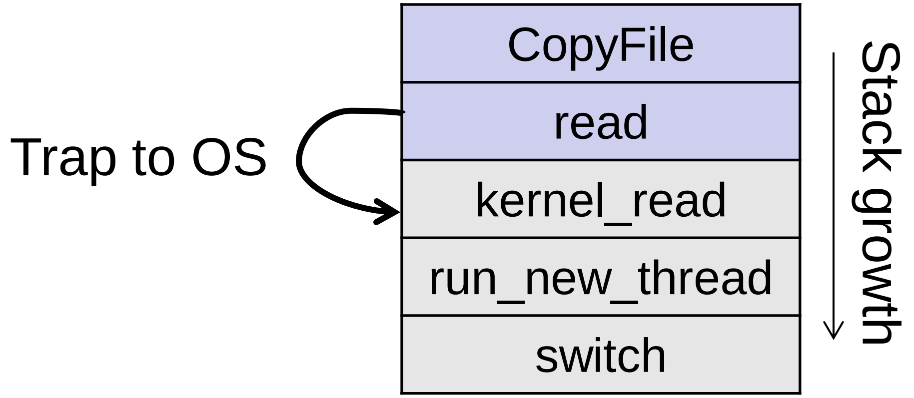
- Moral: Context switching depends mostly on cache limits and the process or thread's hunger for memory.

# Comparing times




https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/

# What happens when thread blocks on I/O?



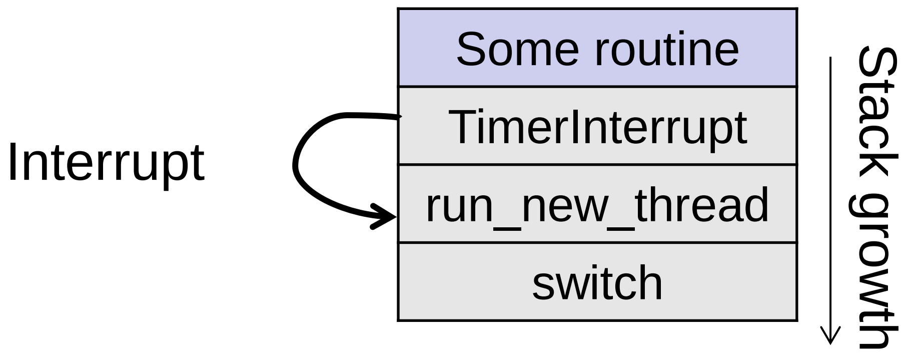

- What happens when a thread requests a block of data from the file system?
  - User code invokes a system call
  - Read operation is initiated
  - Run new thread/switch
- Thread communication similar
  - Wait for Signal/Join
  - Networking

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

# 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();
}
```

# Example: Network Interrupt

```
add $r1, $r2, $r3
sub $r4, $r1, #4
sll $r4, $r4, #2
```

External Interrupt

Pipeline Flush

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

PC Saved
Disable all ints
Supervisor mode

Restore PC
User Mode

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

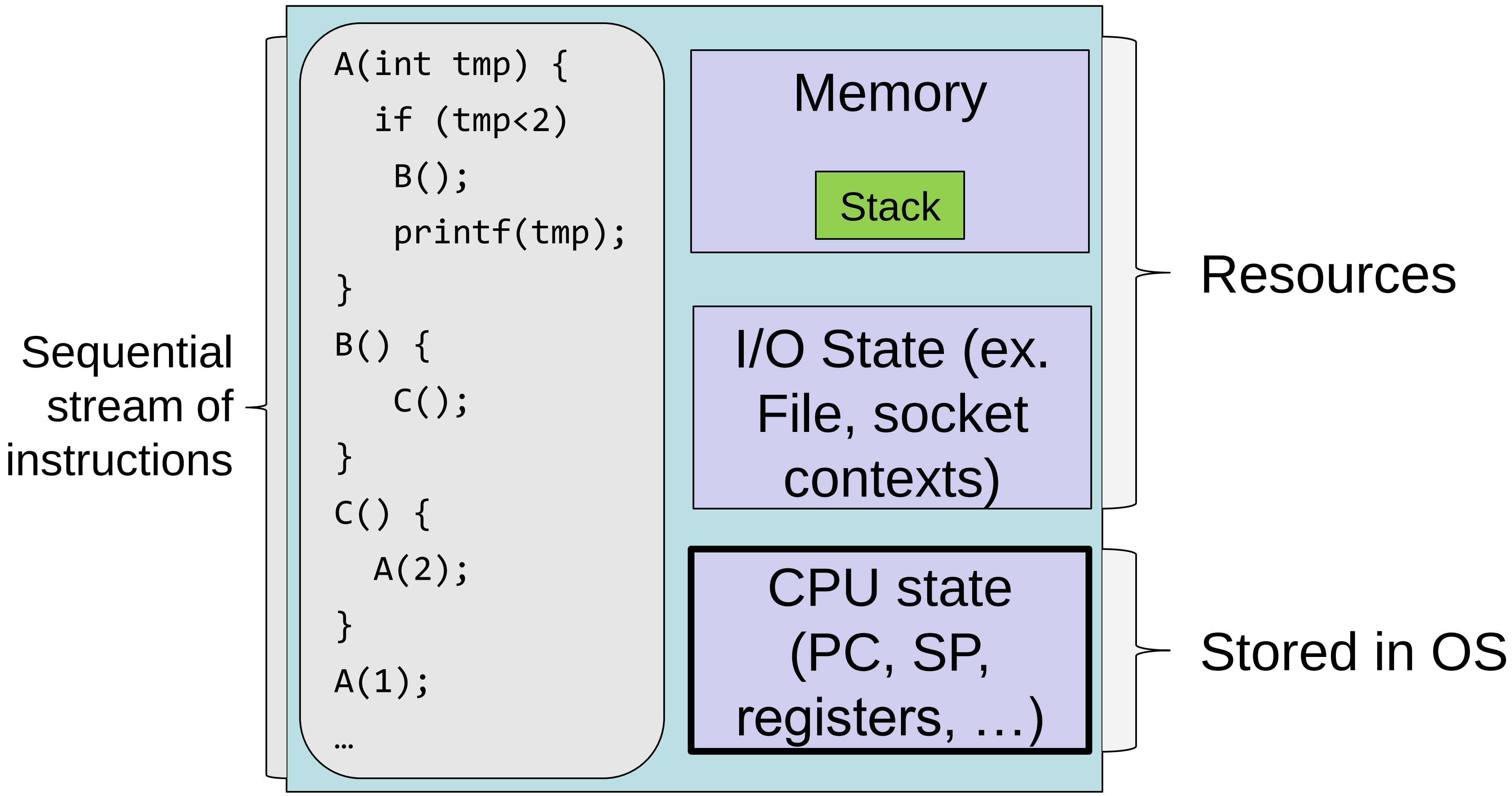
Interrupt Handler

# So Far

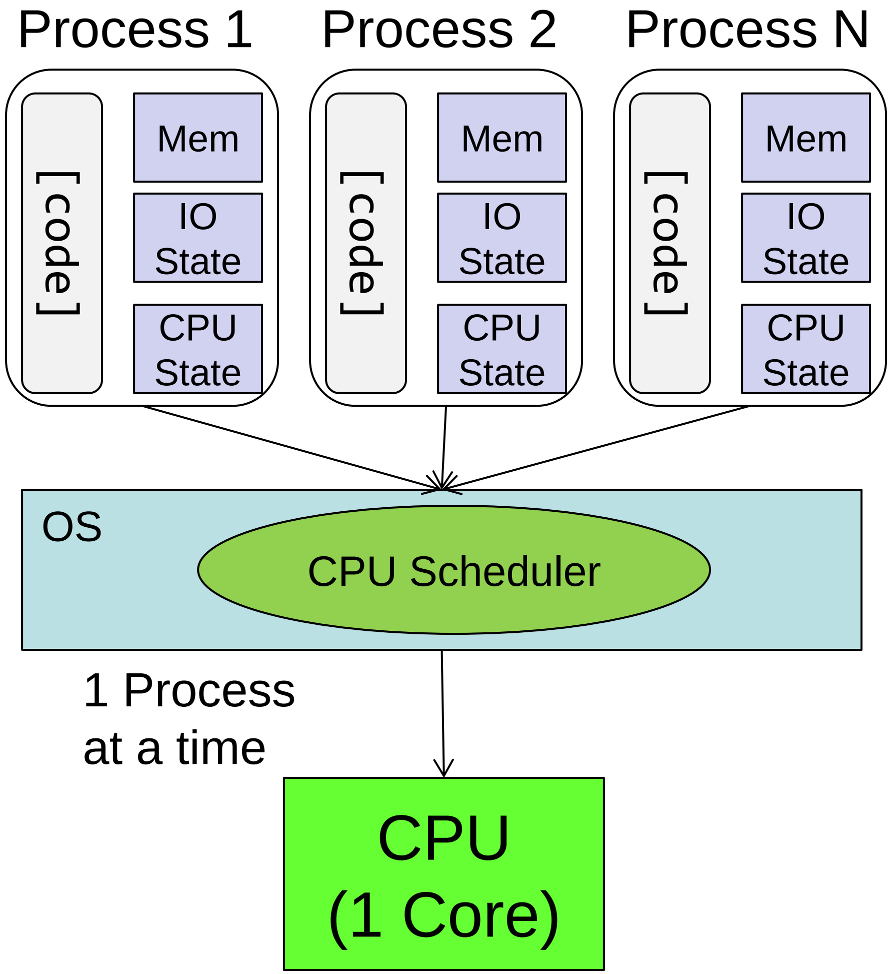
- Concurrency: Processes
- Threads
  - Kernel Supported Threads vs. User Supported Threads
  - Switching between threads
- Concurrency: Putting it together
- Cooperating threads
- Concurrency challenge

# Putting it together: Process

(UNIX) Process

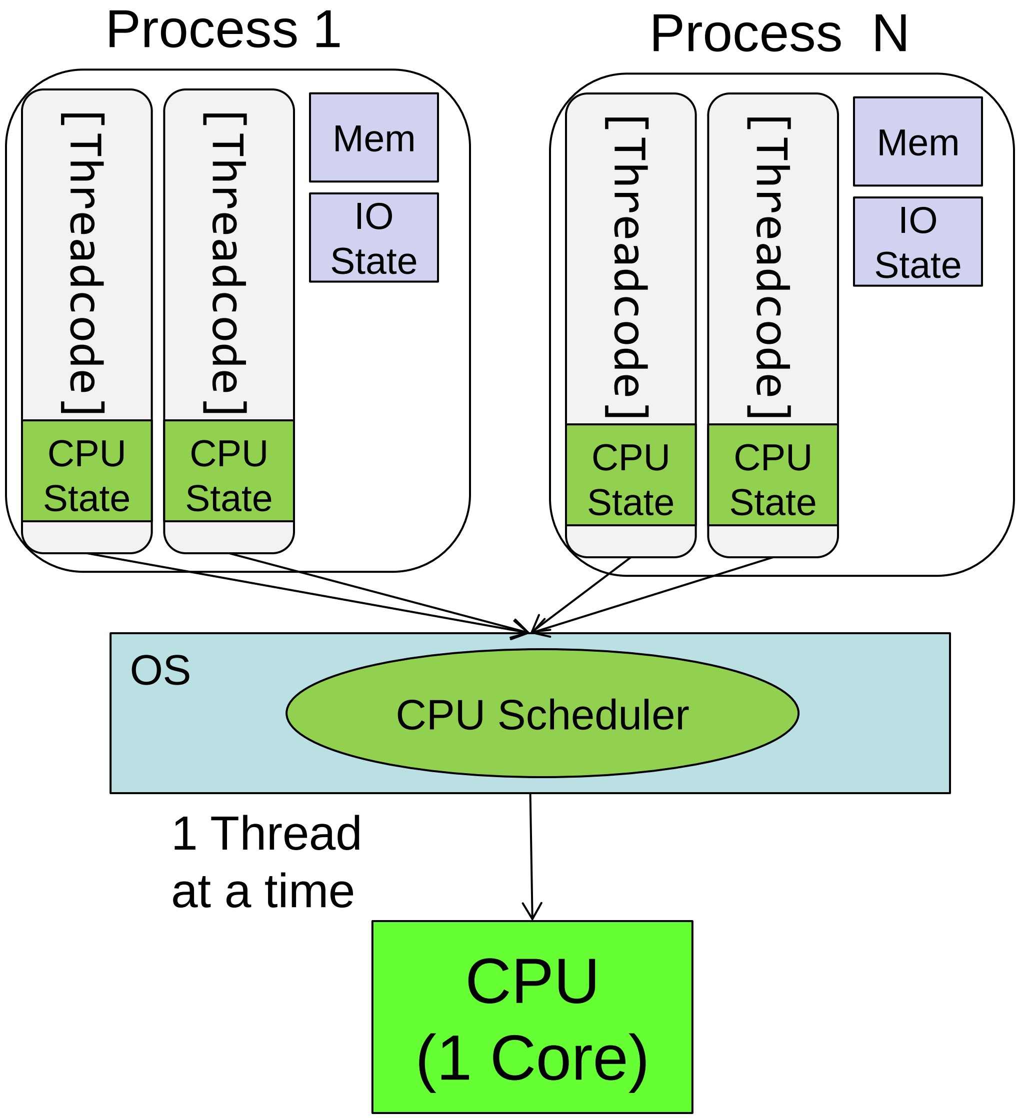


# Putting it together: Processes



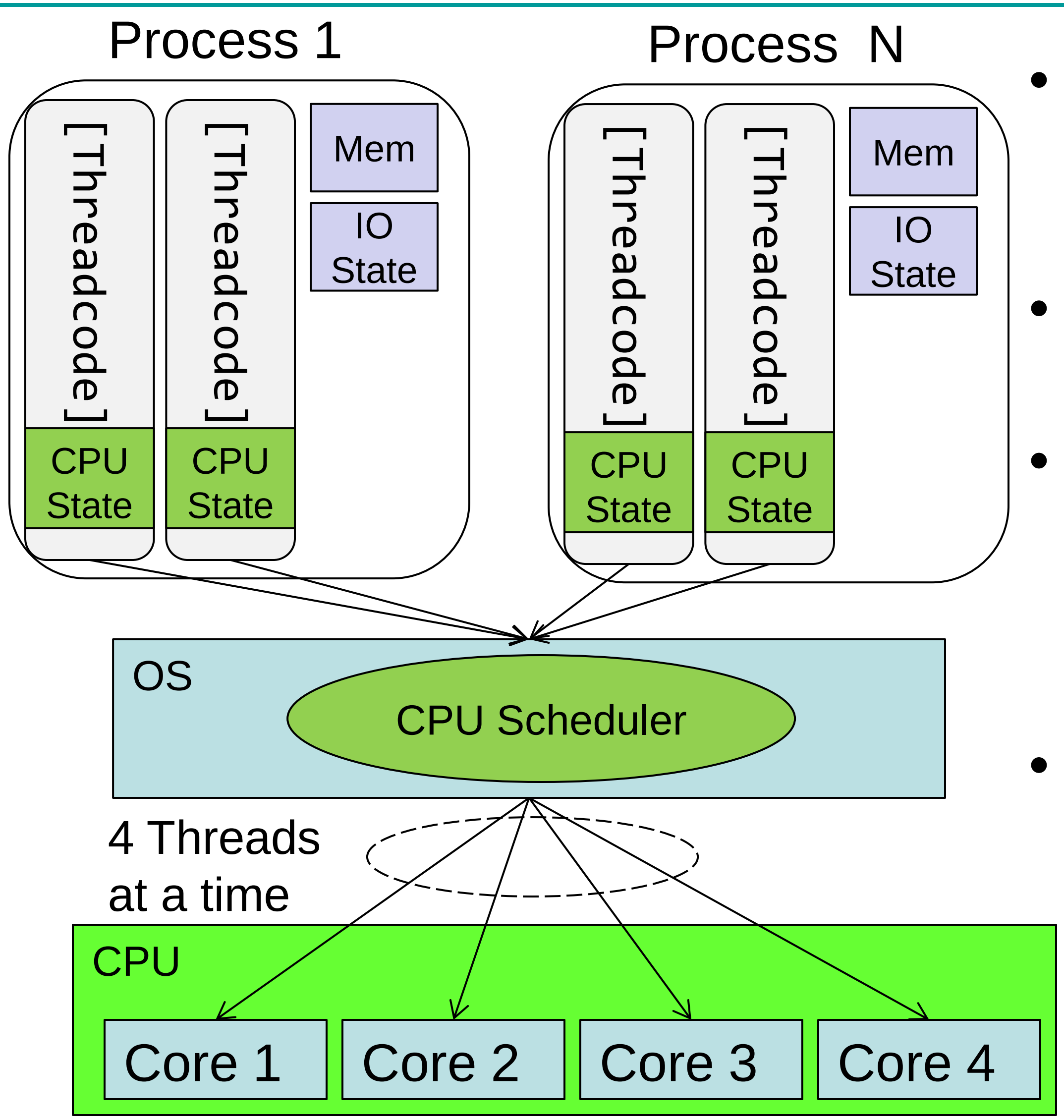

- Switch overhead: high
  - CPU state: **low**
  - Memory/IO state: high
- Process creation: high
- Protection
  - CPU: **yes**
  - Memory/IO: **yes**
- Sharing overhead: high (involves at least a context switch)

# Putting it together: Threads



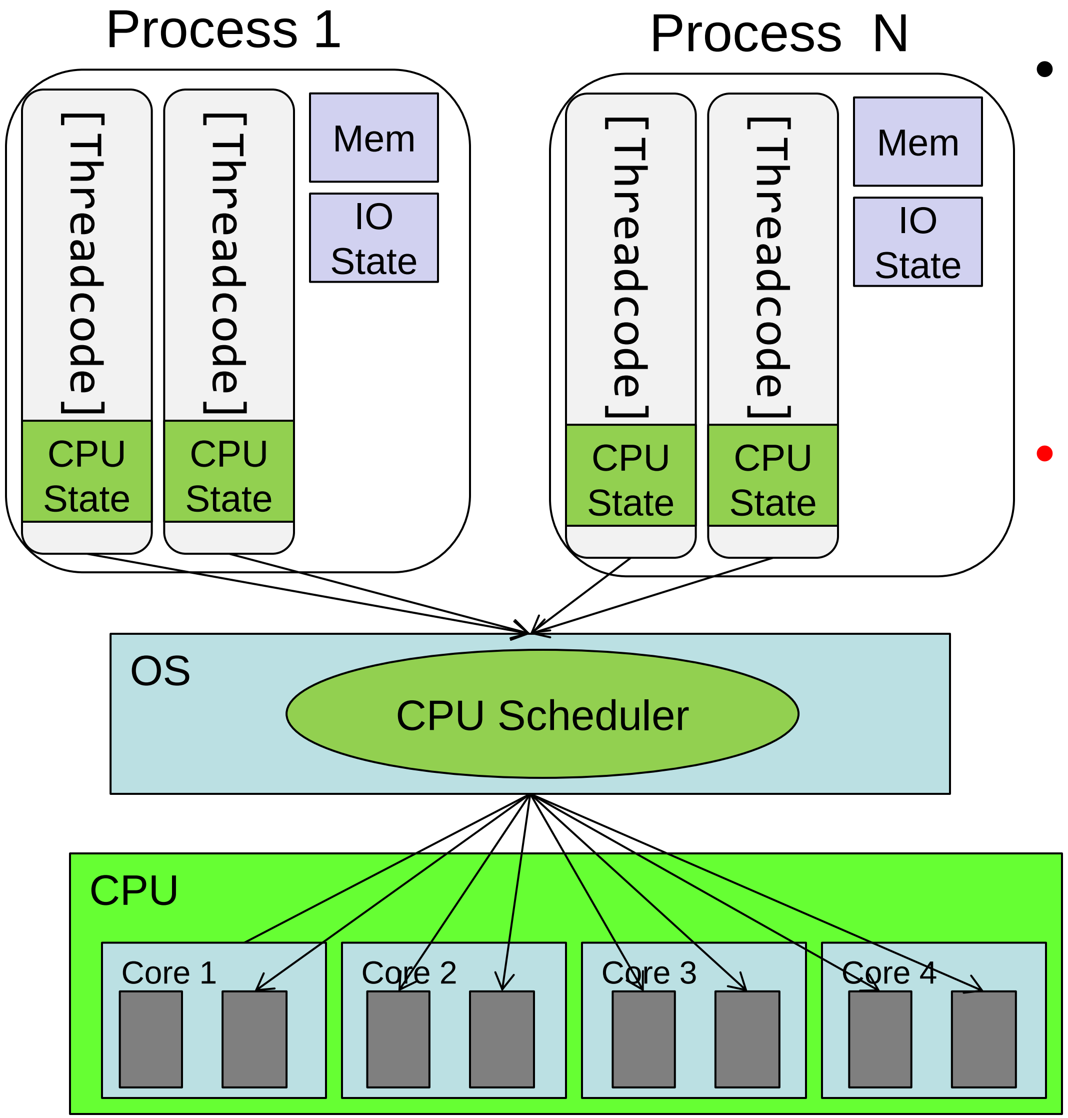

- Switch overhead: low (Only CPU state)
- Thread creation: low
- Protection
  - CPU: **yes**
  - Memory/IO: **no**
- Sharing overhead: low (thread switch overhead low)

# 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, may not need to switch at all)

# Putting it together: Hyper-Threading




- System overhead between hardware-threads: very low (done in hardware)
- Contention for ALUs/FPUs may hurt performance

Hardware threads (SMT or hyper-threading)

8 Threads at a time

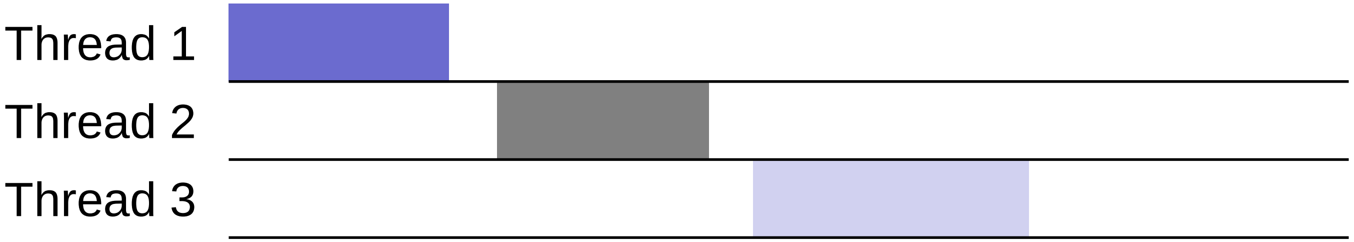
# So Far

- Concurrency: Processes
- Threads
  - Kernel Supported Threads vs. User Supported Threads
  - Switching between threads
- Concurrency: Putting it together
- Cooperating threads
- Concurrency challenge

# Programmer vs. Processor View

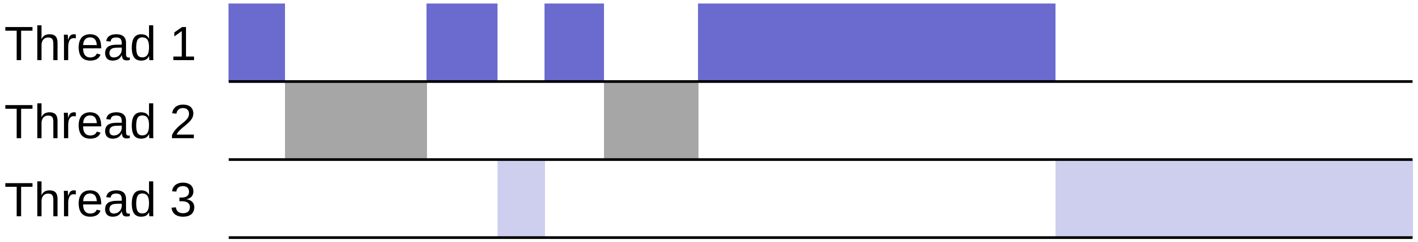
| Programmer's View | Possible Execution #1 | Possible Execution #2 | Possible Execution #3 |
| --- | --- | --- | --- |
| . | . | . | . |
| . | . | . | . |
| . | . | . | . |
| `x=x+1;` | `x=x+1;` | `x=x+1;` | `x=x+1;` |
| `y=y+x;` | `y=y+x;` | Thread suspended | `y=y+x;` |
| `z=x+5y;` | `z=x+5y;` | Others run | Thread suspended |
| . | . | Thread resumed | Others run |
| . | . | `y=y+x;` | Thread resumed |
| . | . | `z=x+5y;` | `z=x+5y;` |

# Possible Executions




One Execution

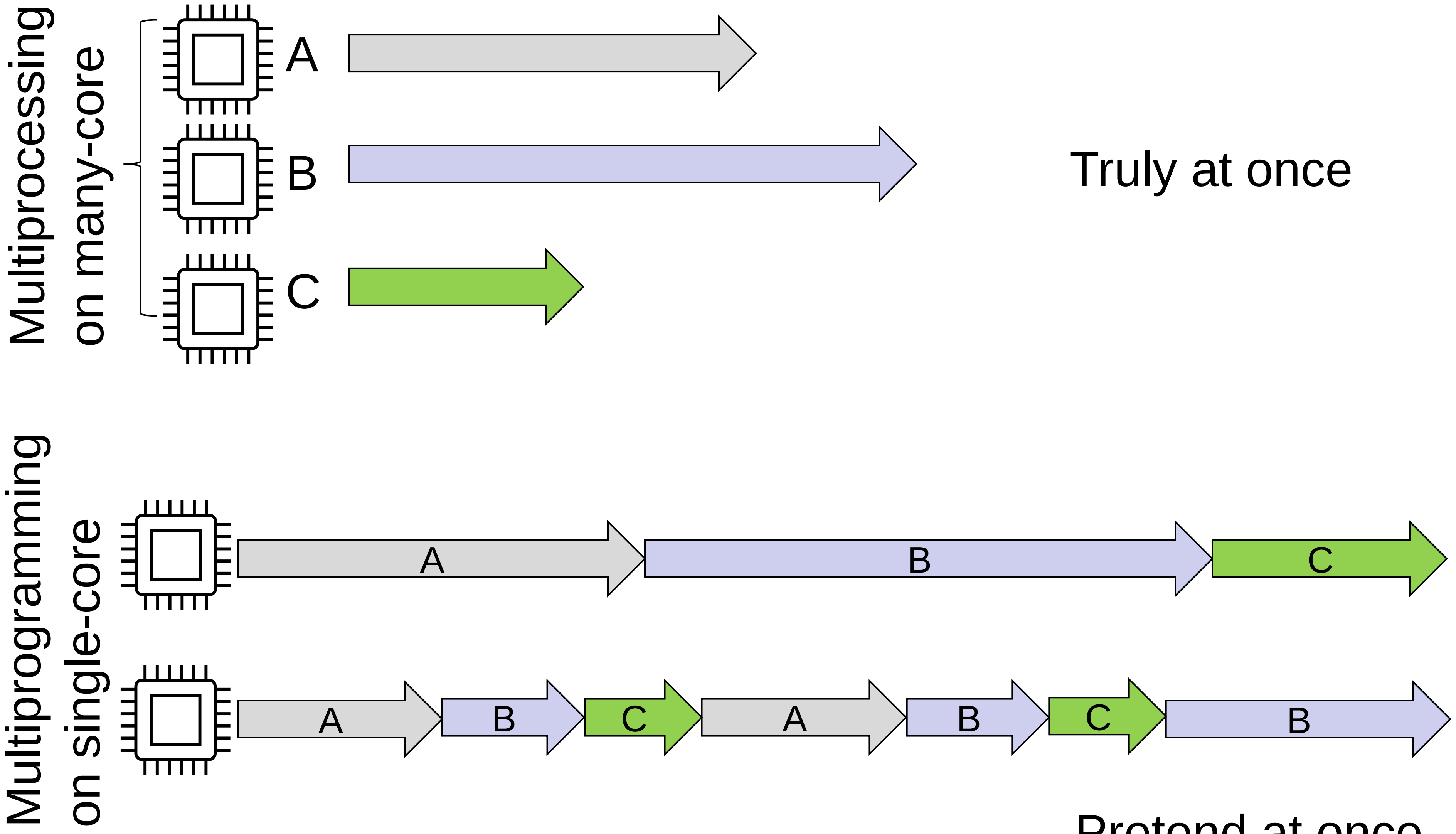

Another Execution




A Third Execution

https://www.youtube.com/watch?v=YVrrw83U--I

# Multiprocessing vs Multiprogramming




Pretend at once

# Why use Pretend At Once?

- Most programs do nothing most of the time
- Some tasks involve lots of waiting
- Users want to have multiple things running “at once”
- Brain smooths over small time discontinuities

# Correctness for systems with concurrent threads

Dispatcher can schedule threads in any way → programs must work under all circumstances

- Can you test this?
- How can you know if your program works?

Ideal: 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!)

Image credit: Steve Jurvetson (flickr)

# Correctness for systems with concurrent threads

- Reality: Cooperating Threads
  - Shared State between multiple threads
  - Non-deterministic
  - Non-reproducible

Image credit: Youtube

- Non-deterministic and Non-reproducible means bugs can be intermittent
  - "Heisenbugs"

# Why allow cooperating threads?

- 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)

# Why allow cooperating threads?

- 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

# Interactions Complicate Debugging

- No programs are truly independent
  - Processes share file system, OS resources, network, etc.
  - Example: buggy device driver causes thread A to crash “independent thread” B
- You 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-determinism makes things impossible

Example: Memory layout of kernel and user programs

- Depends on scheduling, which depends on timer/other things
- Original UNIX had a bunch of non-deterministic errors

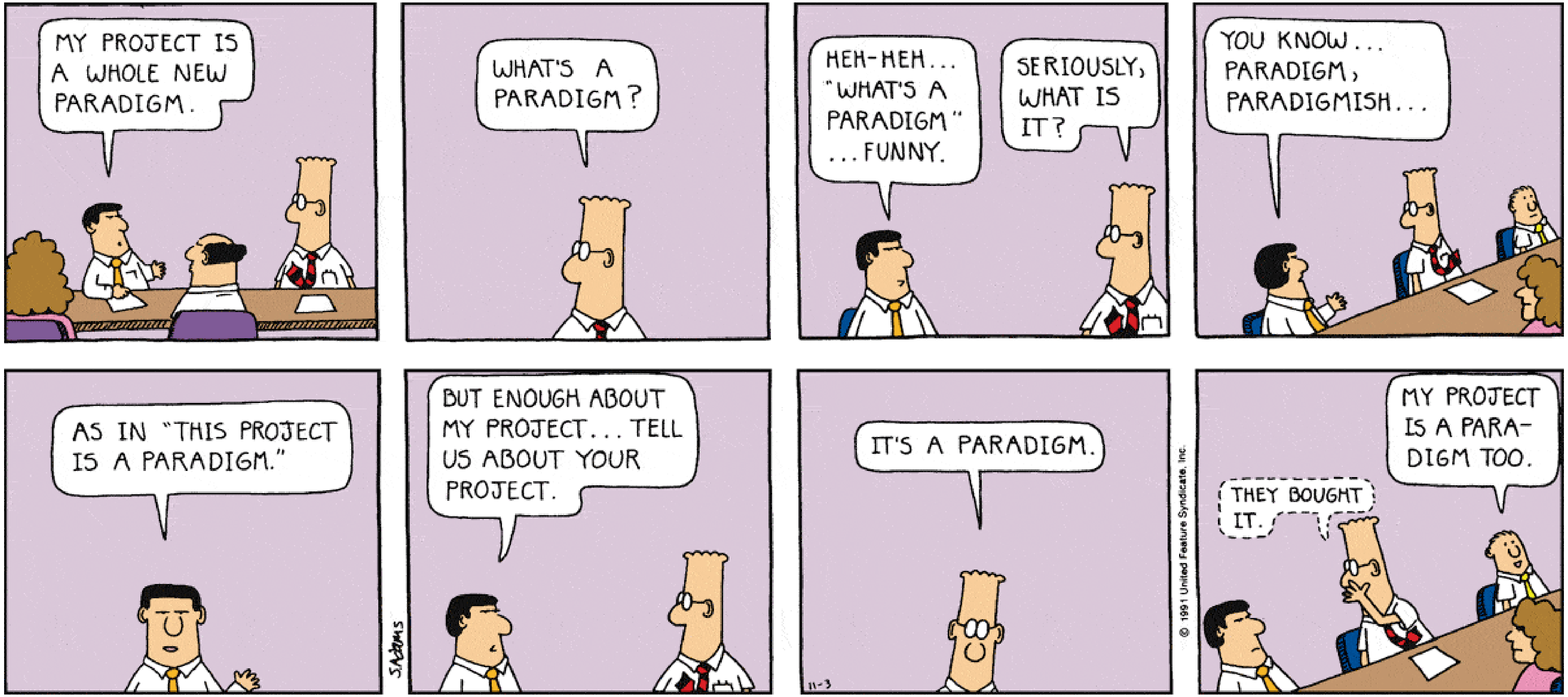
Example: Something which does interesting I/O

- User typing of letters used to help generate secure keys
- Can’t predict → Can’t test → Can never be certain


THE ERRATIC FEEDBACK FROM A RANDOMLY-VARYING WIRELESS SIGNAL CAN MAKE YOU CRAZY.

Image source: XKCD (http://imgs.xkcd.com/comics/feedback.png)

# Goal: Paradigms!

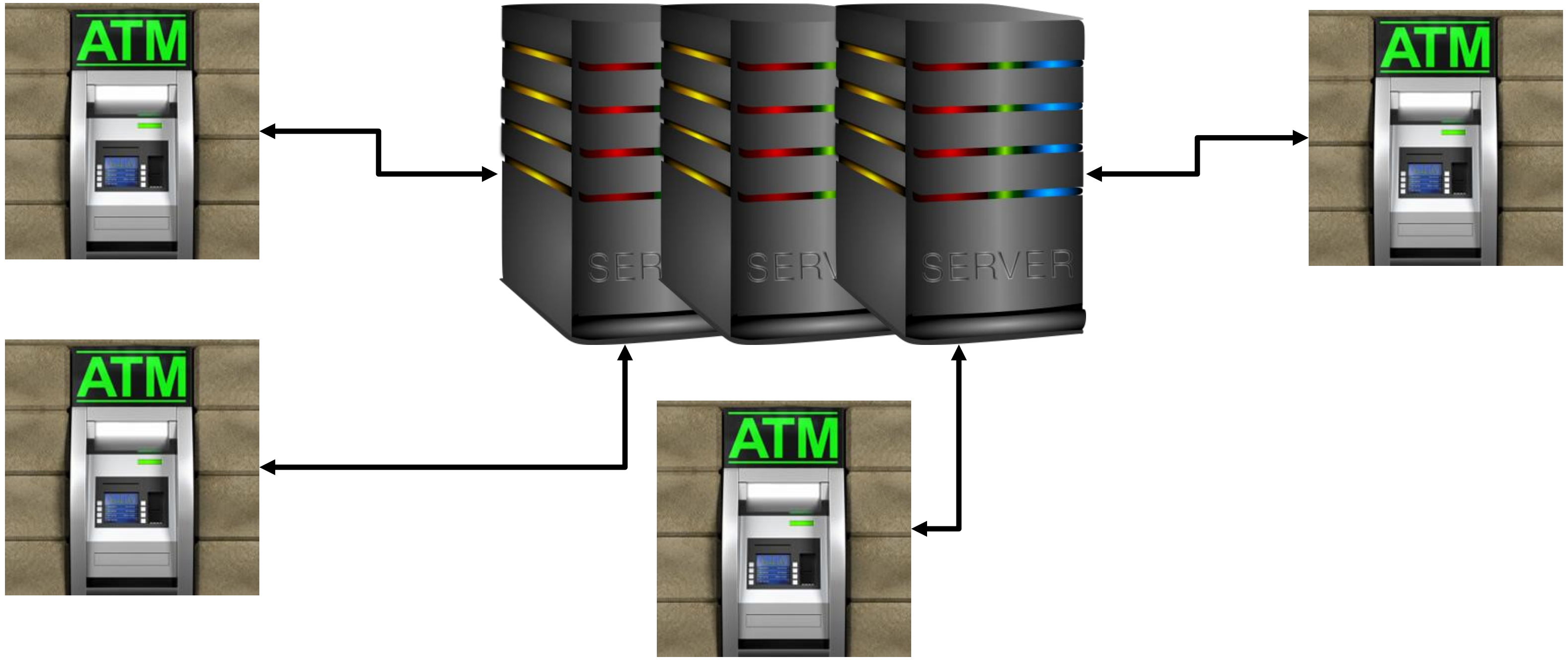



Source: Dilbert (3 Nov 1991) by Scott Adams

# So Far

- Concurrency: Processes
- Threads
  - Kernel Supported Threads vs. User Supported Threads
  - Switching between threads
- Concurrency: Putting it together
- Cooperating threads
- Concurrency challenge

# Example: ATM Bank Server




- ATM server problem:
  - Service a set of requests
  - Don’t corrupt database
  - Don’t hand out too much money

# Basic Bank Server Code

```
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 */
}
```

# What can go wrong?

- 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?
- Or, what are the possible values of x below?

| Thread A | Thread B |
|---|---|
| `x = 1;` | `x = 2;` |

  – X could be 1 or 2 (non-deterministic!)
- Could even be 3 for serial processors:
  – Thread A writes 0001, B writes 0010.
  – Scheduling order ABABABBA yields 3!

# Conclusion

- Concurrency: Processes
- Threads
  - Kernel Supported Threads vs. User Supported Threads
  - Switching between threads
- Concurrency: Putting it together
- Cooperating threads
- Concurrency challenge