Intro to Threads
- after tying up loose ends -

David E. Culler

CS162 – Operating Systems and Systems Programming

Lecture 6

Sept 12, 2014

Reading: A&D Ch4.1-5
HW 1 due Mon 9/15
Proj 1 out next week

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

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

First some Loose ends

Namespaces for communication

- Hostname
  – www.eecs.berkeley.edu
- IP address
  – 128.32.244.172 (ipv6?)
- Port Number
  – 0-1023 are "well known" or "system" ports
    - Superuser privileges to bind to one
  – 1024 – 49151 are "registered" ports (registry)
    - Assigned by IANA for specific services
  – 49152–65535 ($2^{15}+2^{14}$ to $2^{16}-1$) are "dynamic" or "private"
    - Automatically allocated as "ephemeral Ports"

Recall: UNIX Process Management

- UNIX fork – system call to create a copy of the current process, and start it running
  – No arguments!
- UNIX exec – system call to *change the program* being run by the current process
- UNIX wait – system call to wait for a process to finish
- UNIX signal – system call to send a notification to another process

Signals – infloop.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 %d - phew!\n",signum);
  exit(1);
}

int main() {
  signal(SIGINT, signal_callback_handler);

  while (1) {}
}
```

Got top?

Process races: fork.c

```
if (cpid > 0) {
   mypid = getpid();
   printf("[%d] parent of [%d]\n", mypid, cpid);
   for (i=0; i<100; i++) {
     printf("[%d] parent: %d\n", mypid, i);
     //       sleep(1);
   }
 }  else if (cpid == 0) {
   mypid = getpid();
   printf("[%d] child\n", mypid);
   for (i=0; i>-100; i--) {
     printf("[%d] child: %d\n", mypid, i);
     //       sleep(1);
   }
 }
```

UNIX Process Management

- UNIX fork – system call to create a copy of the current process, and start it running
  - No arguments!
- UNIX exec – system call to *change the program* being run by the current process
- UNIX wait – system call to wait for a process to finish
- UNIX signal – system call to send a notification to another process

Signals – infloop.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 %d - phew!\n",signum);
  exit(1);
}

int main() {
  signal(SIGINT, signal_callback_handler);

  while (1) {}
}
```

Got top?

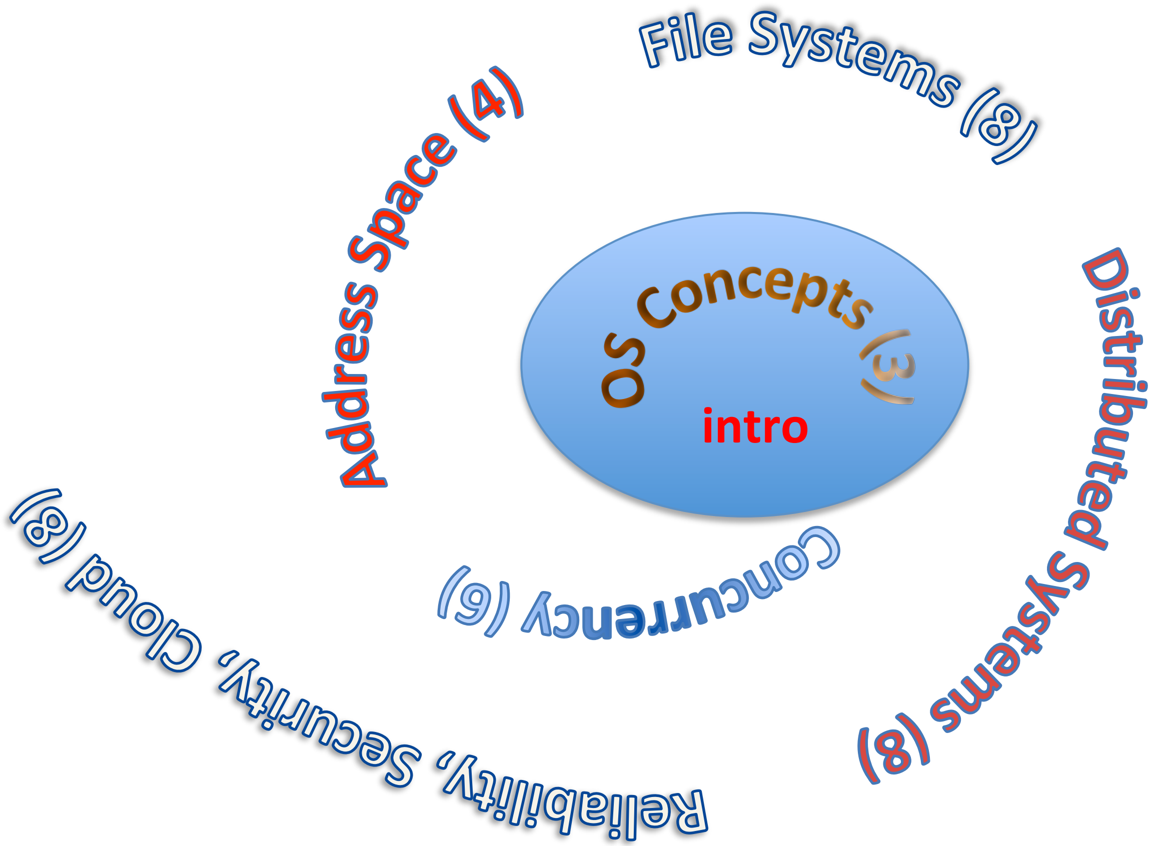
Process races: fork.c

```
if (cpid > 0) {
   mypid = getpid();
   printf("[%d] parent of [%d]\n", mypid, cpid);
   for (i=0; i<100; i++) {
     printf("[%d] parent: %d\n", mypid, i);
     //      sleep(1);
   }
 }  else if (cpid == 0) {
   mypid = getpid();
   printf("[%d] child\n", mypid);
   for (i=0; i>-100; i--) {
     printf("[%d] child: %d\n", mypid, i);
     //      sleep(1);
   }
 }
```

BIG OS Concepts so far

- Processes
- Address Space
- Protection
- Dual Mode
- Interrupt handlers (including syscall and trap)
- File System
  - Integrates processes, users, cwd, protection
- Key Layers: OS Lib, Syscall, Subsystem, Driver
  - User handler on OS descriptors
- Process control
  - fork, wait, signal, exec
- Communication through sockets
- Client-Server Protocol

Course Structure: Spiral

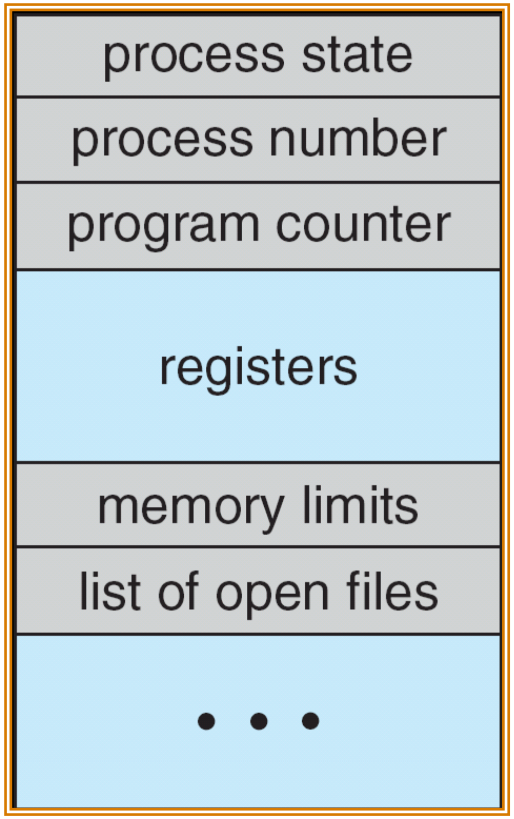


Traditional UNIX Process

- Process: *Operating system abstraction to represent what is needed to run a single program*
  – Often called a “HeavyWeight Process”
- Two parts:
  – Sequential program execution stream
    - Code executed as a *sequential* stream of execution (i.e., thread)
    - Includes State of CPU registers
  – Protected resources:
    - Main memory state (contents of Address Space)
    - I/O state (i.e. file descriptors)

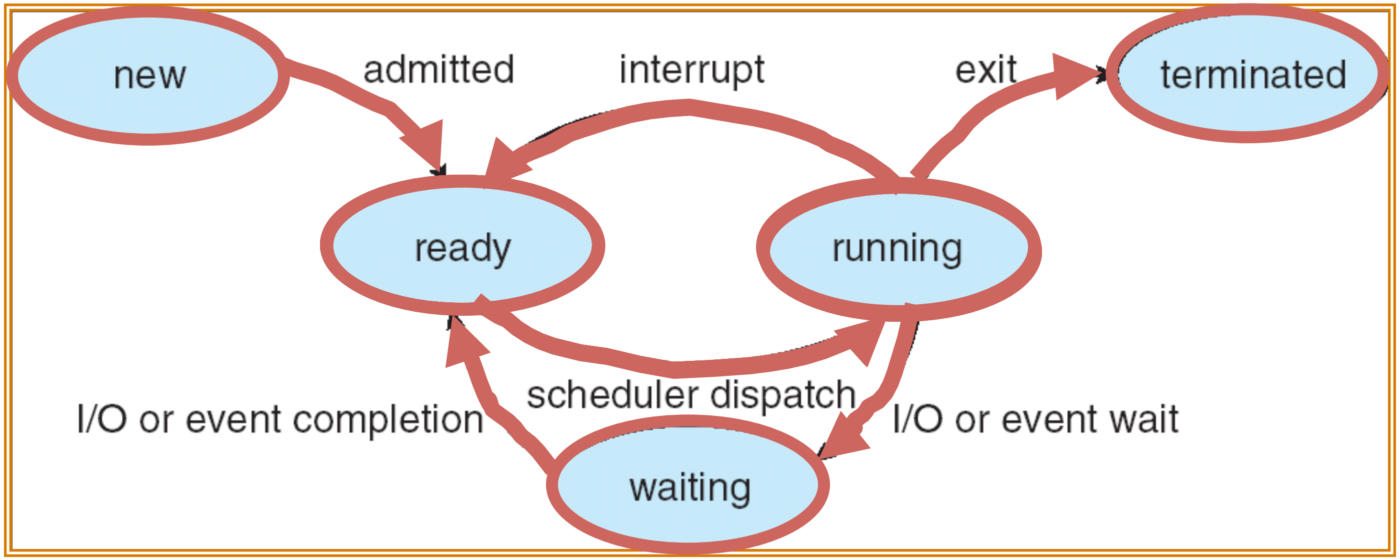
How do we Multiplex Processes?

- 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 Mapping: Give each process their own address space
    - Kernel/User duality: Arbitrary multiplexing of I/O through system calls



Process Control Block

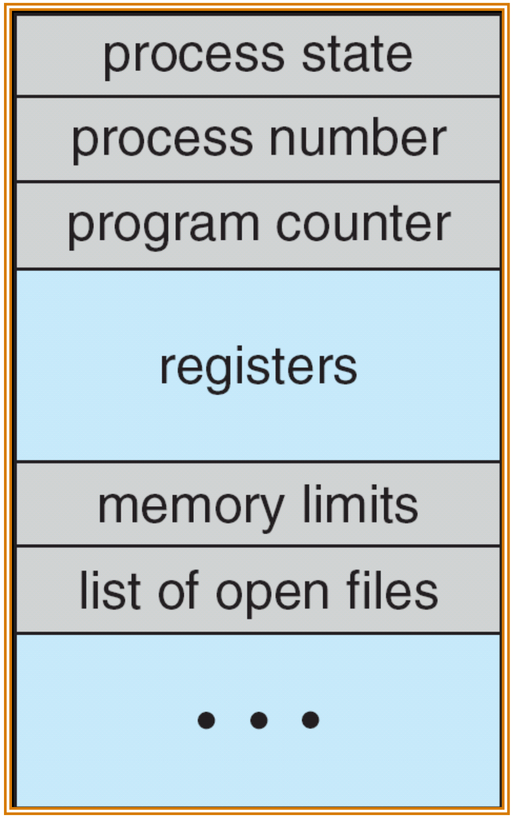
Lifecycle of a Process



- 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 Control Block

- The current state of process held in a process control block (PCB): (for a single-threaded process)

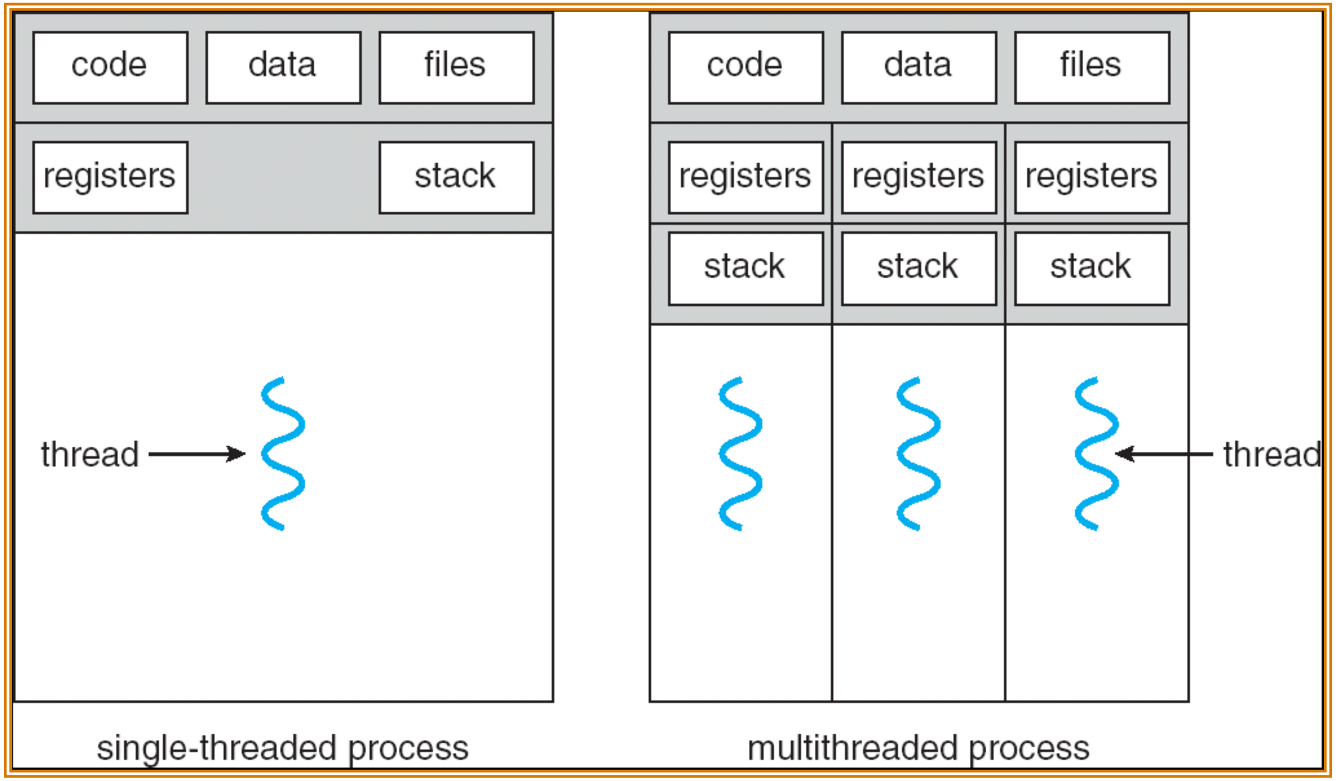


Process Control Block

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*
  - 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

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?

Examples multithreaded programs

- Embedded systems
  - Elevators, Planes, Medical systems, Wristwatches
  - Single Program, concurrent operations
- Most modern OS kernels
  - Internally concurrent because have to deal with concurrent requests by multiple users
  - But no protection needed within kernel
- Database Servers
  - Access to shared data by many concurrent users
  - Also background utility processing must be done

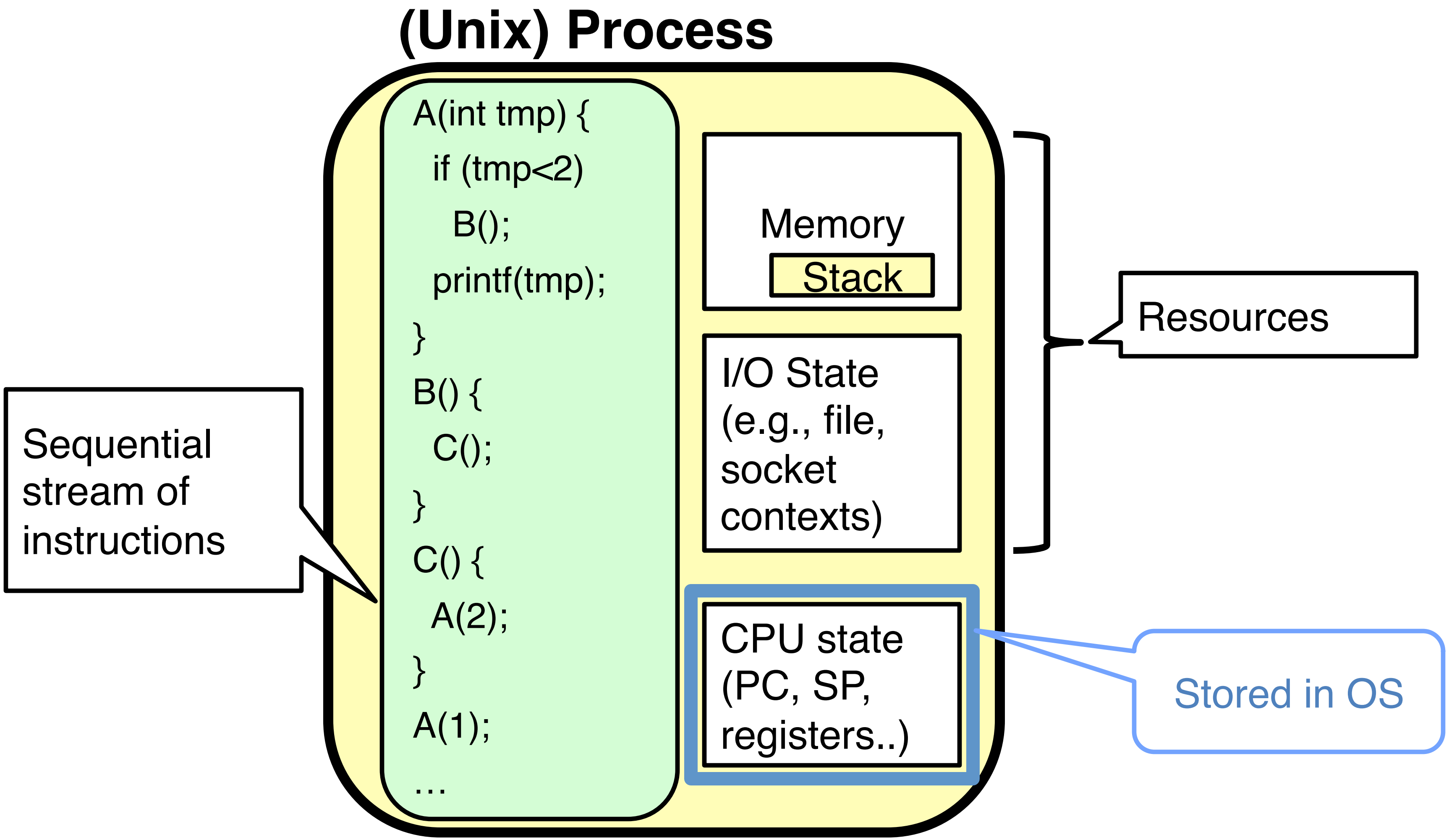
Example multithreaded programs (con't)

- Network Servers
  - Concurrent requests from network
  - Again, single program, multiple concurrent operations
  - File server, Web server, and airline reservation systems

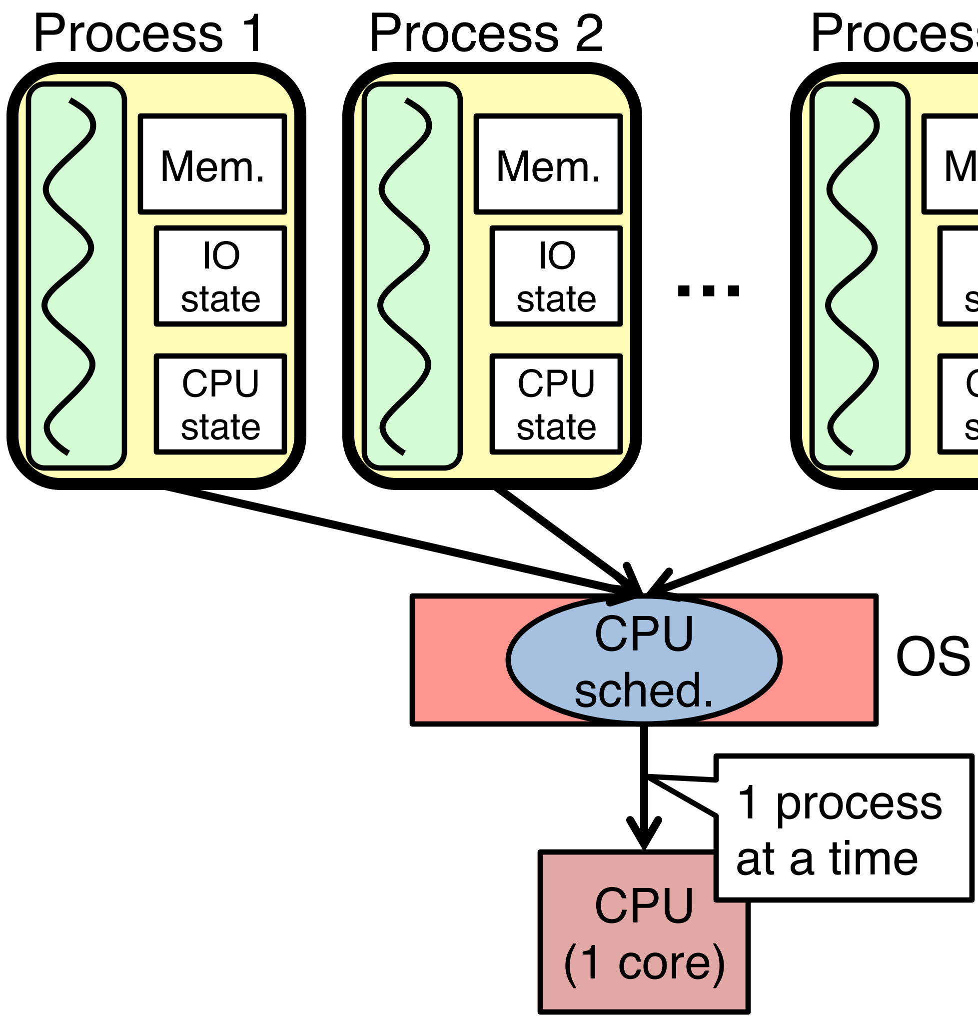
- Parallel Programming (More than one physical CPU)
  - Split program into multiple threads for parallelism
  - This is called Multiprocessing

- Some multiprocessors are actually uniprogrammed:
  - Multiple threads in one address space but one program at a time

Putting it together: Process

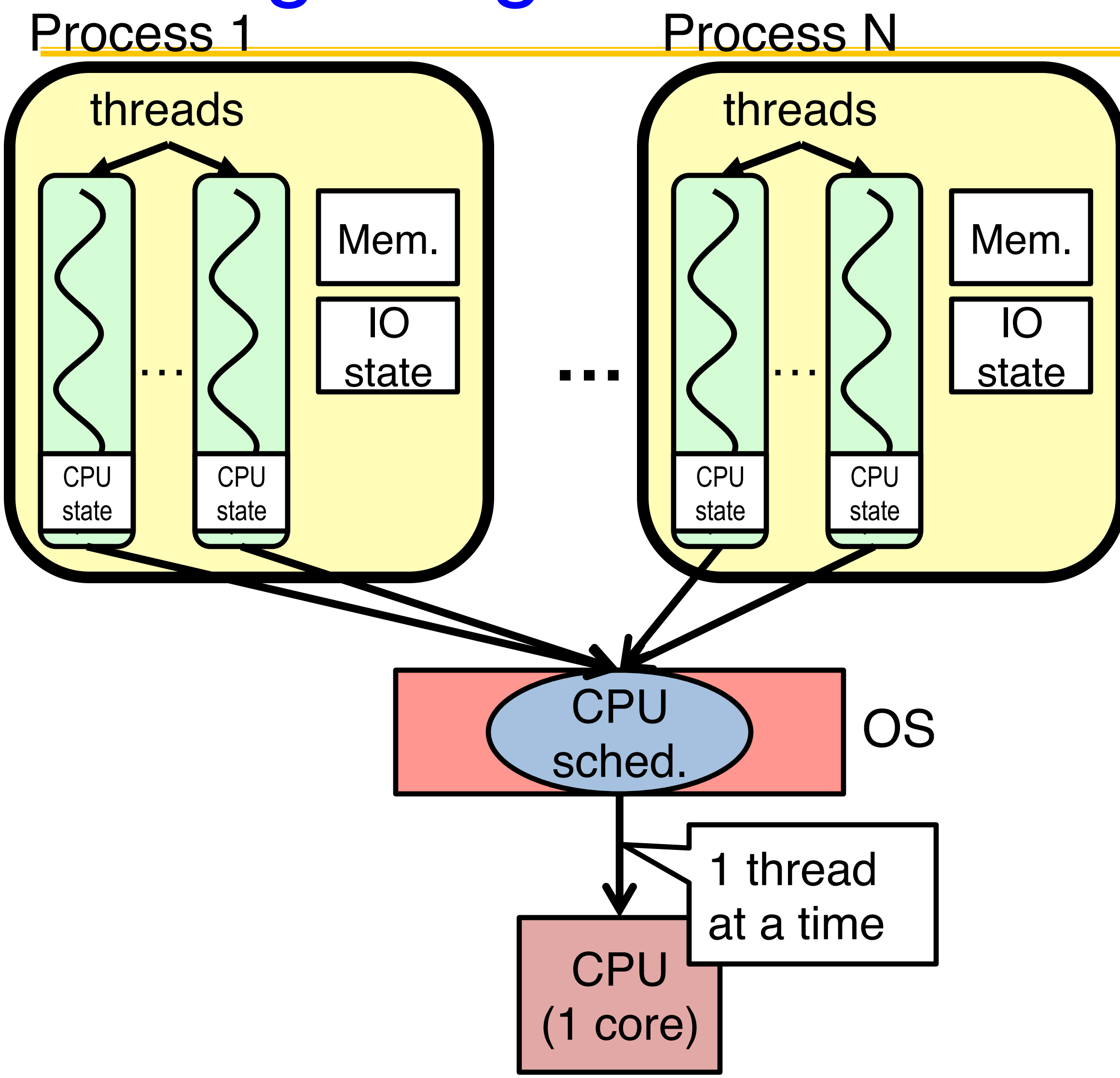


Putting it together: Processes



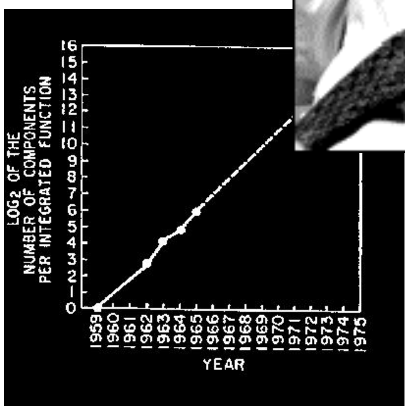
- Switch overhead: high
  - CPU state: low
  - Memory/IO state: high
- Process creation: high
- Protection
  - CPU: yes
  - Memory/IO: yes
- Sharing overhead: high (involves at least a context switch)

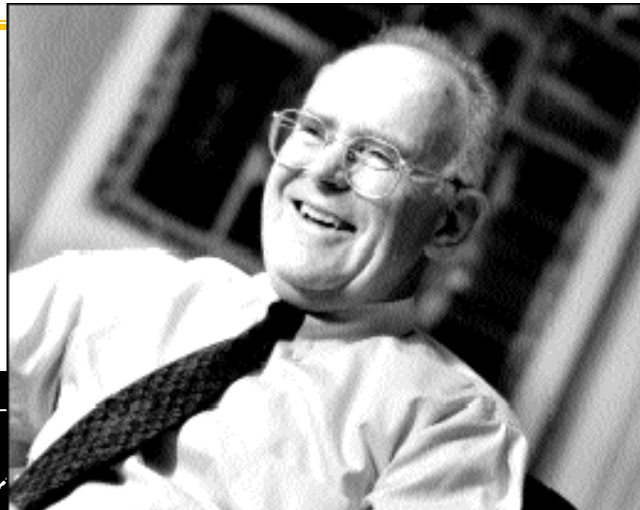
Putting it together: Threads

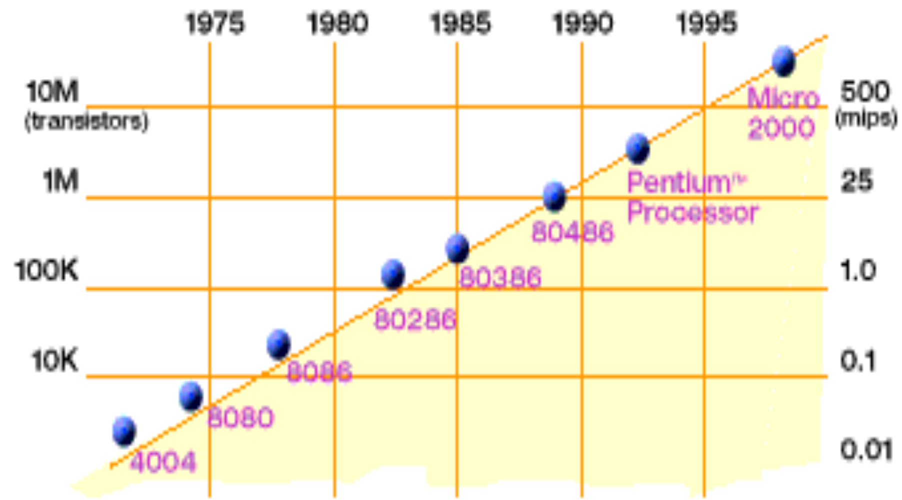


- Switch overhead: low (only CPU state)
- Thread creation: low
- Protection
  - CPU: yes
  - Memory/IO: No
- Sharing overhead: low (thread switch overhead low)

Technology Trends: Moore's Law







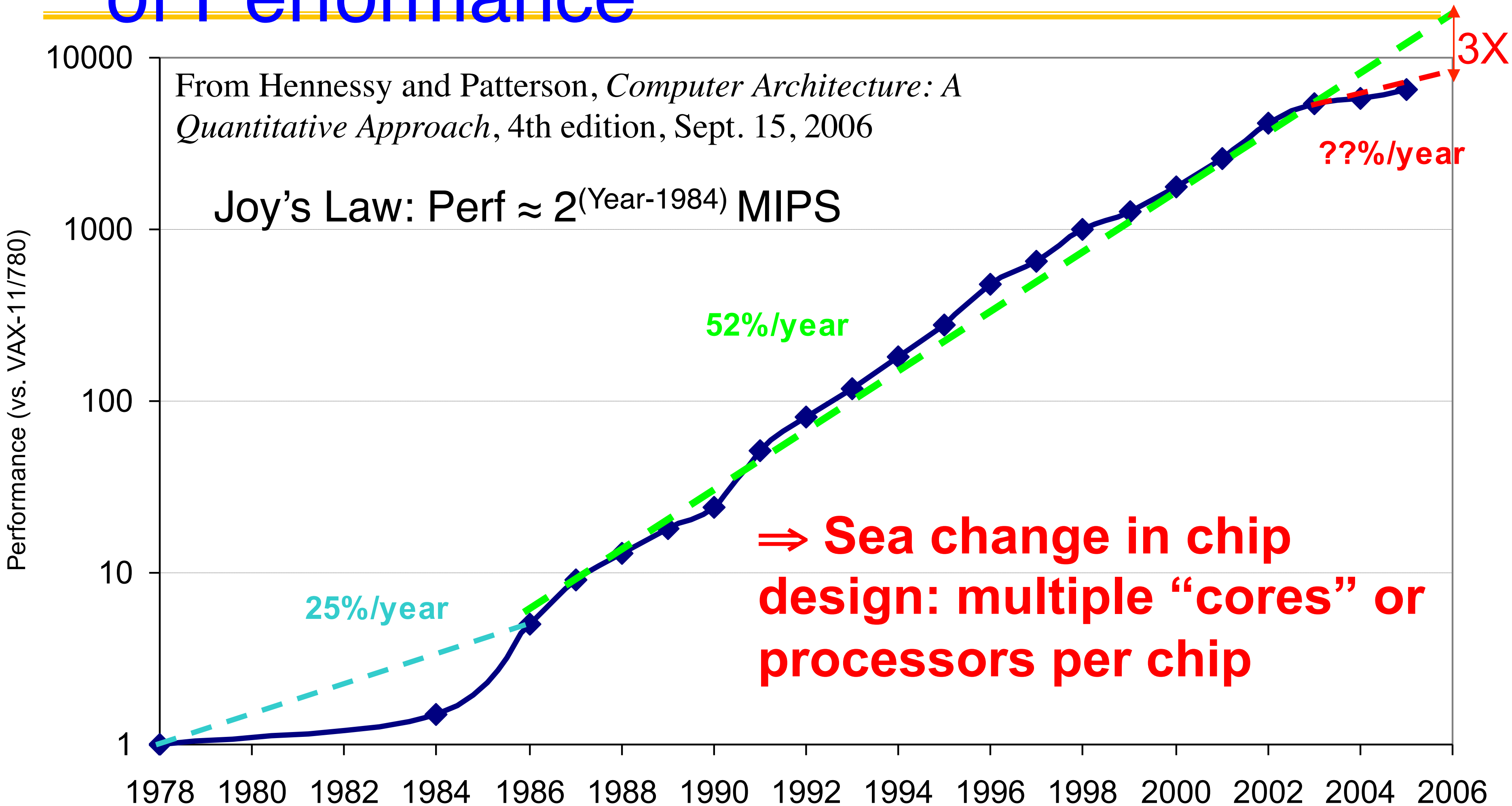
2X transistors/Chip Every 1.5 years

Called "Moore's Law"

Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

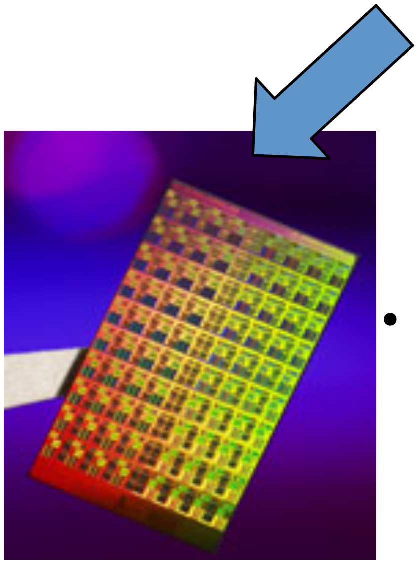
Microprocessors have become smaller, denser, and more powerful.

New Challenge: Slowdown in Joy's law of Performance

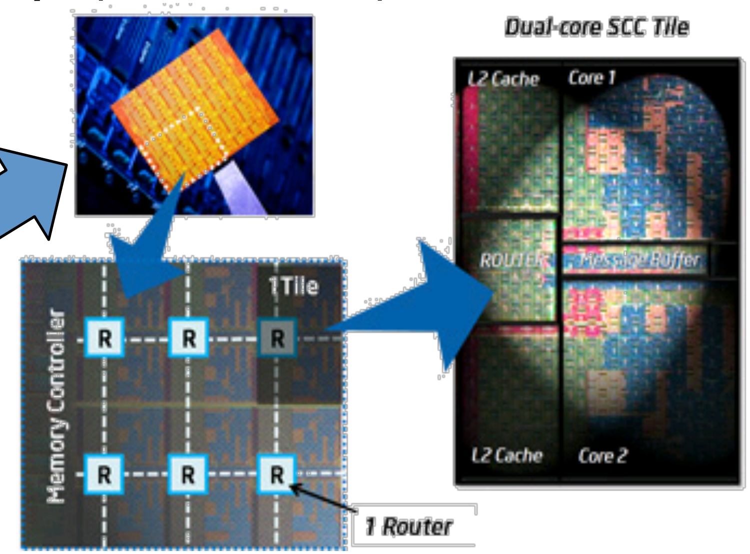


- VAX : 25%/year 1978 to 1986
- RISC + x86: 52%/year 1986 to 2002
- RISC + x86: ??%/year 2002 to present

ManyCore Chips: The future is here

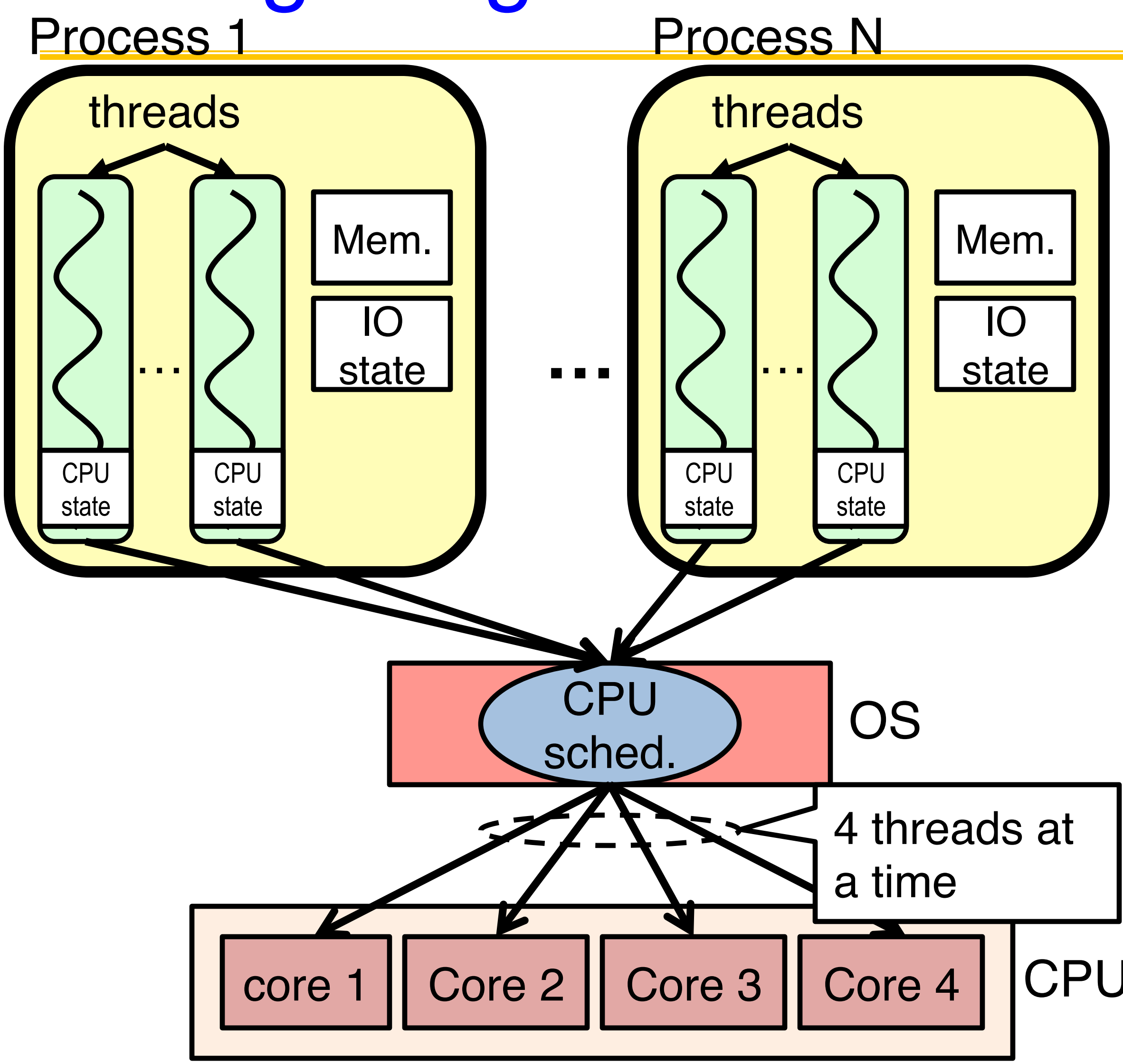


- Intel 80-core multicore chip (Feb 2007)
  - 80 simple cores
  - Two FP-engines / core
  - Mesh-like network
  - 100 million transistors



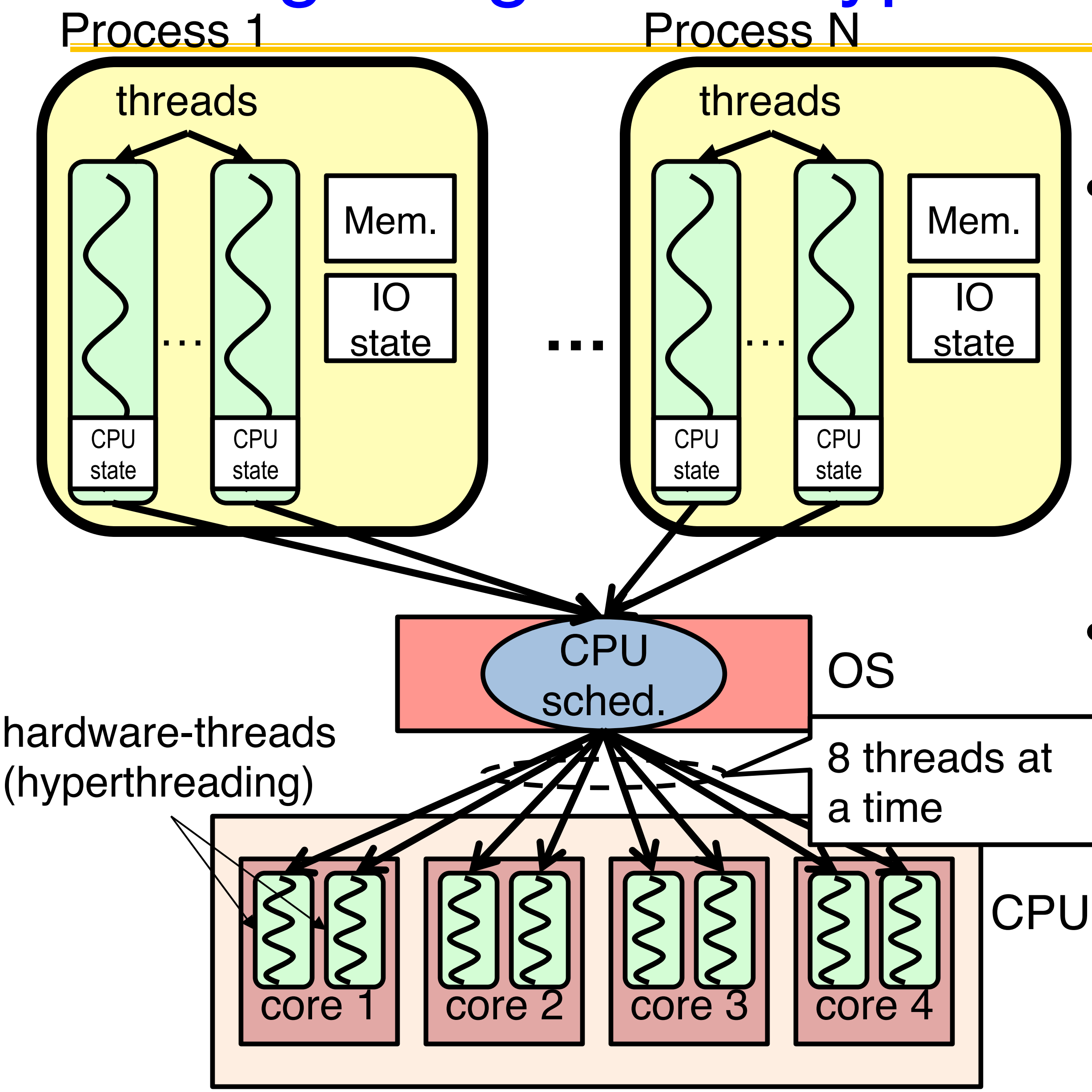
- Intel Single-Chip Cloud Computer (August 2010)
  - 24 "tiles" with two cores/tile
  - 24-router mesh network
  - 4 DDR3 memory controllers
  - Hardware support for message-passing
- "ManyCore" refers to many processors/chip
  - 64? 128? Hard to say exact boundary
- How to program these?
  - Use 2 CPUs for video/audio
  - Use 1 for word processor, 1 for browser
  - 76 for virus checking???
- Parallelism must be exploited at all levels

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)

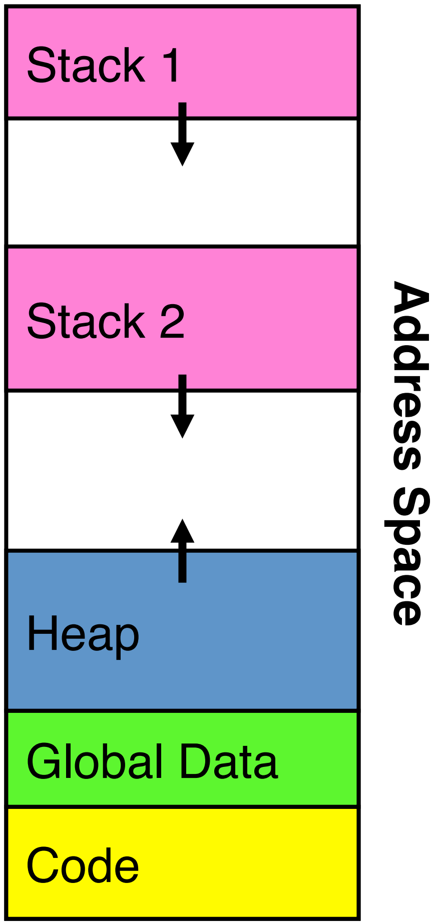
Putting it together: Hyper-Threading



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

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?



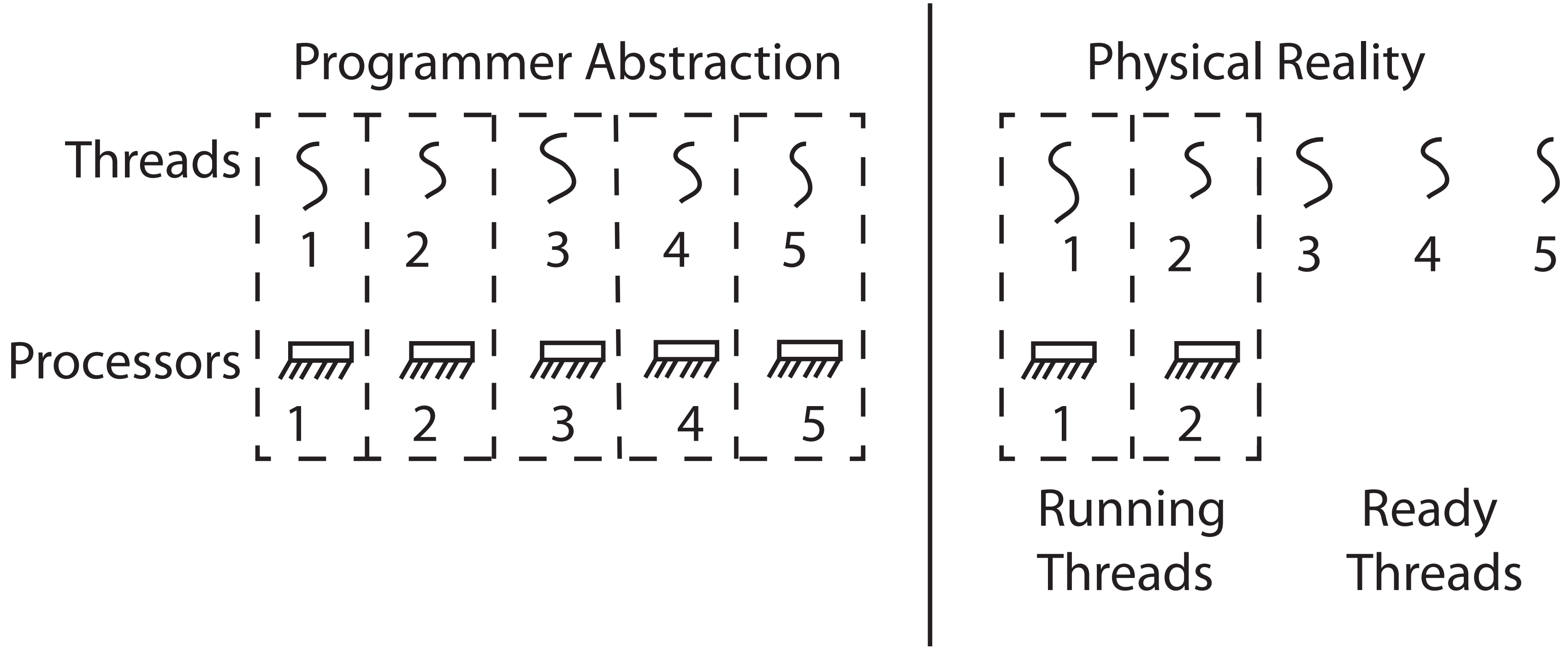
Thread Operations

- 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
- thread_exit
  - Quit thread and clean up, wake up joiner if any
  - Pintos: thread_exit

http://cs162.eecs.berkeley.edu/static/lectures/code06/pthread.c

Thread Abstraction

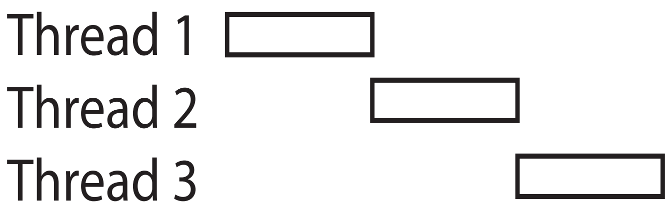
- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule



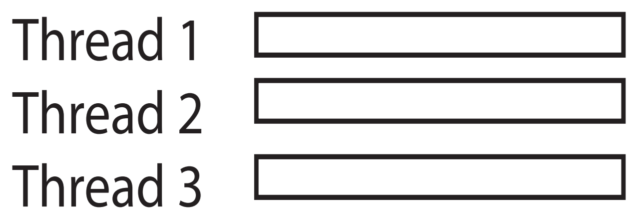
Programmer vs. Processor View

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
$x = x + 1;$	$x = x + 1;$	$x = x + 1$	$x = x + 1$
$y = y + x;$	$y = y + x;$	....	$y = y + x$
$z = x + 5y;$	$z = x + 5y;$	thread is suspended	....
.	.	other thread(s) run	thread is suspended
.	.	thread is resumed	other thread(s) run
.	.	....	thread is resumed
		$y = y + x$	....
		$z = x + 5y$	$z = x + 5y$

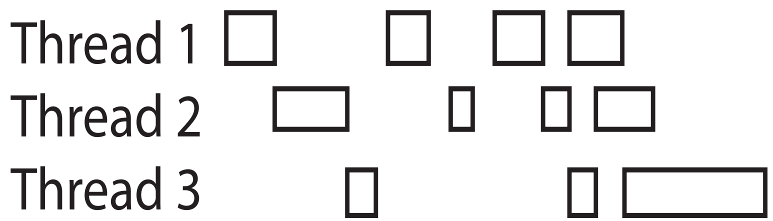
Possible Executions



a) One execution



b) Another execution



c) Another execution

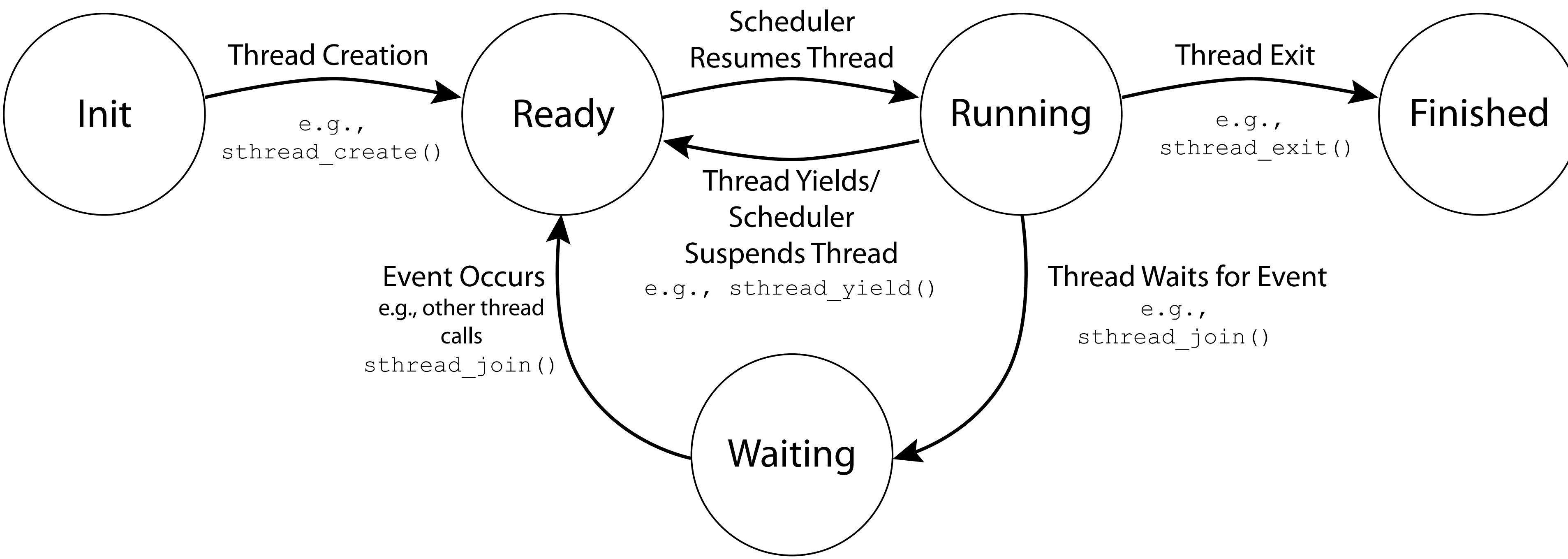
Thread State

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

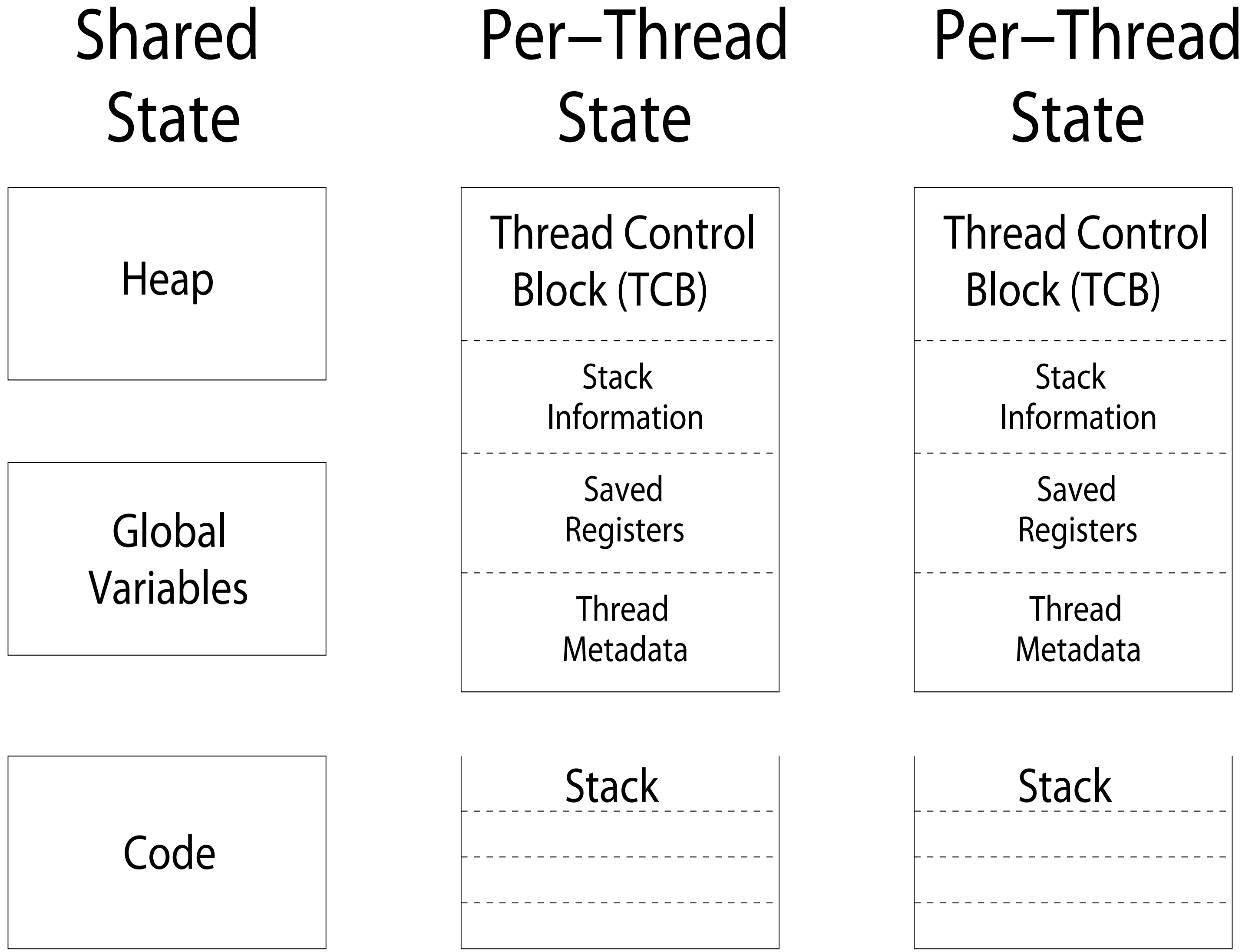
- State "private" to each thread
  - Kept in TCB ≡ Thread Control Block
  - CPU registers (including, program counter)
  - Execution stack – what is this?

- Execution Stack
  - Parameters, temporary variables
  - Return PCs are kept while called procedures are executing

Thread Lifecycle



Shared vs. Per-Thread State



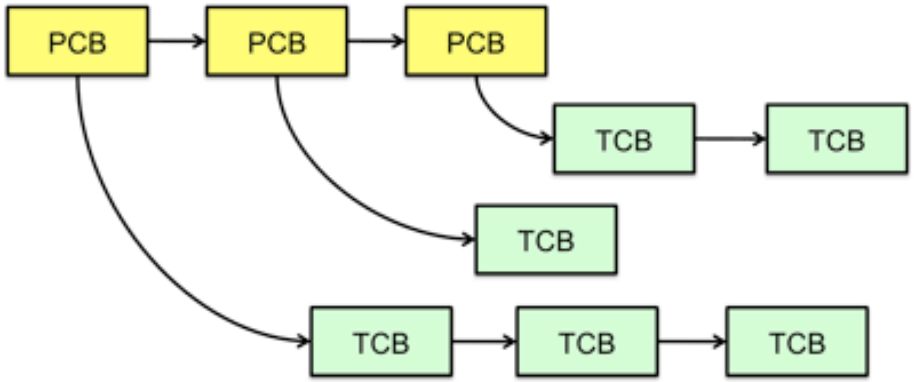
Per Thread State

- 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 …

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.

The Numbers

Context switch in Linux: 3-4 μsecs (Current Intel i7 & E5).

•Thread switching faster than process switching (100 ns).

•But switching across cores about 2x more expensive than within-core switching.

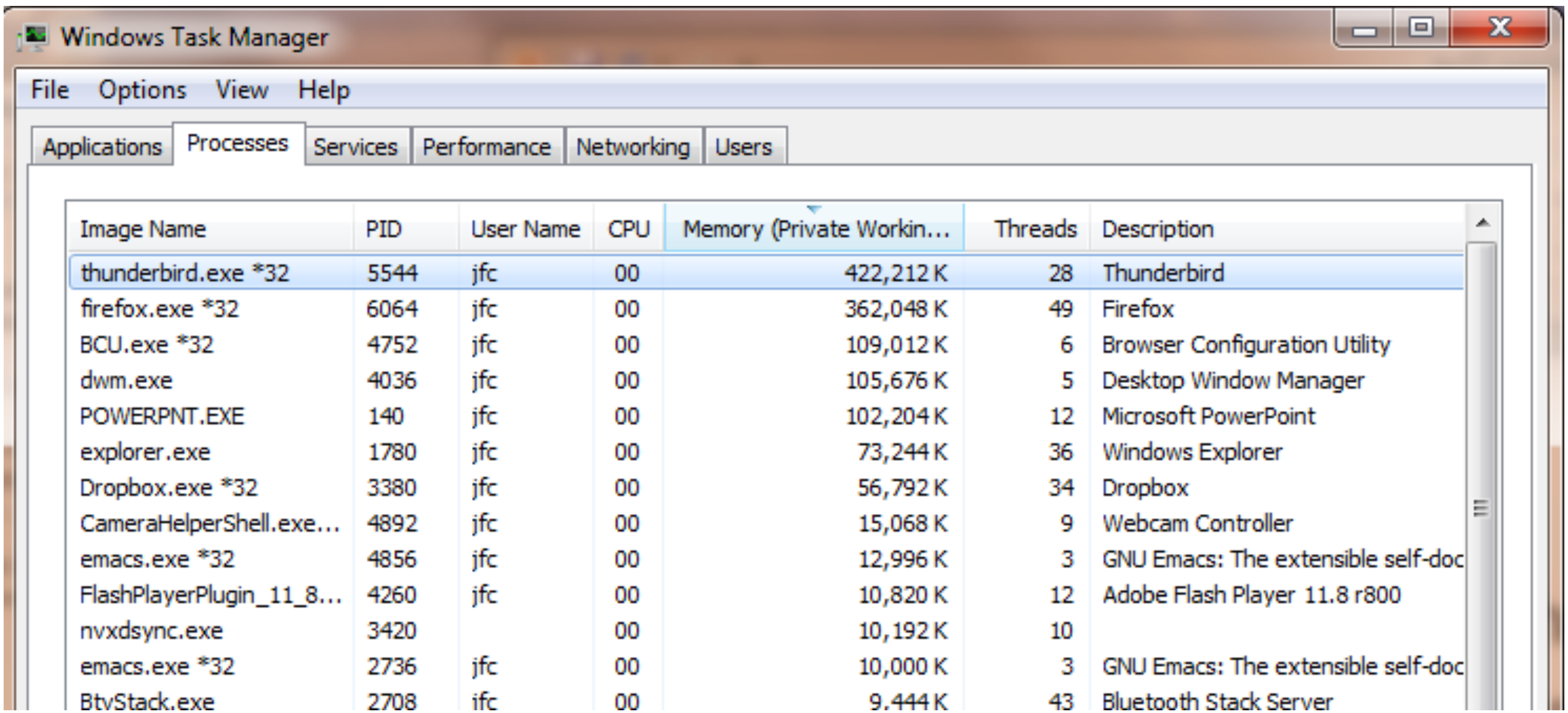
•Context switch time increases sharply with the size of the working set*, and can increase 100x or more.

* 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.

The Numbers

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



Windows Task Manager

File Options View Help

Applications | Processes | Services | Performance | Networking | Users

Image Name	PID	User Name	CPU	Memory (Private Workin...	Threads	Description
thunderbird.exe *32	5544	jfc	00	422,212 K	28	Thunderbird
firefox.exe *32	6064	jfc	00	362,048 K	49	Firefox
BCU.exe *32	4752	jfc	00	109,012 K	6	Browser Configuration Utility
dwm.exe	4036	jfc	00	105,676 K	5	Desktop Window Manager
POWERPNT.EXE	140	jfc	00	102,204 K	12	Microsoft PowerPoint
explorer.exe	1780	jfc	00	73,244 K	36	Windows Explorer
Dropbox.exe *32	3380	jfc	00	56,792 K	34	Dropbox
CameraHelperShell.exe...	4892	jfc	00	15,068 K	9	Webcam Controller
emacs.exe *32	4856	jfc	00	12,996 K	3	GNU Emacs: The extensible self-doc
FlashPlayerPlugin_11_8...	4260	jfc	00	10,820 K	12	Adobe Flash Player 11.8 r800
nvxdsync.exe	3420		00	10,192 K	10	
emacs.exe *32	2736	jfc	00	10,000 K	3	GNU Emacs: The extensible self-doc
BtvStack.exe	2708	jfc	00	9,444 K	43	Bluetooth Stack Server

Threads in a Process

- Threads are useful at user-level
  - Parallelism, hide I/O latency, interactivity
- 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 (Linux, MacOS, Windows): use kernel threads
  - System calls for thread fork, join, exit (and lock, unlock,…)
  - Kernel does context switching
  - Simple, but a lot of transitions between user and kernel mode
- Option C (Windows): scheduler activations
  - Kernel allocates processors to user-level library
  - Thread library implements context switch
  - System call I/O that blocks triggers upcall
- Option D: Asynchronous I/O

Classification

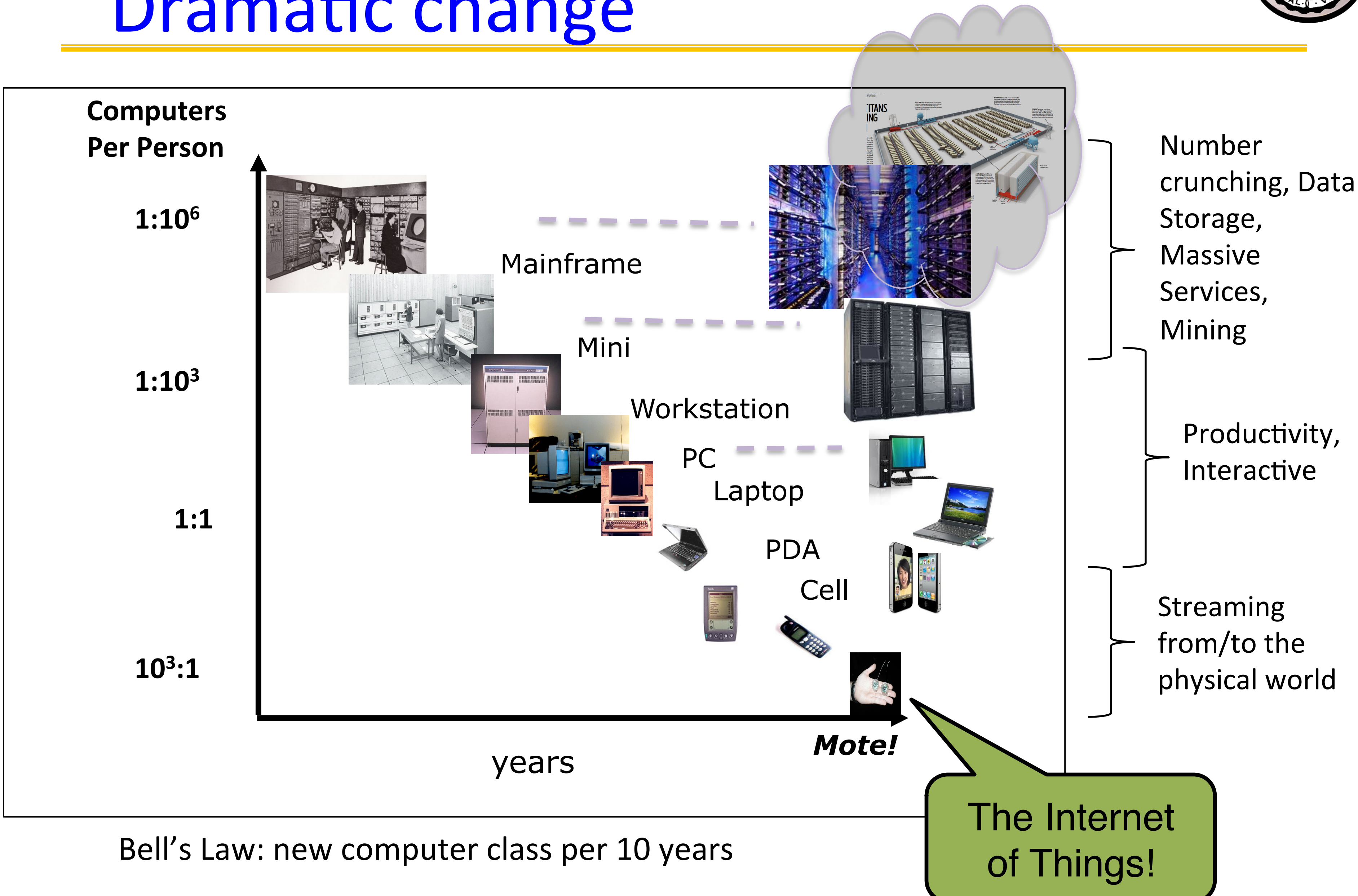
# threads Per AS: / # of addr spaces:	One	Many
One	MS/DOS, early Macintosh	Traditional UNIX
Many	Embedded systems (Geoworks, VxWorks, JavaOS,etc) JavaOS, Pilot(PC)	Mach, OS/2, HP-UX, Win NT to 8, Solaris, OS X, Android, iOS

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

OS Archaeology

- Because of the cost of developing an OS from scratch, most modern OSes have a long lineage:
- Multics → AT&T Unix → BSD Unix → Ultrix, SunOS, NetBSD,…
- Mach (micro-kernel) + BSD → NextStep → XNU → Apple OSX, iphone iOS
- Linux → Android OS
- CP/M → QDOS → MS-DOS → Windows 3.1 → NT → 95 → 98 → 2000 → XP → Vista → 7 → 8 → phone → …
- Linux → RedHat, Ubuntu, Fedora, Debian, Suse,…

Dramatic change



Bell's Law: new computer class per 10 years

Migration of OS Concepts and Features