Concurrency
Welcome to cs378

Chris Rossbach

Outline for Today

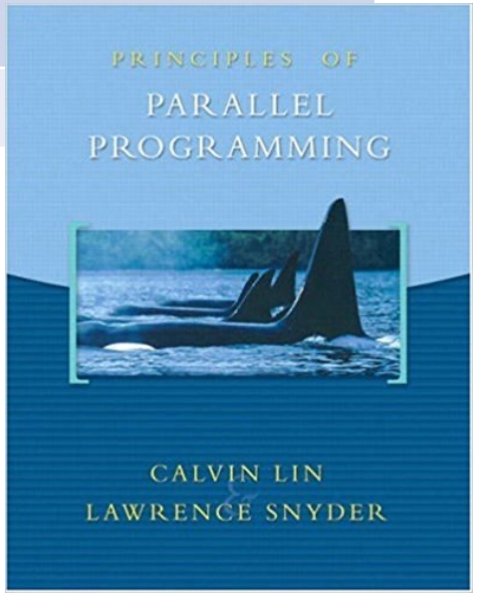
- Questions?
- Administrivia
- Course Overview
- Course Details and Logistics
- Concurrency & Parallelism Basics

Acknowledgments: some materials in this lecture borrowed from:

- *Emmett Witchel, who borrowed them from: Kathryn McKinley, Ron Rockhold, Tom Anderson, John Carter, Mike Dahlin, Jim Kurose, Hank Levy, Harrick Vin, Thomas Narten, and Emery Berger*
- *Mark Silberstein, who borrowed them from: Blaise Barney, Kunle Olukoton, Gupta*

Course Details

Course Name:	**CS378 – Concurrency**
Unique Number:	53015
Lectures:	M-W 9:30-11:00AM Zoom and/or UTC 1.144
Class Web Page:	http://www.cs.utexas.edu/users/rossbach/cs378
Instructor:	Chris Rossbach
TA:	Carter Chu
Text:	Principles of Parallel Programming (ISBN-10: 0321487907)



Please read the syllabus!

…More on this shortly…

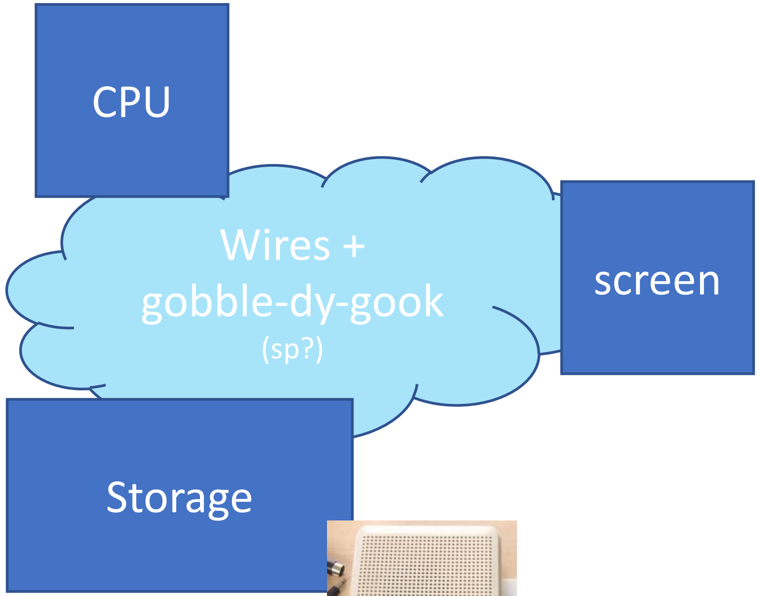
Why you should take this course

- Concurrency is super-cool, and super-important
- You'll learn important concepts and background
- Have *fun* programming cool systems
  - GPUs! (optionally) FGPAs!
  - Modern Programming languages: Go! Rust!
  - Interesting synchronization primitives (not just boring old locks)
  - Programming tools people use to program *super-computers (ooh…)*

Two perspectives:

- The "just eat your kale and quinoa" argument
- The "it's going to be fun" argument

My first computer



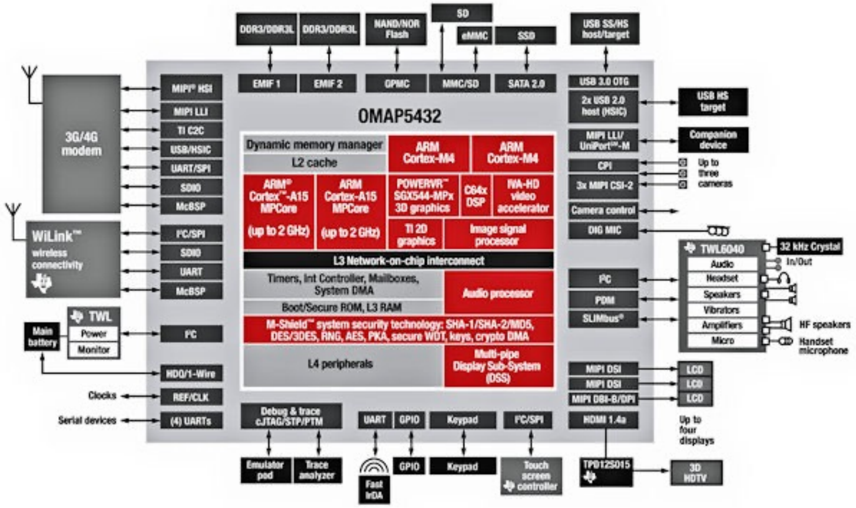
Tape drive!
(also good for playing heavy metal music)

My current computer

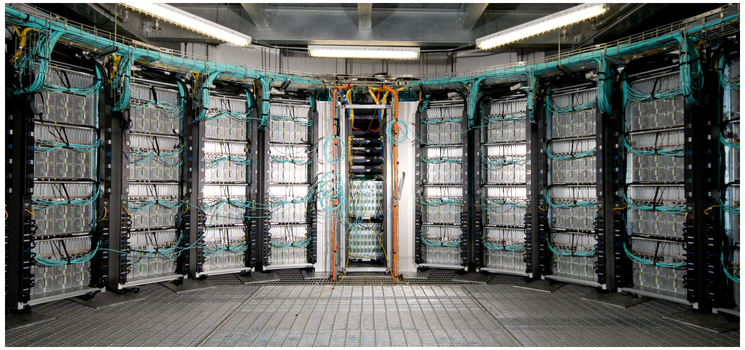
Too boring…

Another of my current computers

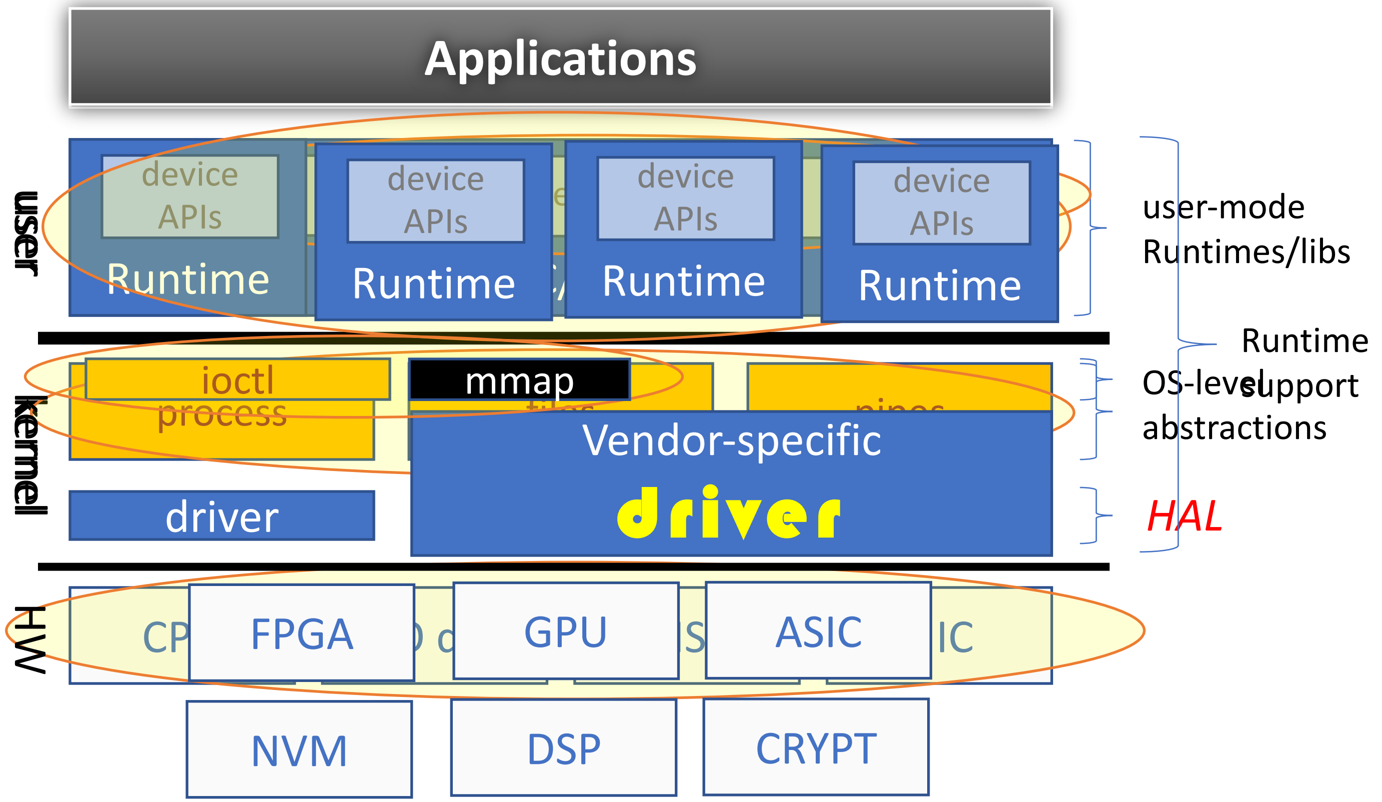
A lot has changed but...
the common theme is...??



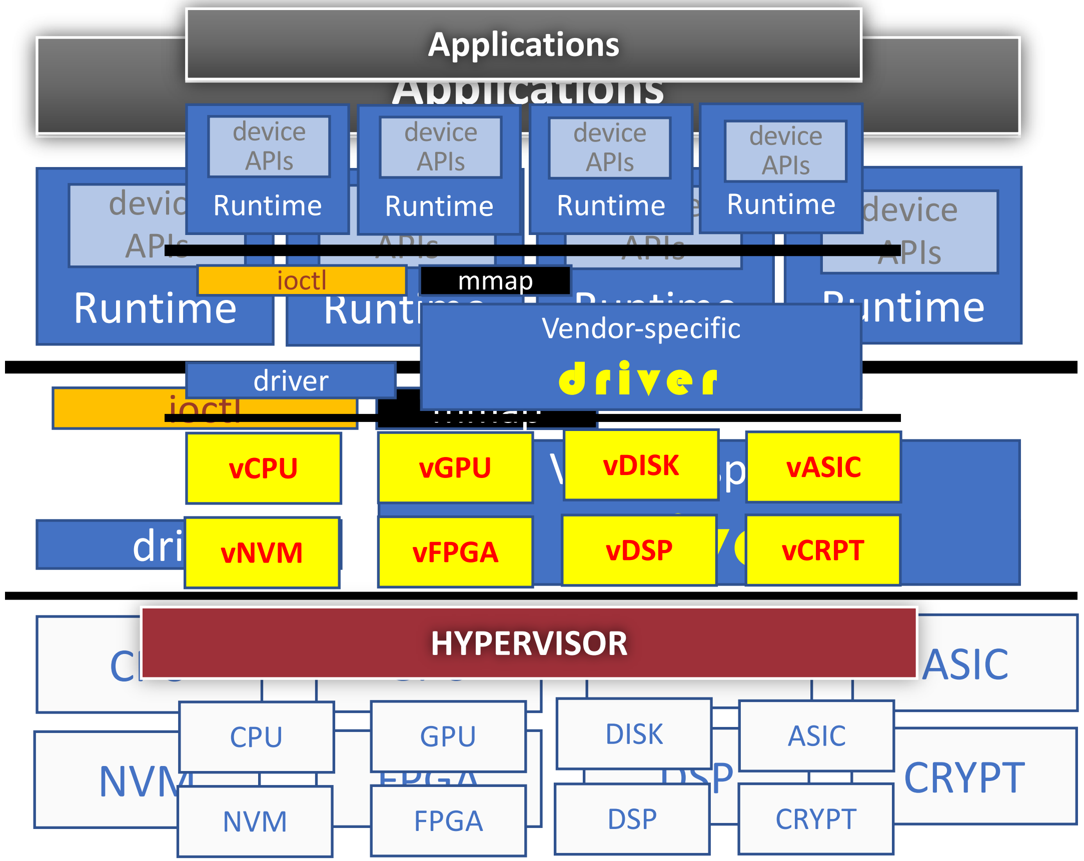
- CPU
- CPU
- GPU
- Image DSP
- Crypto
- ...



Modern Technology Stack



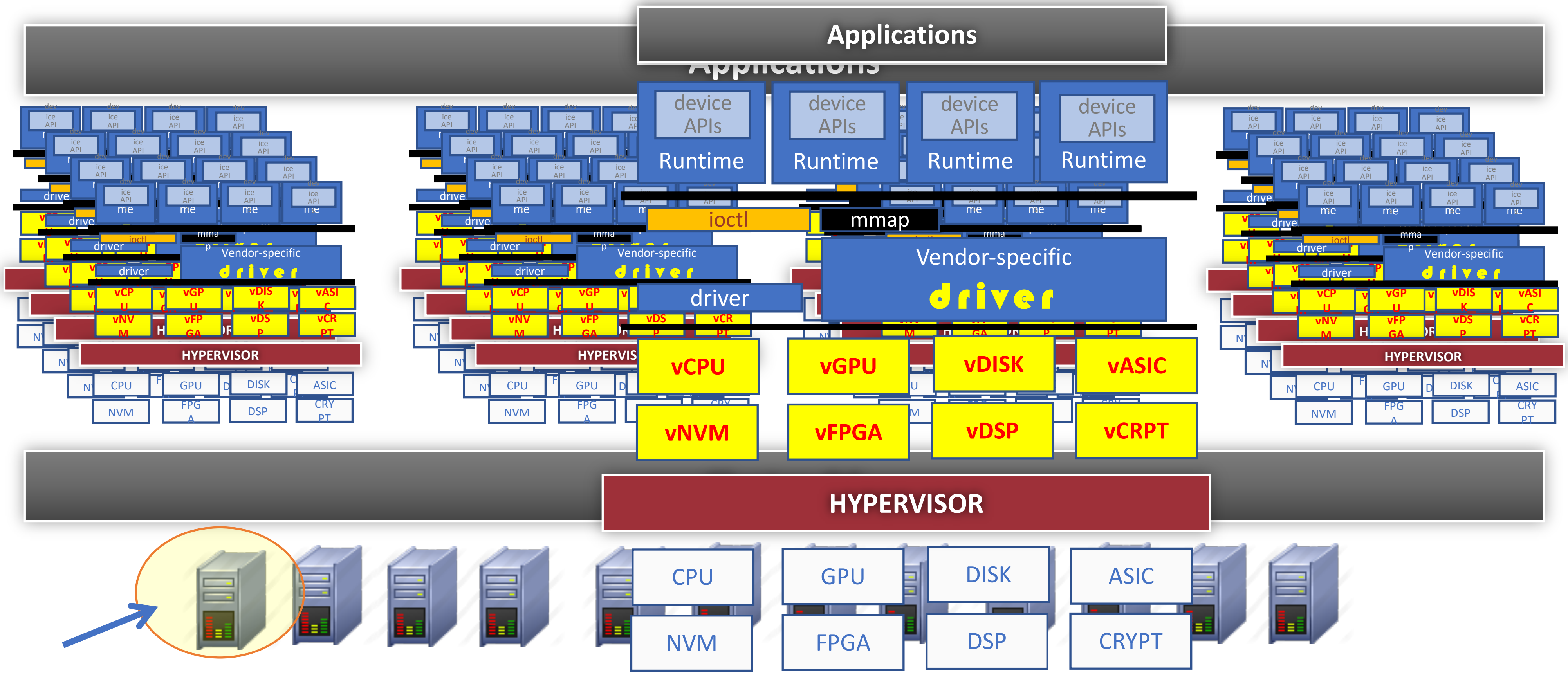
Concurrency and Parallelism are Everywhere



Wait!

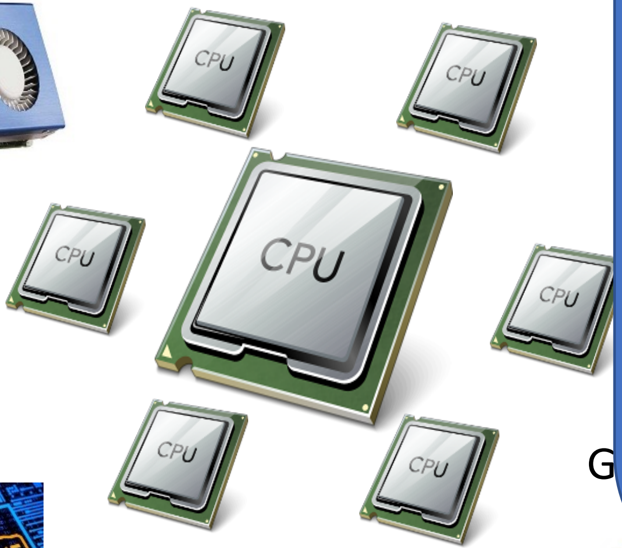
- What’s concurrency?
- What’s parallelism?

Concurrency and Parallelism are Everywhere



Concurrency and Parallelism are everywhere

DSP



Crypto

G

How much parallel and concurrent programming have you learned so far?

- Concurrency/parallelism can't be avoided anymore (want a job?)
- A program or two playing with locks and threads isn't enough
- I've worked in industry a lot—I know

Course goal is to expose you to lots of ways of programming systems like these

…So "you should take this course because it's good for you" (eat your #$(*& kale!)

CPU(s)

GPU

Image DSP

Crypto

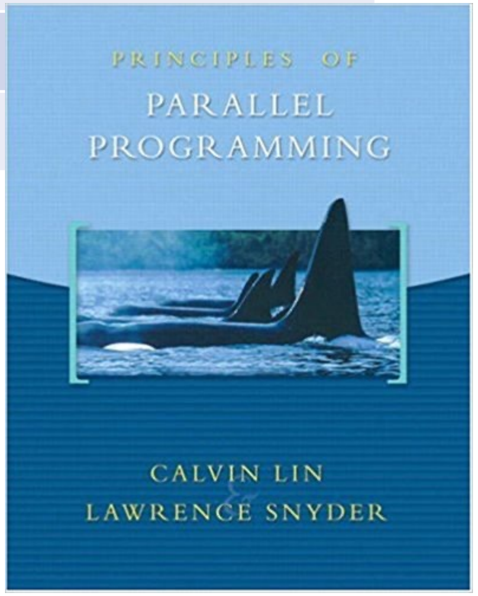
…

Goal: Make Concurrency Your Close Friend
Method: Use Many Different Approaches to Concurrency

Abstract	Concrete
Locks and Shared Memory Synchronization	Prefix Sum with pthreads
Language Support	Go lab: condition variables, channels, go routines Rust lab: 2PC
Parallel Architectures	GPU Programming Lab ***(Optional) FPGA Programming Lab***
HPC	Optional MPI lab
Distributed Computing / Big Data	Rust 2PC / MPI labs
Modern/Advanced Topics	• Specialized Runtimes / Programming Models • Auto-parallelization • Race Detection
Whatever Interests YOU	Project

Logistics Reprise

Course Name:	**CS378 – Concurrency**
Unique Number:	53015
Lectures:	MW 9:30-11:00AM UTC 144 and/or zoom
Class Web Page:	**http://www.cs.utexas.edu/users/rossbach/cs378**
Instructor:	Chris Rossbach
TA:	Carter Chu
Text:	Principles of Parallel Programming (ISBN-10: 0321487907)



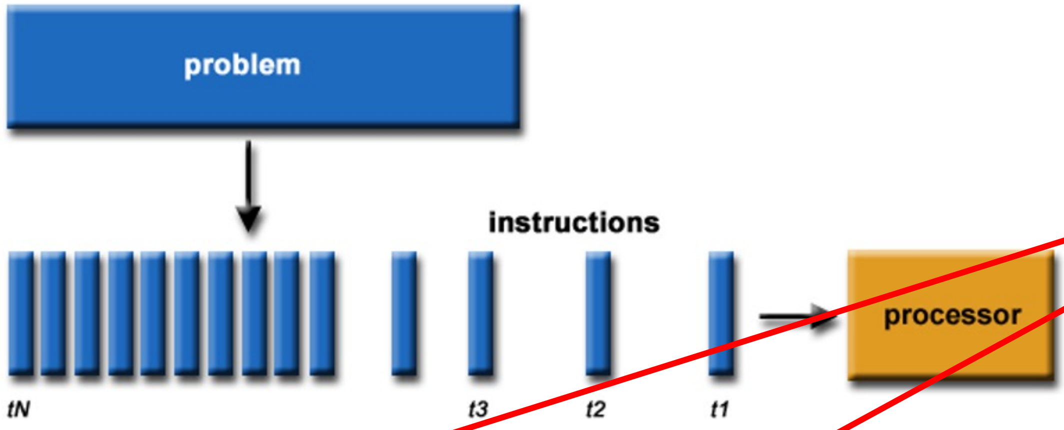
Seriously, read the syllabus!
Also, start Lab 1!

Two Super-Serious Notes

- Inclusivity and respect are *absolute* musts

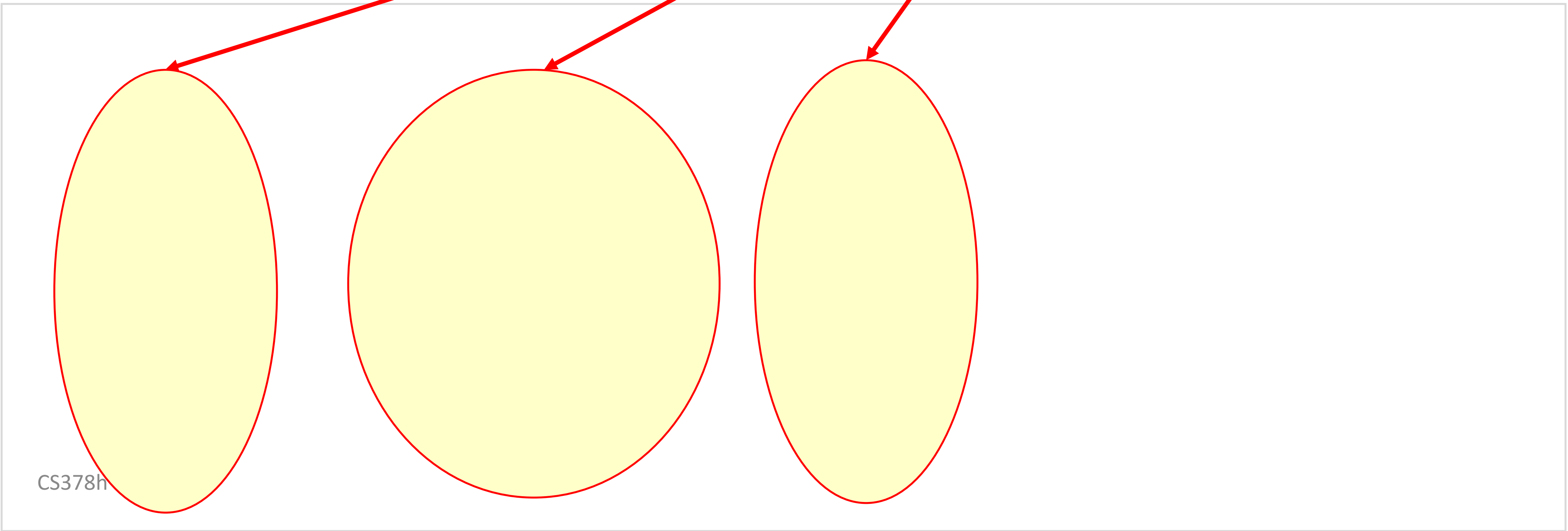
- Don’t make your repos public or look at other people’s public repos
- Don’t make your repos public or look at other people’s public repos
- Don’t make your repos public or look at other people’s public repos
- Don’t make your repos public or look at other people’s public repos
- Don’t make your repos public or look at other people’s public repos

Serial vs. Parallel Program

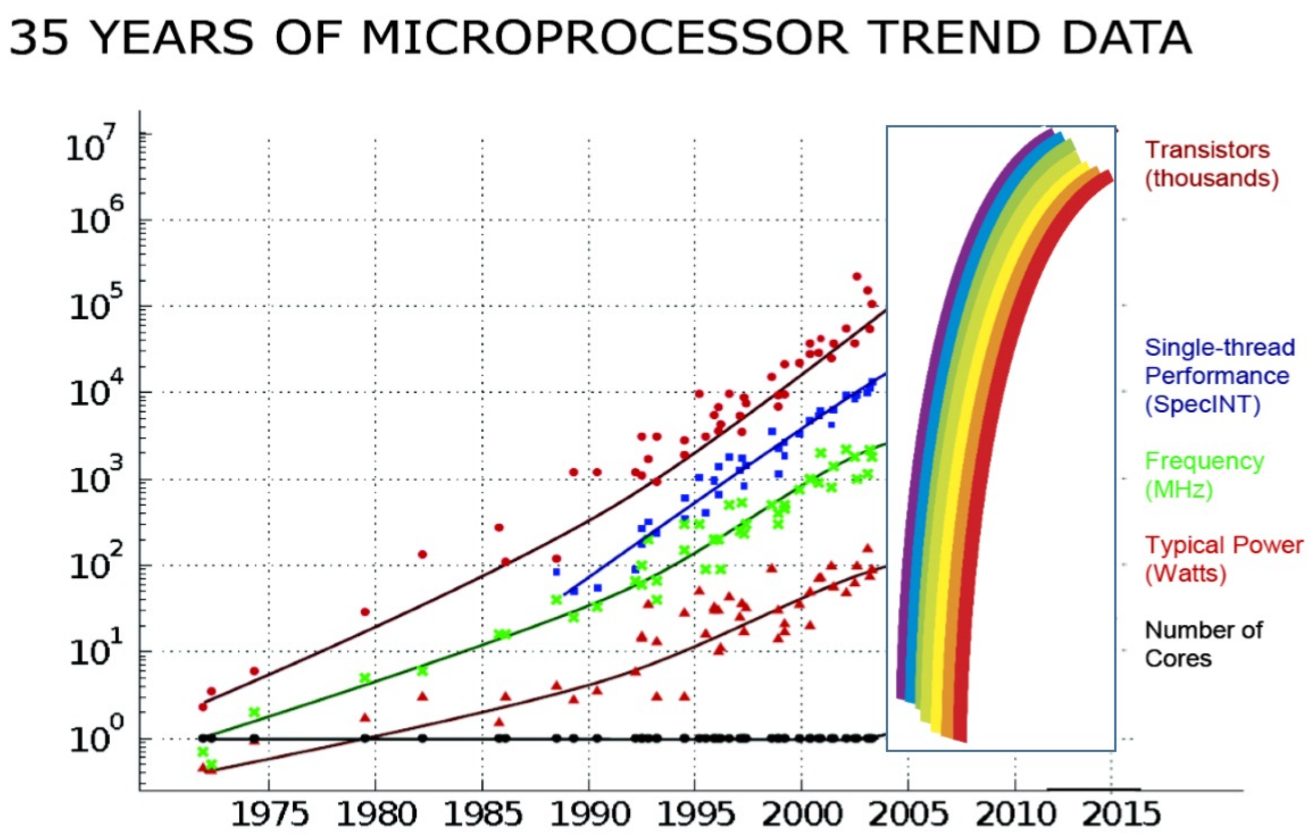


Key concerns:

- Programming model
- Execution Model
- Performance/Efficiency
- Exposing parallelism

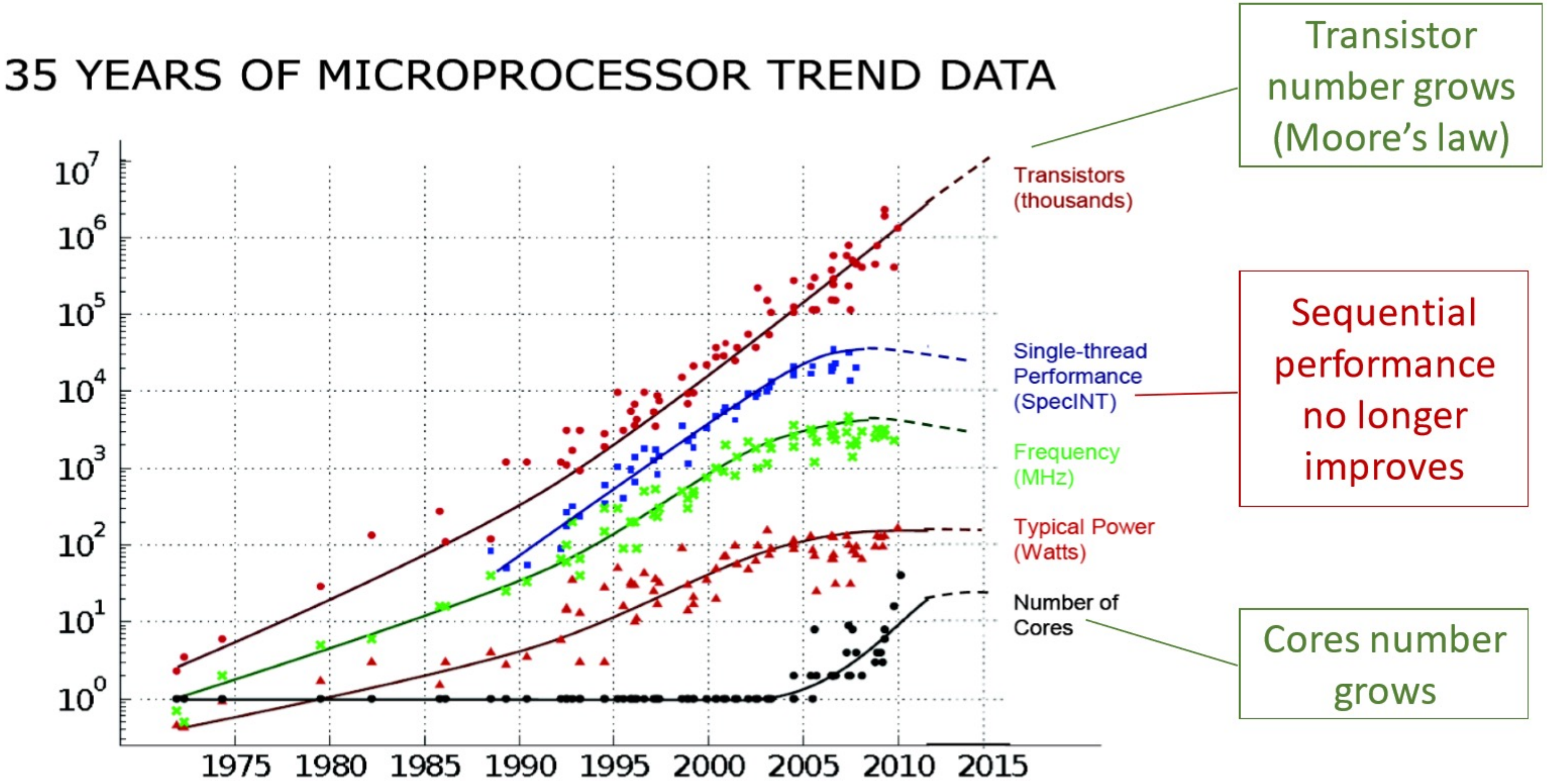


Free lunch…



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Free lunch – is over ☹



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

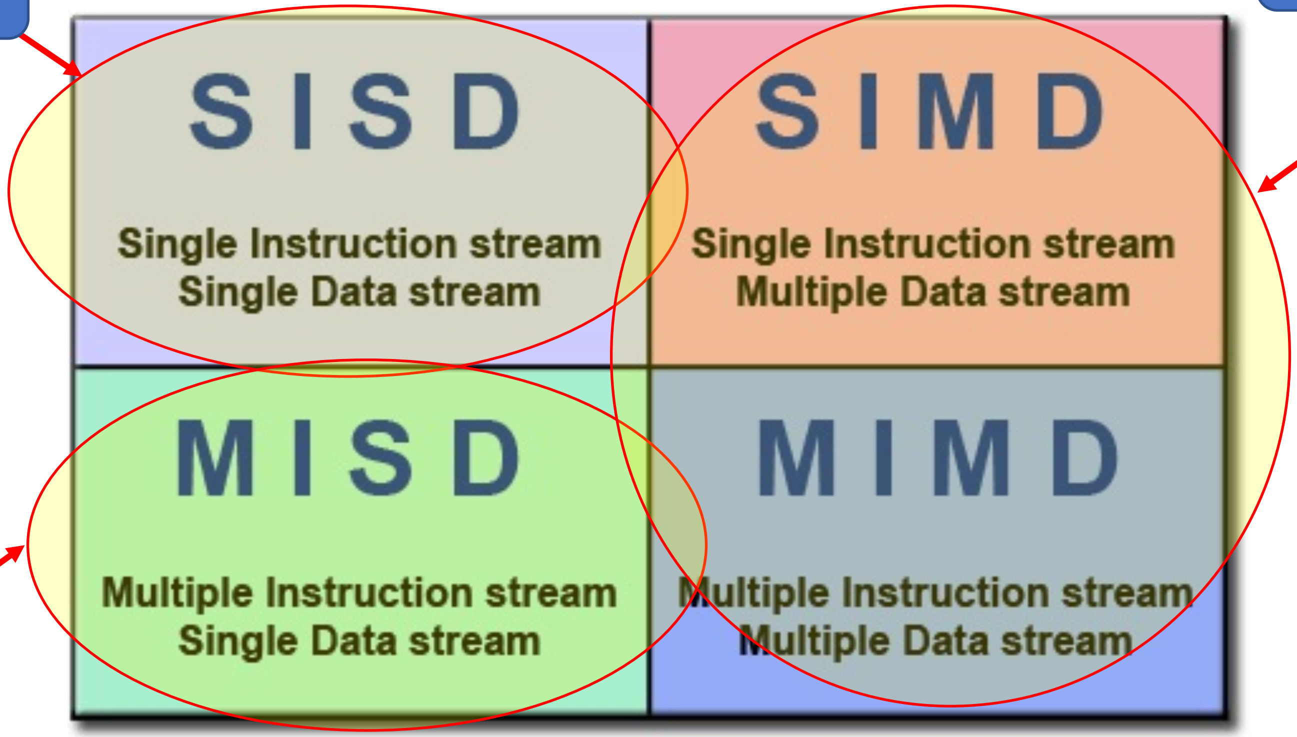
Flynn’s Taxonomy

SISD	SIMD
MISD	MIMD

Execution Models: Flynn's Taxonomy

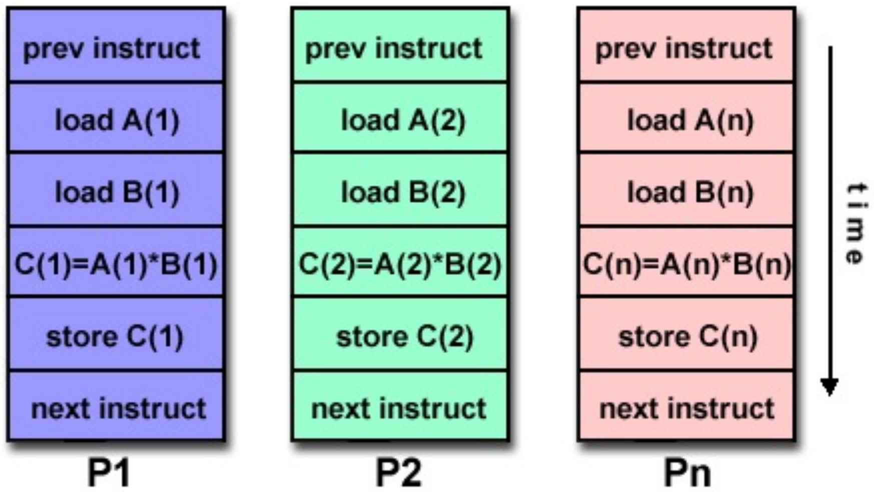
Normal Serial program

Our main focus

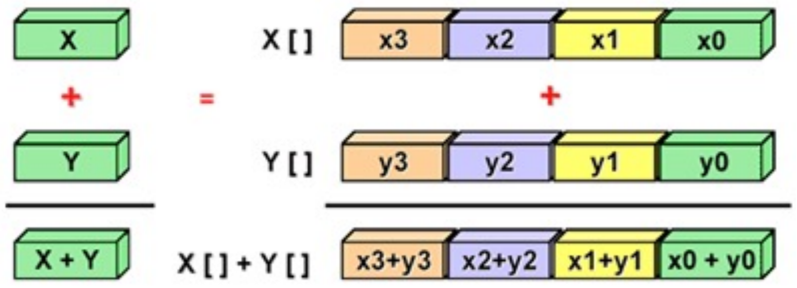


Uncommon architecture:
Fault – tolerance
Pipeline parallelism

SIMD

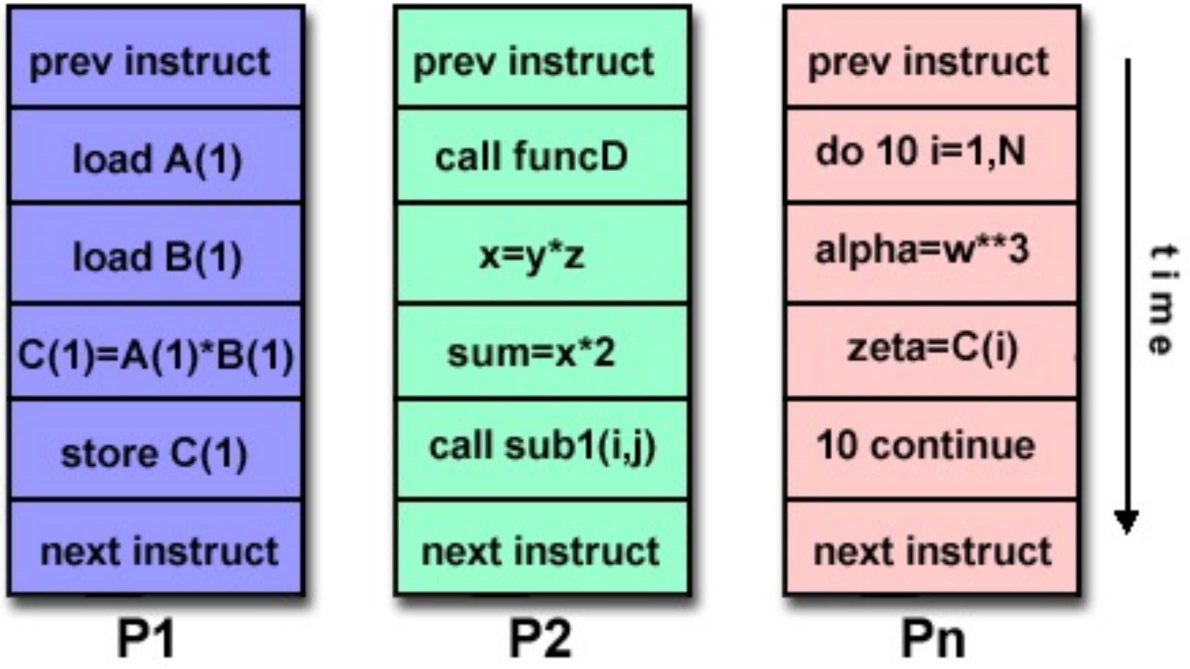


- Example: vector operations (e.g., Intel SSE/AVX, GPU)



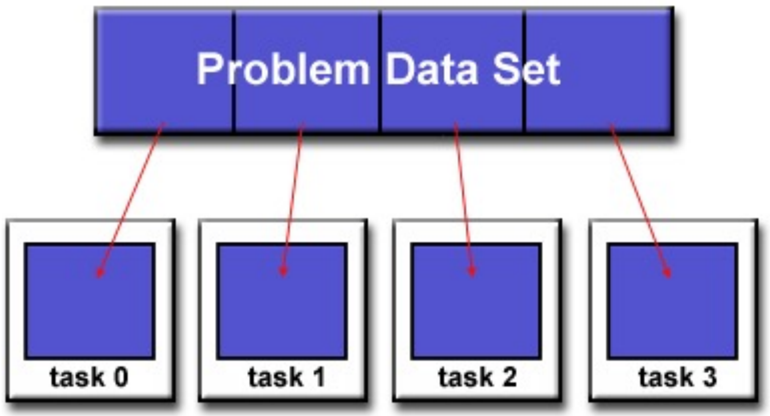
MIMD

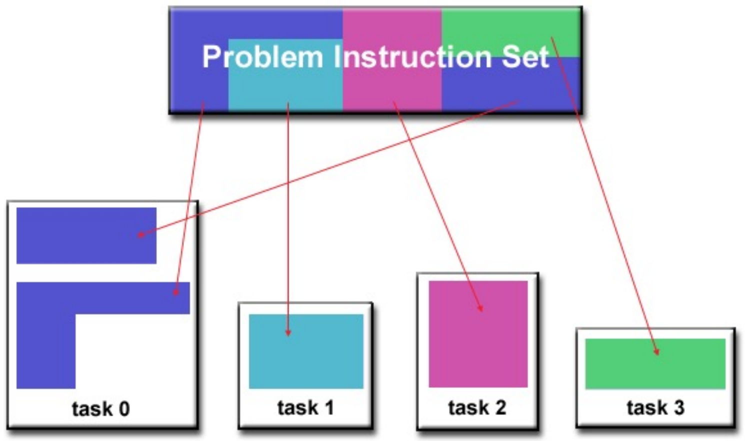
- Example: multi-core CPU



Problem Partitioning

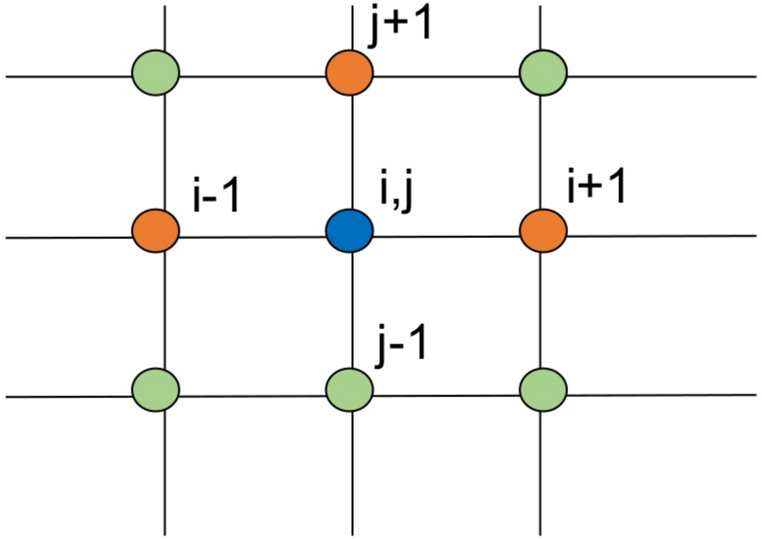
- Decomposition: Domain v. Functional
- Domain Decomposition
  - SPMD
  - Input domain
  - Output Domain
  - Both
- Functional Decomposition
  - MPMD
  - Independent Tasks
  - Pipelining





Game of Life

- Given a 2D Grid:
- $v_t(i,j) = F\big(v_{t-1}(of\ all\ its\ neighbors)\big)$

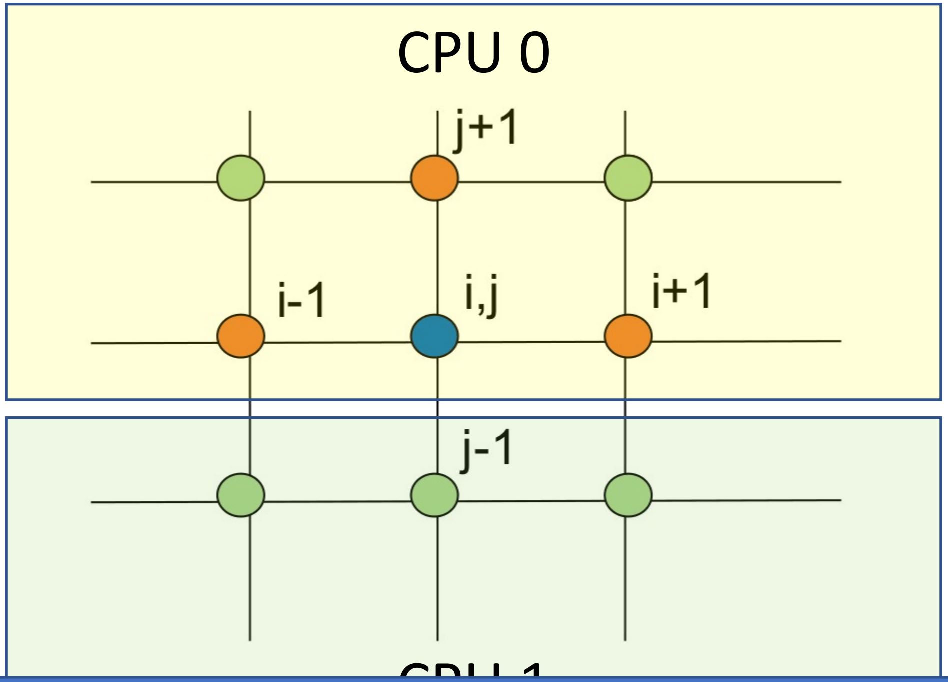


What model fits “best”?

SISD Single Instruction stream Single Data stream	**SIMD** Single Instruction stream Multiple Data stream
MISD Multiple Instruction stream Single Data stream	**MIMD** Multiple Instruction stream Multiple Data stream

Domain decomposition

- Each CPU gets part of the input



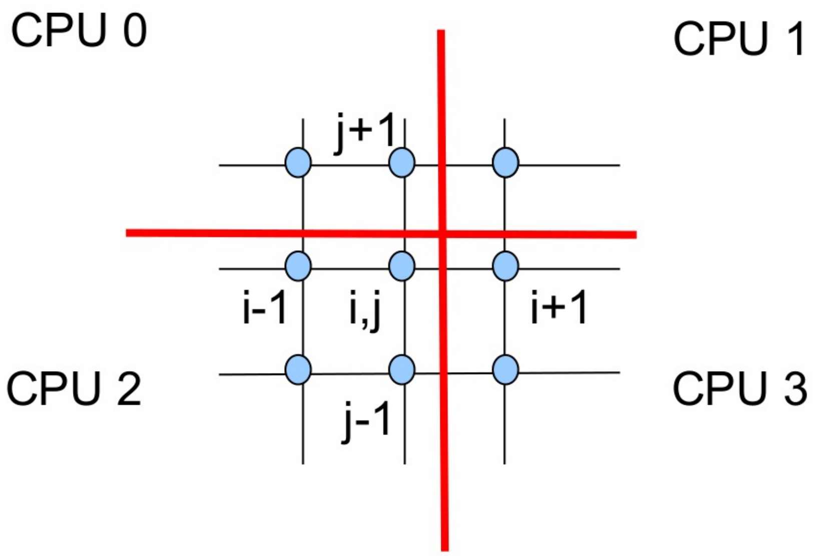
How could we do a functional decomposition?

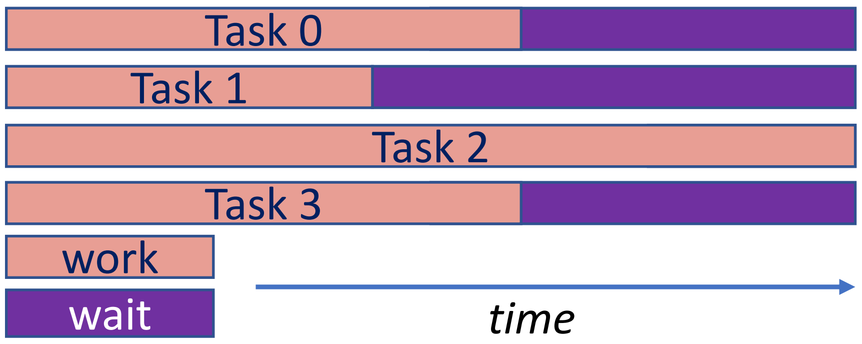
Issues?

- Accessing Data
    - Can we access v(i+1, j) from CPU 0
        - …as in a “normal” serial program?
        - Shared memory? Distributed?
    - Time to access v(i+1,j) == Time to access v(i-1,j) ?
    - *Scalability vs Latency*
- Control
    - Can we assign one vertex per CPU?
    - Can we assign one vertex per process/logical task?
    - *Task Management Overhead*
- *Load Balance*
- Correctness
    - order of reads and writes is non-deterministic
    - synchronization is required to enforce the order
    - *locks, semaphores, barriers, conditionals….*

Load Balancing

- Slowest task determines performance

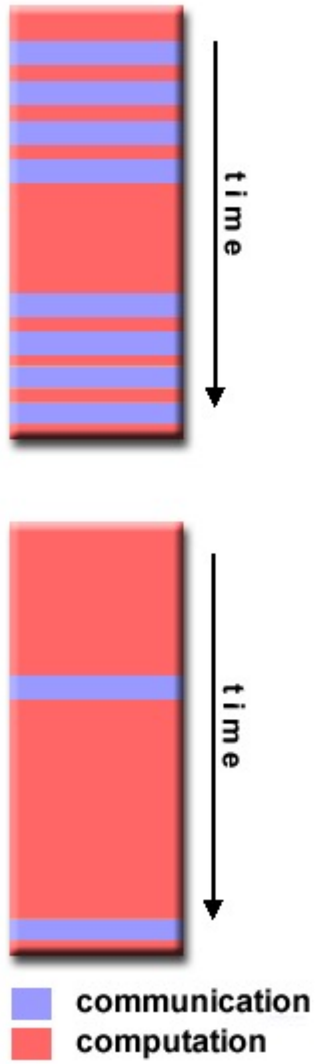




Granularity

$$G = \frac{Computation}{Communication}$$

- Fine-grain parallelism
  - G is small
  - Good load balancing
  - Potentially high overhead
  - Hard to get correct
- Coarse-grain parallelism
  - G is large
  - Load balancing is tough
  - Low overhead
  - Easier to get correct



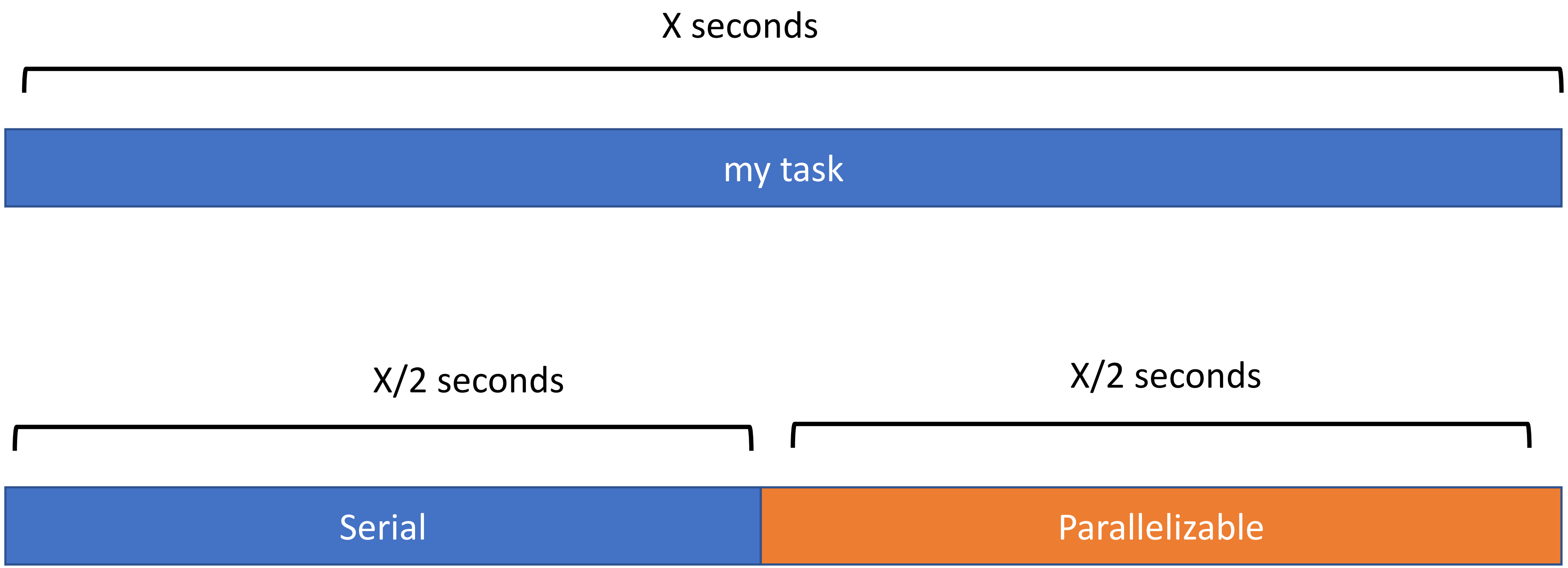
Performance: Amdahl's law

- Sp
- Sp

$$Speedup = \frac{\text{serial run time}}{\text{parallel run time}}$$

$$Speedup(\#CPUs) = \frac{T_{serial}}{T_{parallel}} = \frac{1}{\frac{A}{\#CPUs} + (1 - A)}$$

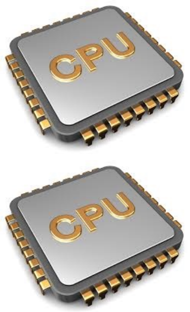
Amdahl's law

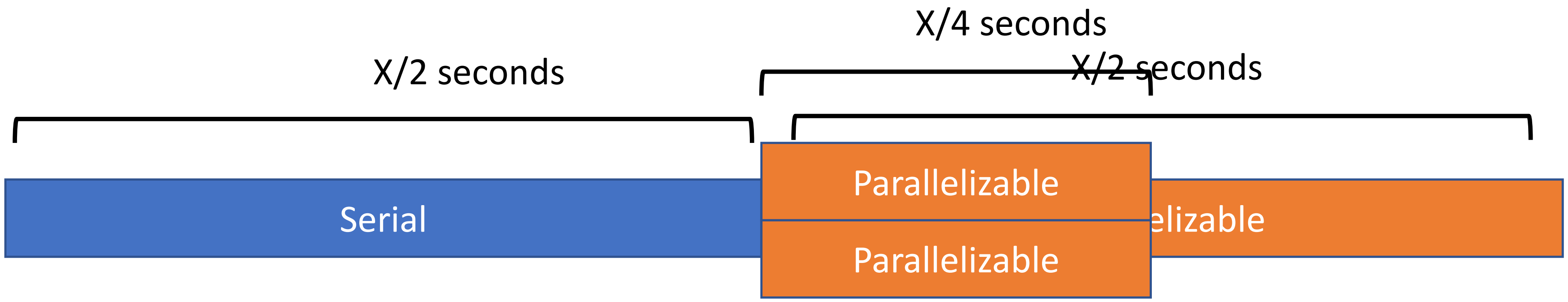


What makes something "serial" vs. parallelizable?

Amdahl's law

2 CPUs





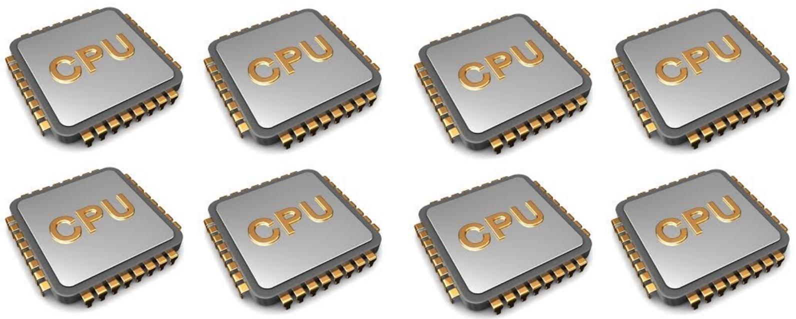
End to end time: (X/2 + X/4) = (3/4)X seconds

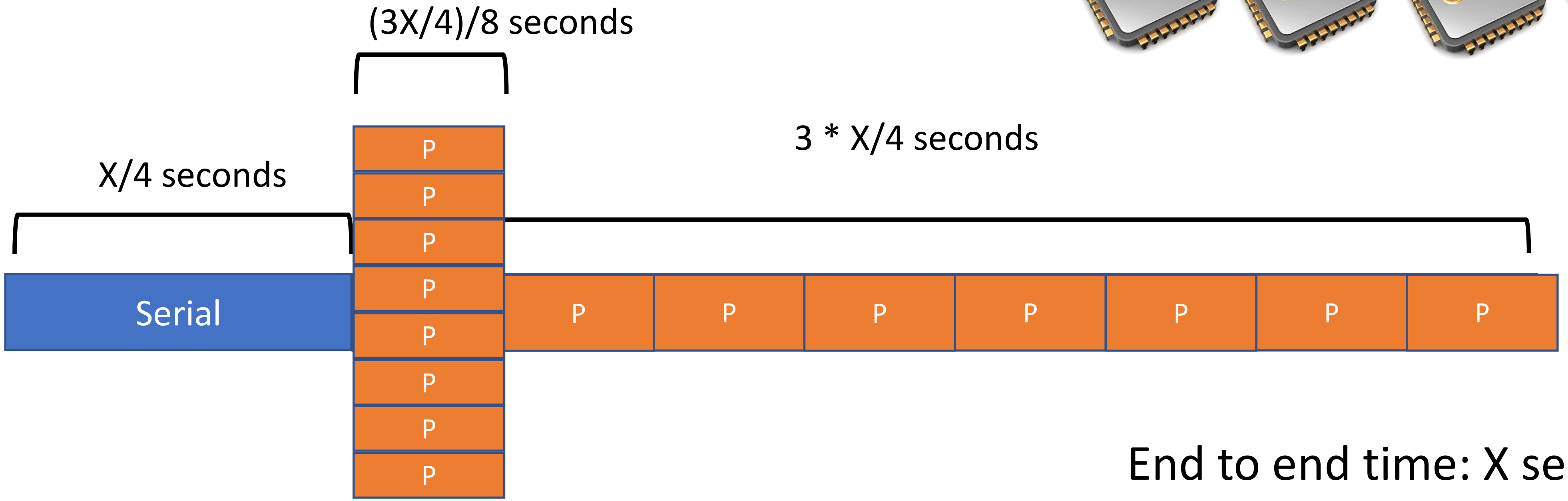
What is the "speedup" in this case?

$$Speedup = \frac{\text{serial run time}}{\text{parallel run time}} = \frac{1}{\frac{A}{\#CPUs} + (1 - A)} = \frac{1}{\frac{.5}{2\ \text{cpus}} + (1-.5)} = 1.333$$

Speedup exercise

8 CPUs



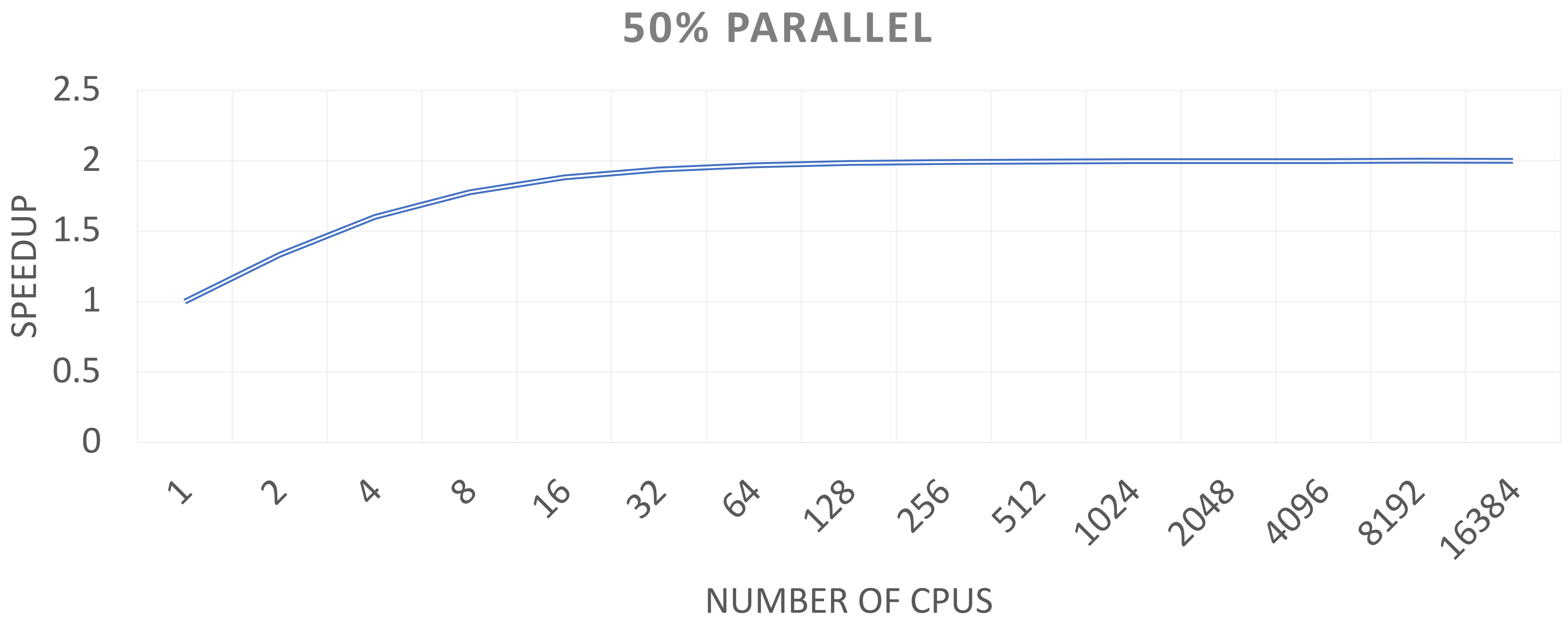


End to end time: X seconds

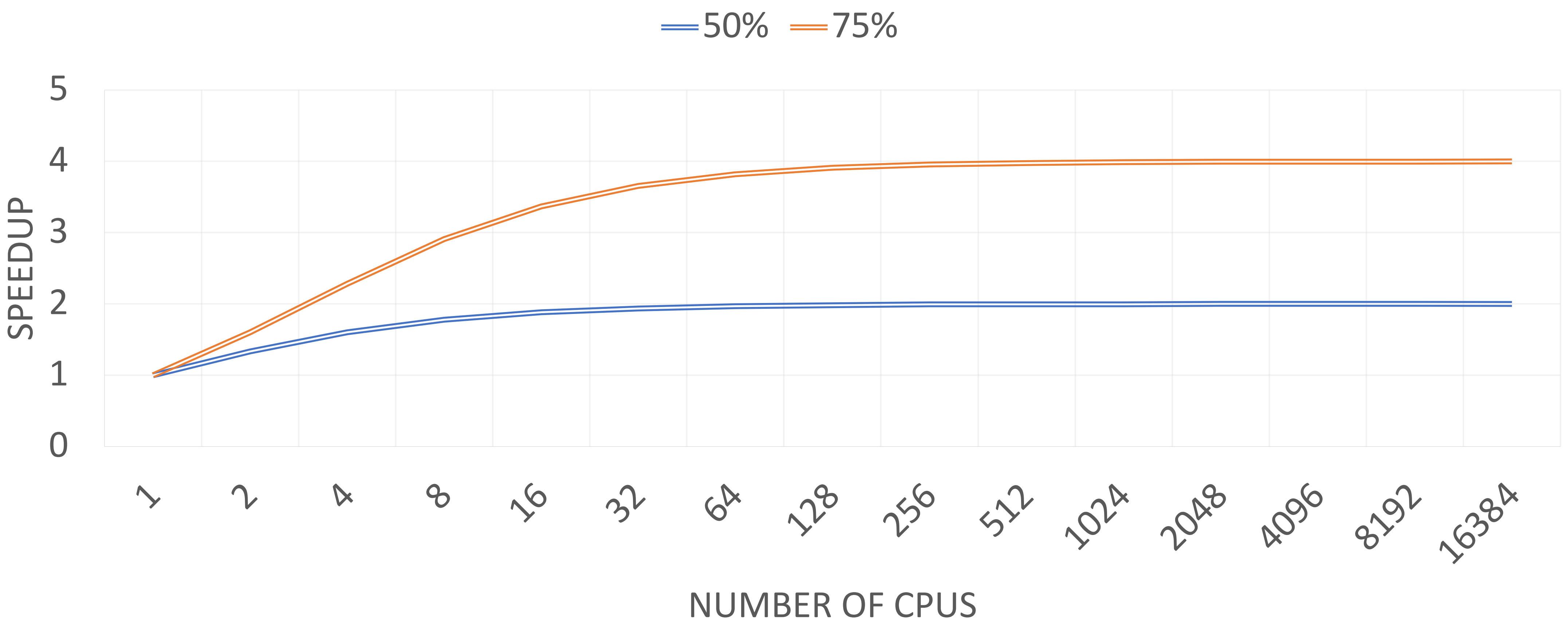
What is the “speedup” in this case?

$$Speedup = \frac{\text{serial run time}}{\text{parallel run time}} = \frac{1}{\frac{A}{\#CPUs} + (1 - A)} = \frac{1}{.75/8 + (1-.75)} = 2.91\text{x}$$

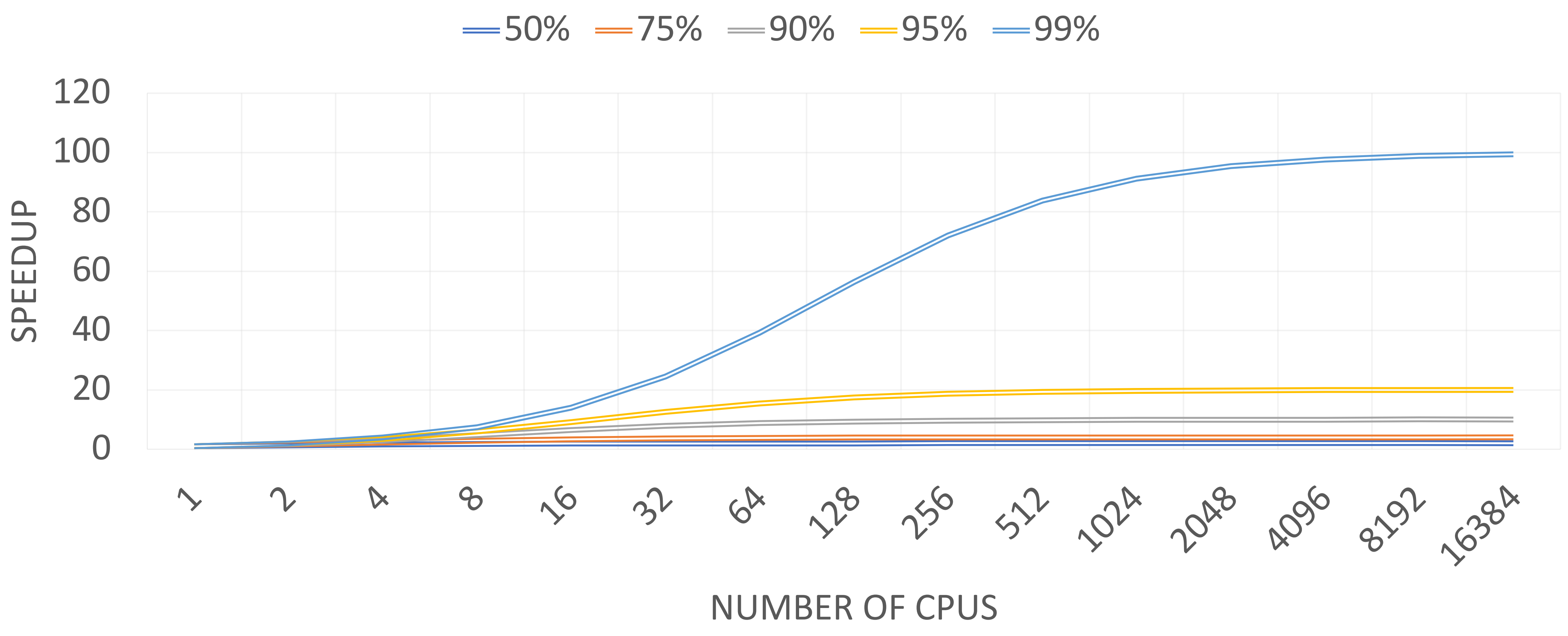
Amdahl Action Zone



Amdahl Action Zone



Amdahl Action Zone

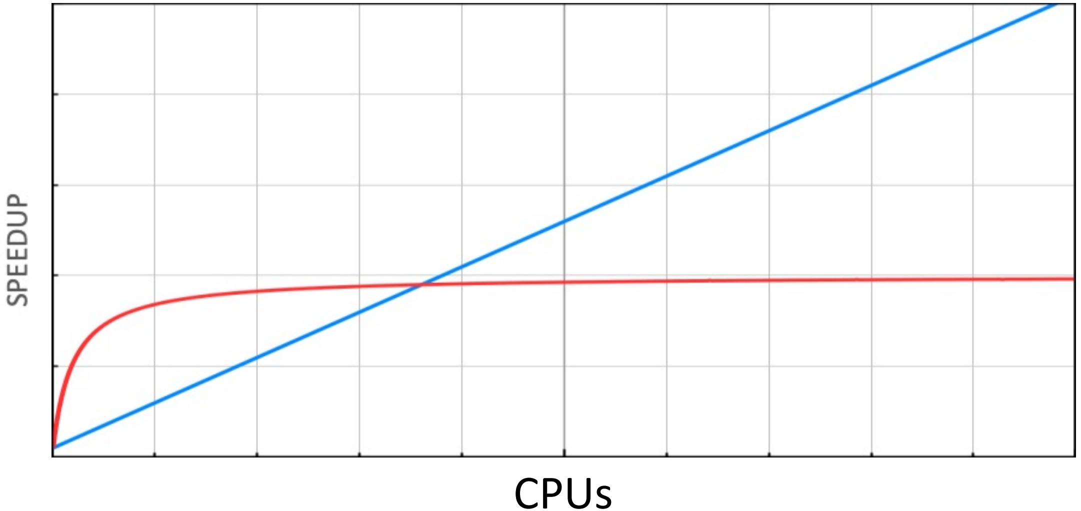


Strong Scaling vs Weak Scaling

- Amdahl vs. Gustafson
  - Amdahl: strong scaling → fixed work
  - Gustafson: scaling → add more work *and* more processors
- Given work W on n CPUs, with α serial
  - Incremental work W’ on (n+1) CPUs:
    $W' = \alpha W + (1-\alpha) n W$
  - Speedup based on case where (1-α) scales perfectly:

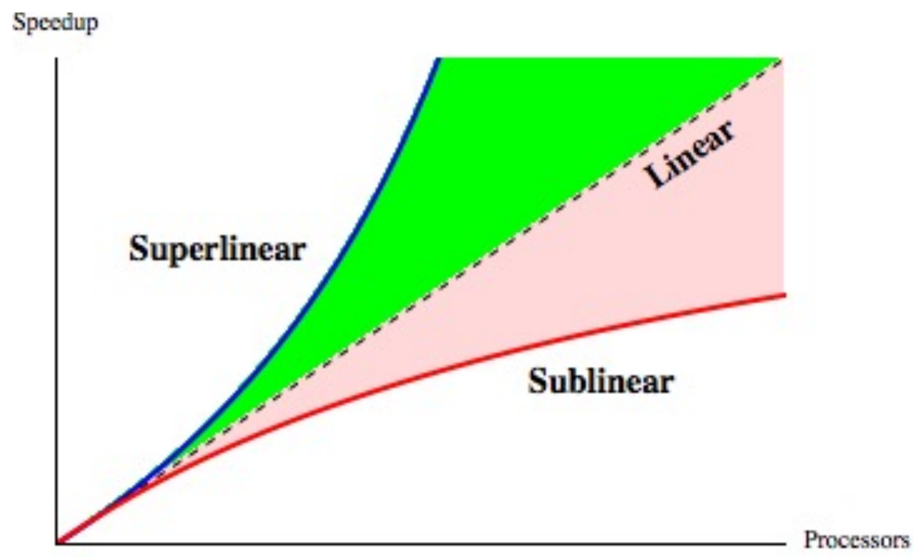
$$S(n) = \frac{\alpha W + (1-\alpha) n W}{\alpha W + \frac{(1-\alpha) n W}{n}}$$

$$S(n) = \alpha + (1-\alpha) n$$



Super-linear speedup

- Possible due to cache
- But usually just poor methodology
- Baseline: ***best*** serial algorithm
- Example:
  - Efficient **bubble sort** takes:
    - Parallel 40s
    - Serial 150s
    - $Speedup = \frac{150}{40} = 3.75$?
  - NO!
    - Serial quicksort runs in 30s
    - $\Longrightarrow Speedup = 0.75$



Concurrency and Correctness

If two threads execute this program concurrently, how many different final values of X are there?

Initially, X == 0.

Thread 1

```
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}
```

Thread 2

```
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}
```

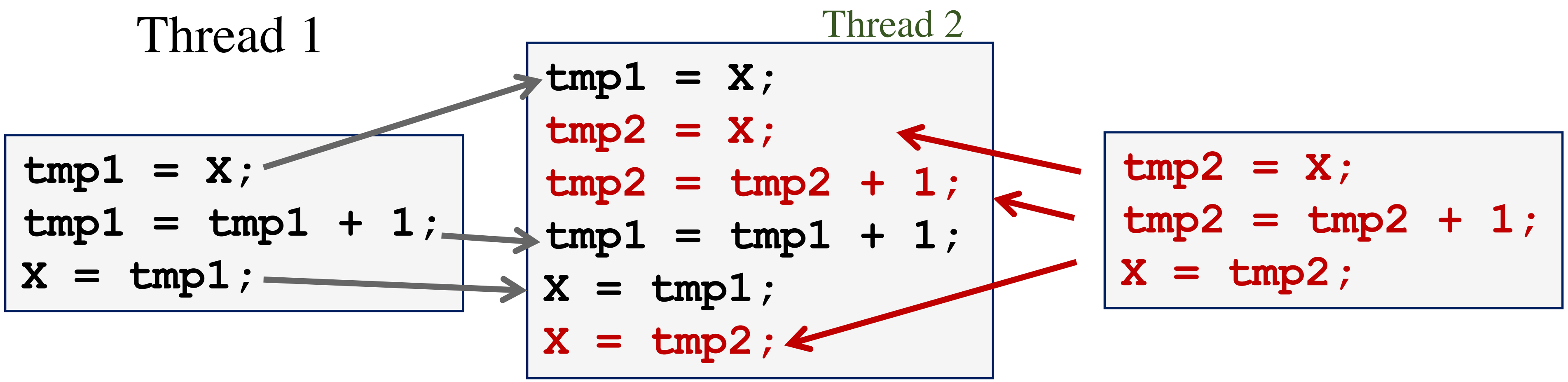
Answer:

- **A. 0**
- **B. 1**
- **C. 2**
- **D. More than 2**

Schedules/Interleavings

Model of concurrent execution

- Interleave statements from each thread into a single thread
- If **any** interleaving yields incorrect results, synchronization is needed



If X==0 initially, X == 1 at the end. WRONG result!

Locks fix this with Mutual Exclusion

```
void increment() {
    lock.acquire();
    int temp = X;
    temp = temp + 1;
    X = temp;
    lock.release();
}
```

Mutual exclusion ensures only safe interleavings

- *But it limits concurrency, and hence scalability/performance*

Is mutual exclusion a good abstraction?

Why Locks are Hard

- Coarse-grain locks
  - Simple to develop
  - Easy to avoid deadlock
  - Few data races
  - Limited concurrency

```
// WITH FINE-GRAIN LOCKS
void move(T s, T d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
```

- Fine-grain locks
  - Greater concurrency
  - Greater code complexity
  - Potential deadlocks
    - Not composable
  - Potential data races
    - Which lock to lock?

Thread 0	Thread 1
`move(a, b, key1);`	
	`move(b, a, key2);`

DEADLOCK!

The correctness conditions

- Safety
  - Only one thread in the critical region
- Liveness
  - Some thread that enters the entry section eventually enters the critical region
  - Even if other thread takes forever in non-critical region
- Bounded waiting
  - A thread that enters the entry section enters the critical section within some bounded number of operations.
- Failure atomicity
  - It is OK for a thread to die in the critical region
  - Many techniques do not provide failure atomicity

```
while(1) {
   Entry section
   Critical section
   Exit section
   Non-critical section
}
```

Read-Modify-Write (RMW)

- Implement locks using read-modify-write instructions
  - As an atomic and isolated action
    1. read a memory location into a register, **AND**
    2. write a new value to the location
  - Implementing RMW is tricky in multi-processors
    - Requires cache coherence hardware. Caches snoop the memory bus.

- Examples:
  - Test&set instructions (most architectures)
    - Reads a value from memory
    - Write "1" back to memory location
  - Compare & swap (68000)
    - Test the value against some constant
    - If the test returns true, set value in memory to different value
    - Report the result of the test in a flag
    - if [addr] == r1 then [addr] = r2;
  - Exchange, locked increment, locked decrement (x86)
  - Load linked/store conditional (PowerPC,Alpha, MIPS)

Implementing Locks with Test&set

```
int lock_value = 0;
int* lock = &lock_value;
```

```
Lock::Acquire() {
while (test&set(lock) == 1)
   ; //spin
}
```

(test & set ~ CAS ~ LLSC)

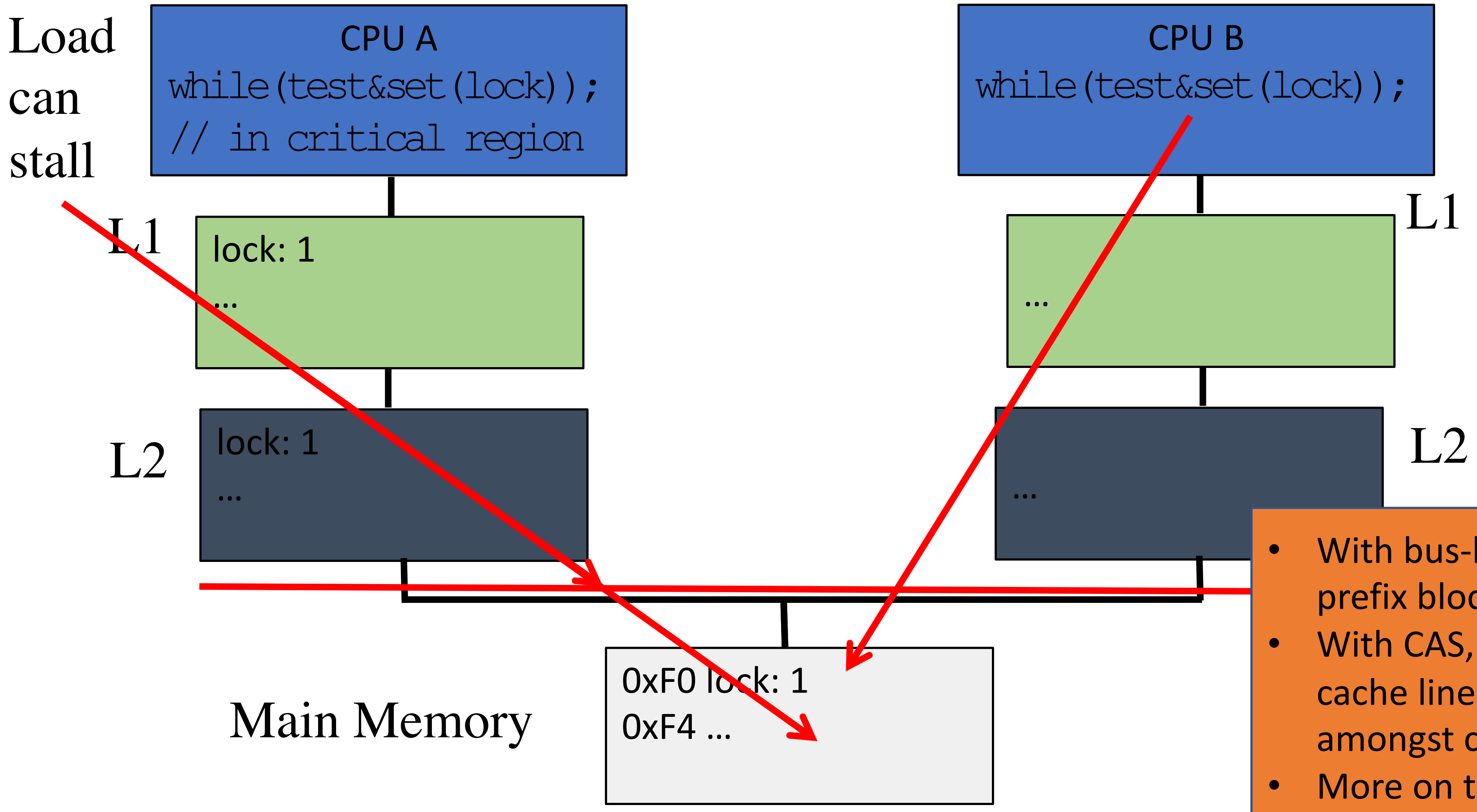
```
Lock::Release() {
   *lock = 0;
}
```

- What is the problem with this?
  - A. CPU usage B. Memory usage C. Lock::Acquire() latency
  - D. Memory bus usage E. Does not work

Test & Set with Memory Hierarchies

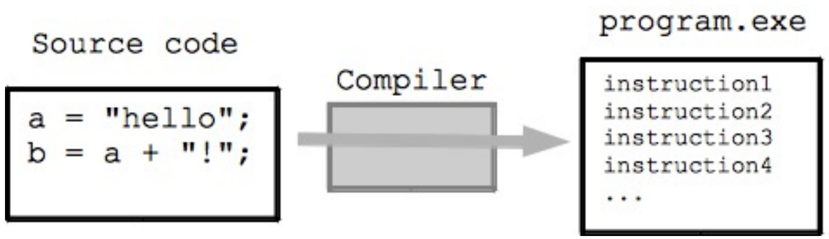
Initially, lock already held by some other CPU—A, B busy-waiting

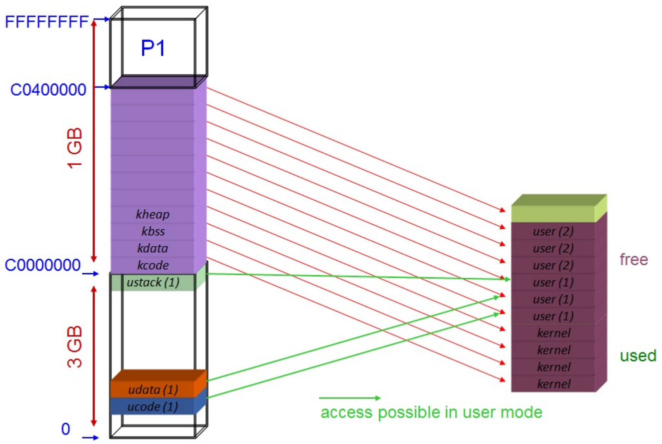
What happens to lock variable's cache line when different cpu's contend?



- With bus-locking, lock prefix blocks *everyone*
- With CAS, LL-SC, cache line cache line "ping pongs" amongst contenders
- More on this next time…

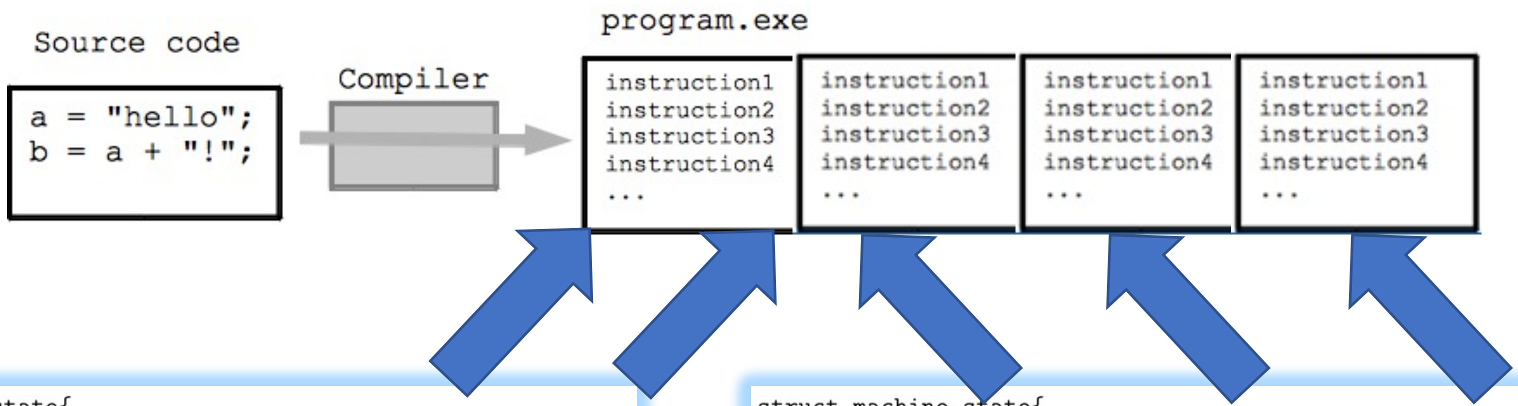
Programming and Machines: a mental model






```
struct machine_state{
  uint64 pc;
  uint64 Registers[16];
  uint64 cr[6]; // control registers cr0-cr4 and EFER on AMD
...
} machine;
while(1) {
  fetch_instruction(machine.pc);
  decode_instruction(machine.pc);
  execute_instruction(machine.pc);
}
void execute_instruction(i) {
  switch(opcode) {
  case add_rr:
   machine.Registers[i.dst] += machine.Registers[i.src];
   break;
}
```

Parallel Machines: a mental model

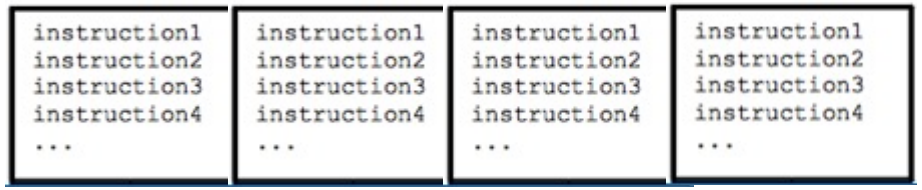


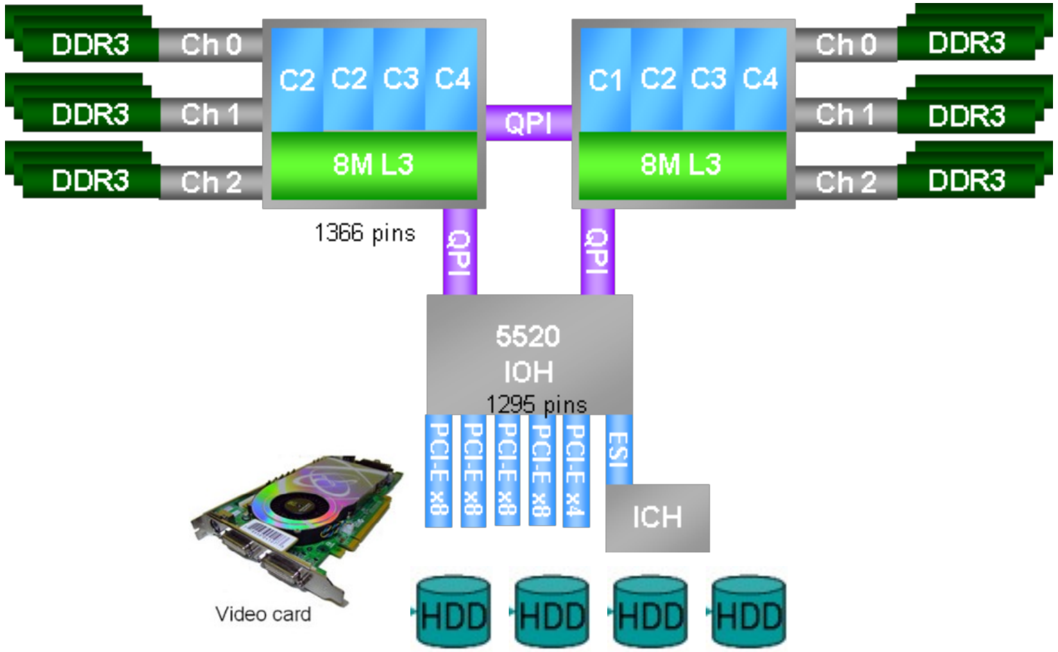

```
struct machine_state{
  uint64 pc;
  uint64 Registers[16];
  uint64 cr[6]; // control registers cr0-cr4 and EFER on AMD
...
} machine;
while(1) {
  fetch_instruction(machine.pc);
  decode_instruction(machine.pc);
  execute_instruction(machine.pc);
}
void execute_instruction(i) {
  switch(opcode) {
  case add_rr:
   machine.Registers[i.dst] += machine.Registers[i.src];
   break;
}
```

```
struct machine_state{
  uint64 pc;
  uint64 Registers[16];
  uint64 cr[6]; // control registers cr0-cr4 and EFER on AMD
...
} machine;
while(1) {
  fetch_instruction(machine.pc);
  decode_instruction(machine.pc);
  execute_instruction(machine.pc);
}
void execute_instruction(i) {
  switch(opcode) {
  case add_rr:
   machine.Registers[i.dst] += machine.Registers[i.src];
   break;
}
```

Processes and Threads

- Abstractions
- Containers
- State
    - Where is shared state?
    - How is it accessed?
    - Is it mutable?

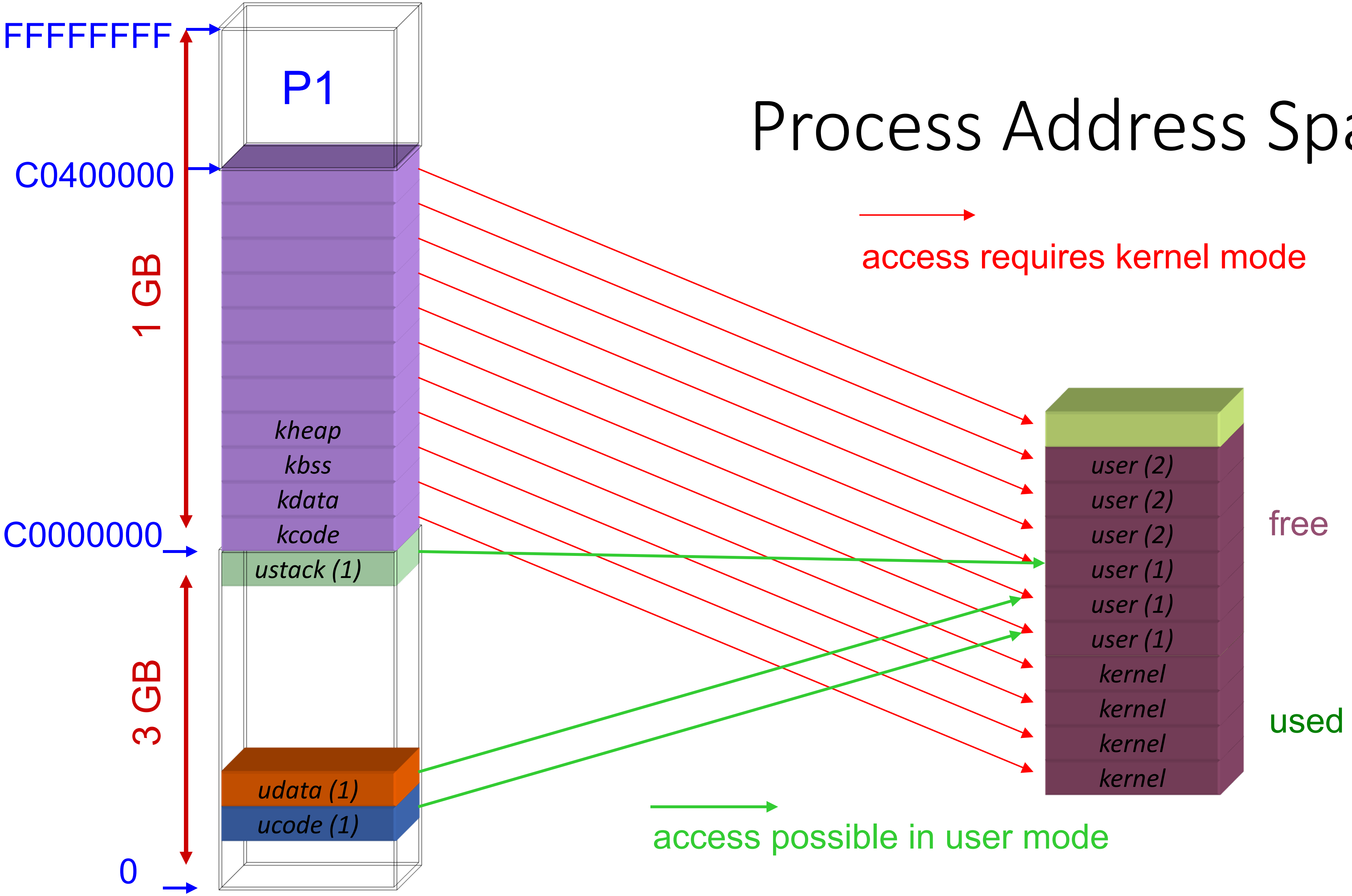




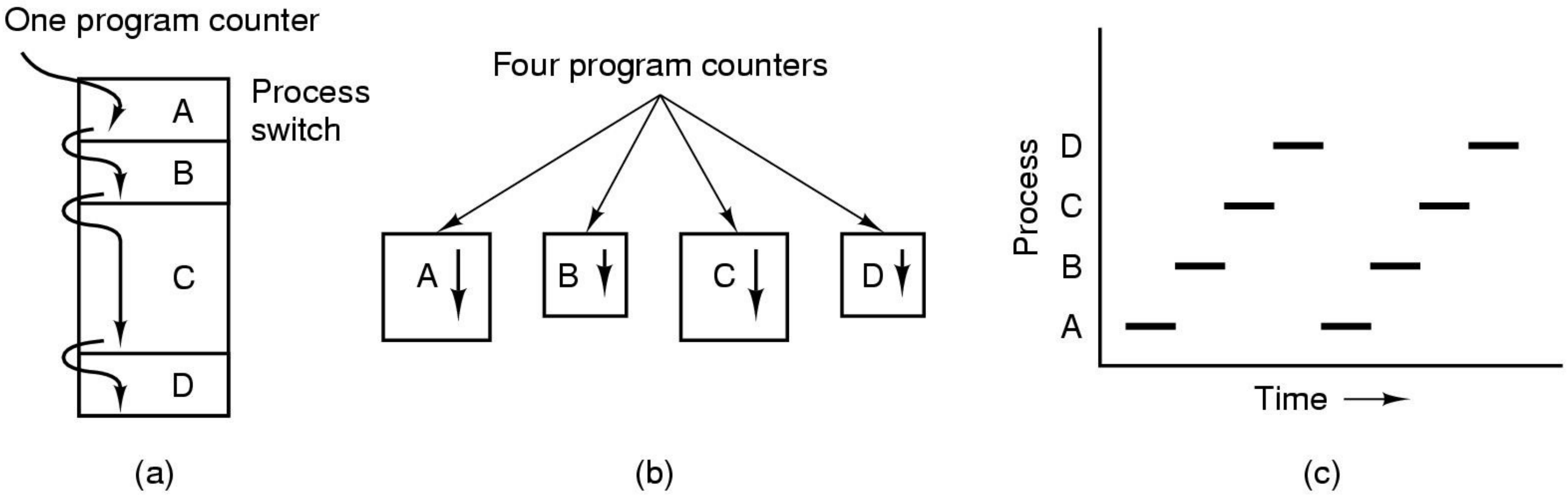
Processes & Virtual Memory

- Virtual Memory: Goals…what are they again?
- Abstraction: contiguous, isolated memory
  - Remember overlays?
- Prevent illegal operations
  - Access to others/OS memory
  - Fail fast (e.g. segv on *(NULL))
  - Prevent exploits that try to execute program data
- **Sharing mechanism/IPC substrate**

Process Address Space



Processes
The Process Model



- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant

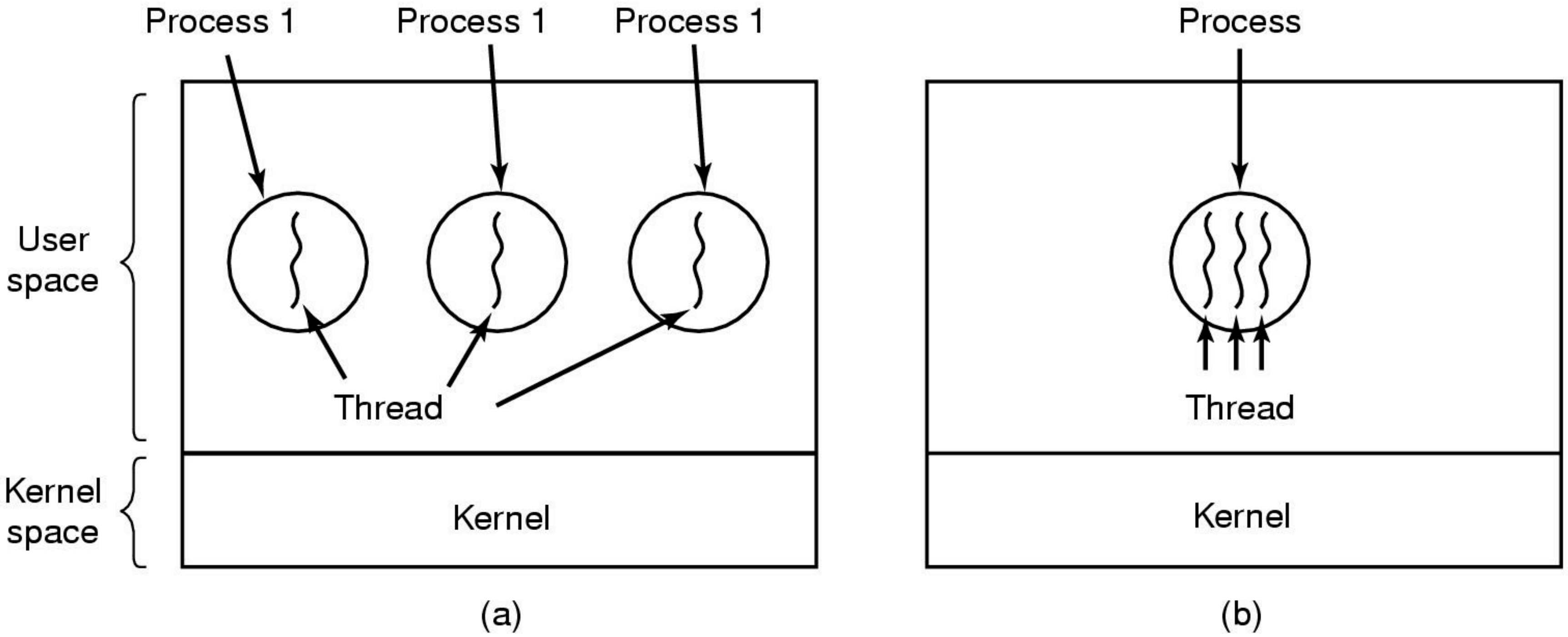
Implementation of Processes

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Fields of a process table entry

Threads

The Thread Model (1)



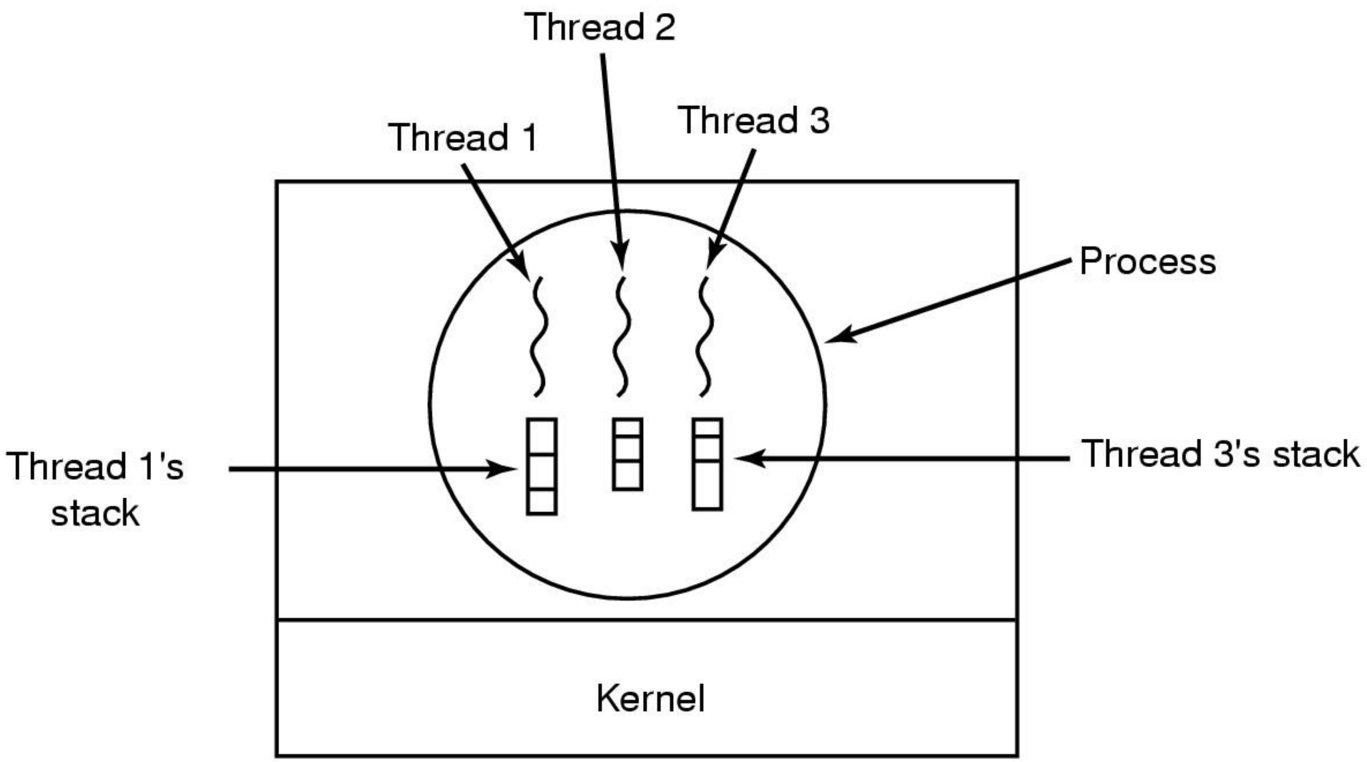
(a) Three processes each with one thread
(b) One process with three threads

The Thread Model

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

- Items shared by all threads in a process
- Items private to each thread

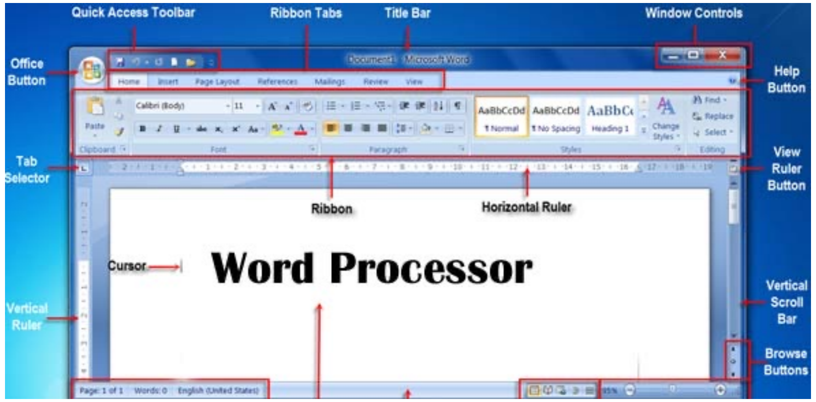
The Thread Model

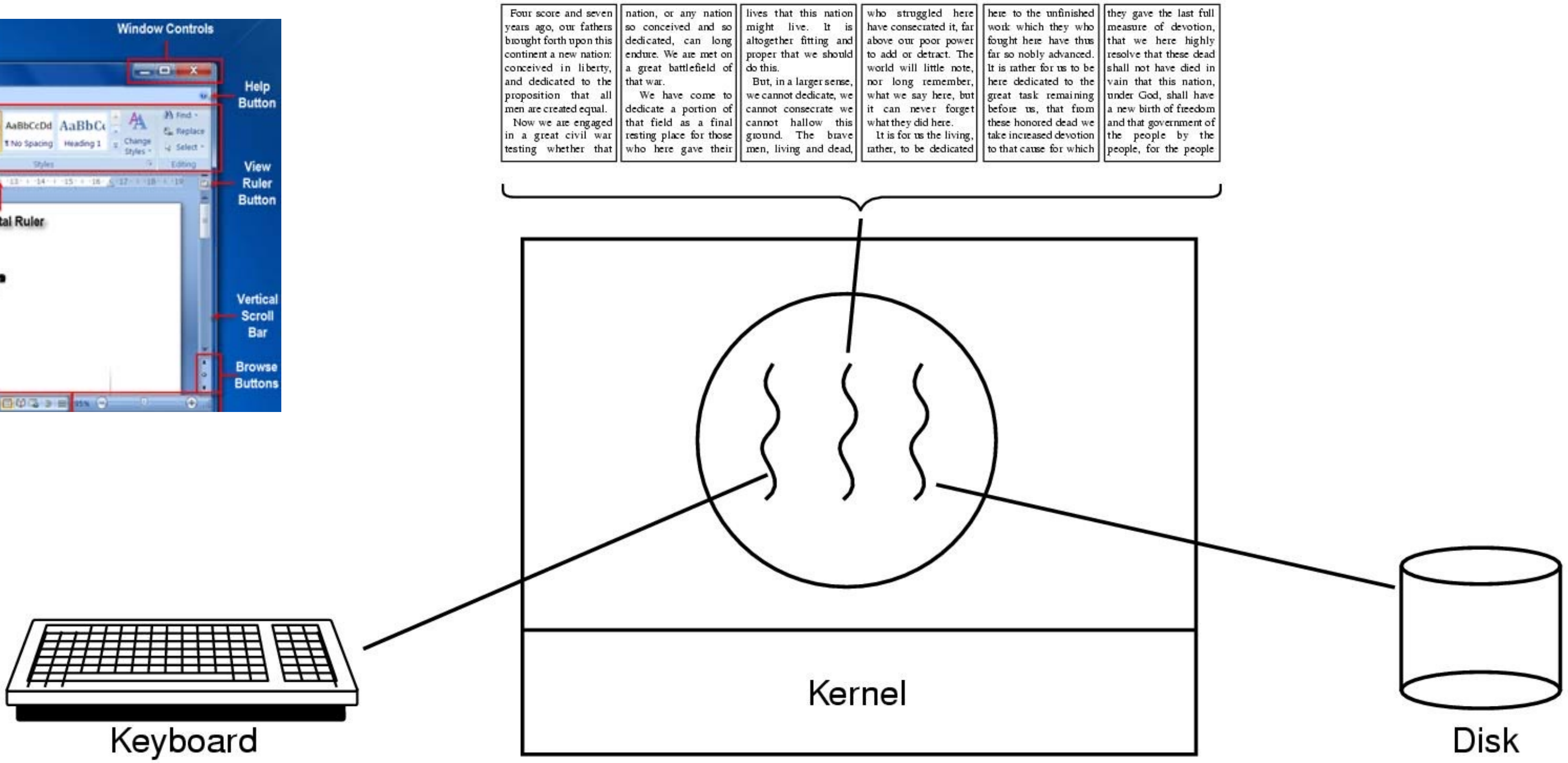


Each thread has its own stack

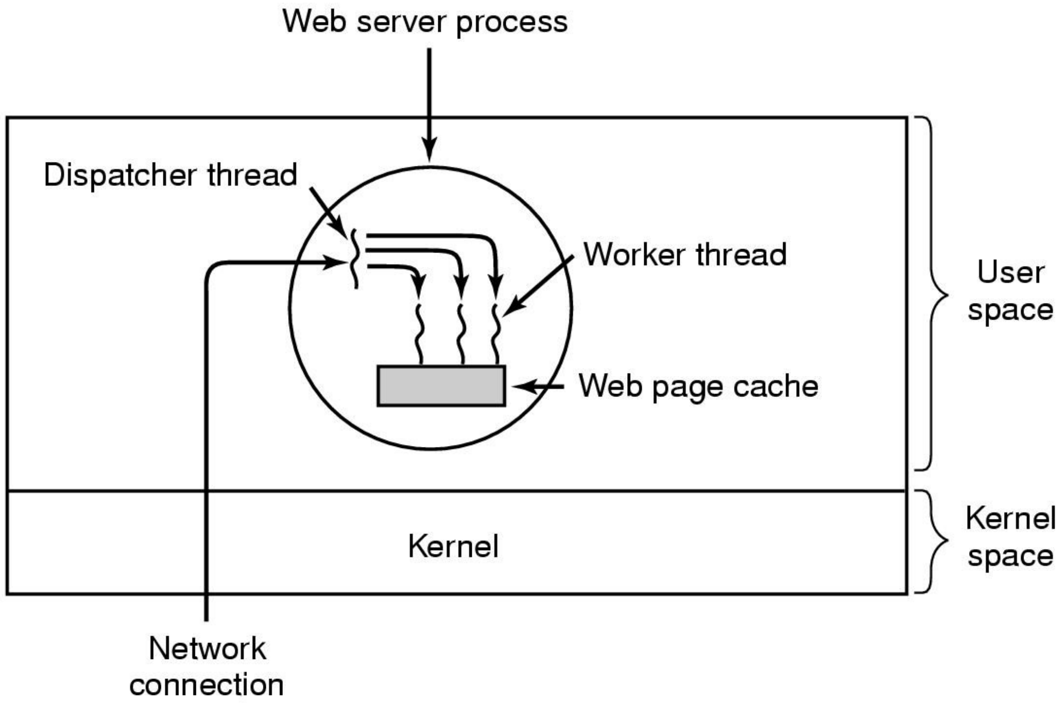
Using threads

Ex. How might we use threads in a word processor program?





Thread Usage



A multithreaded Web server

```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
      read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

(b)

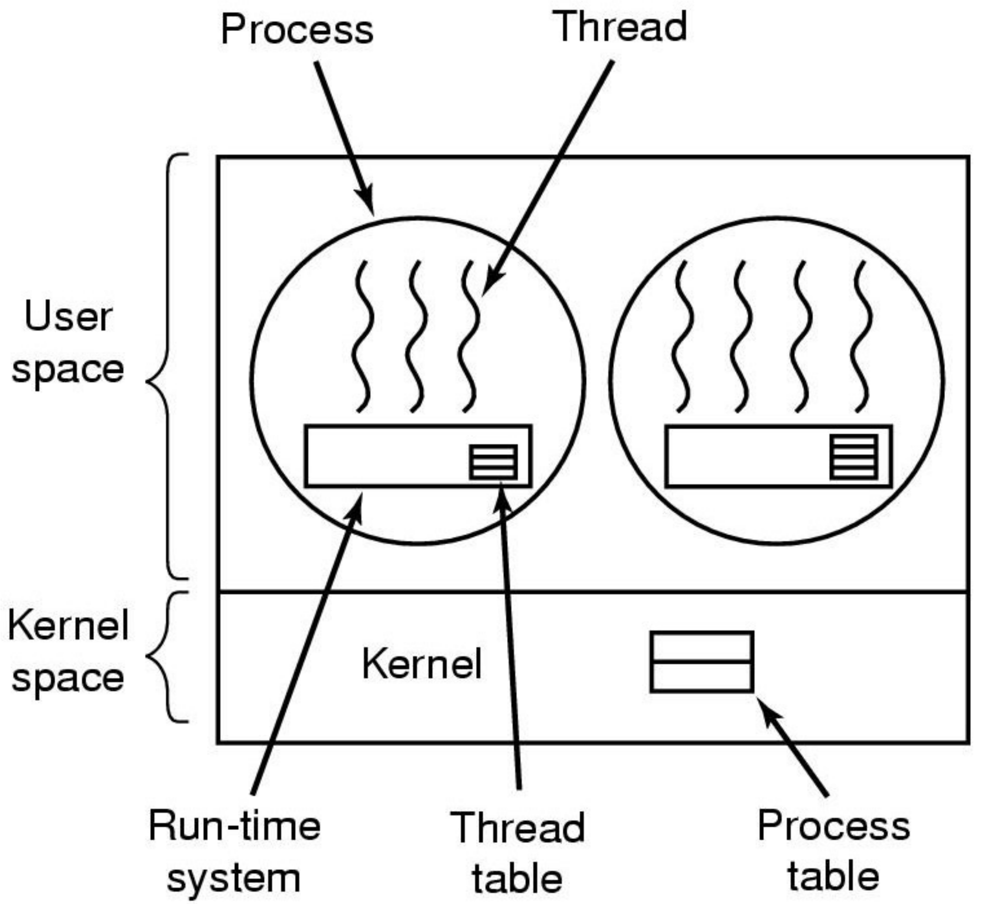
(a) Dispatcher thread
(b) Worker thread

Thread Usage

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

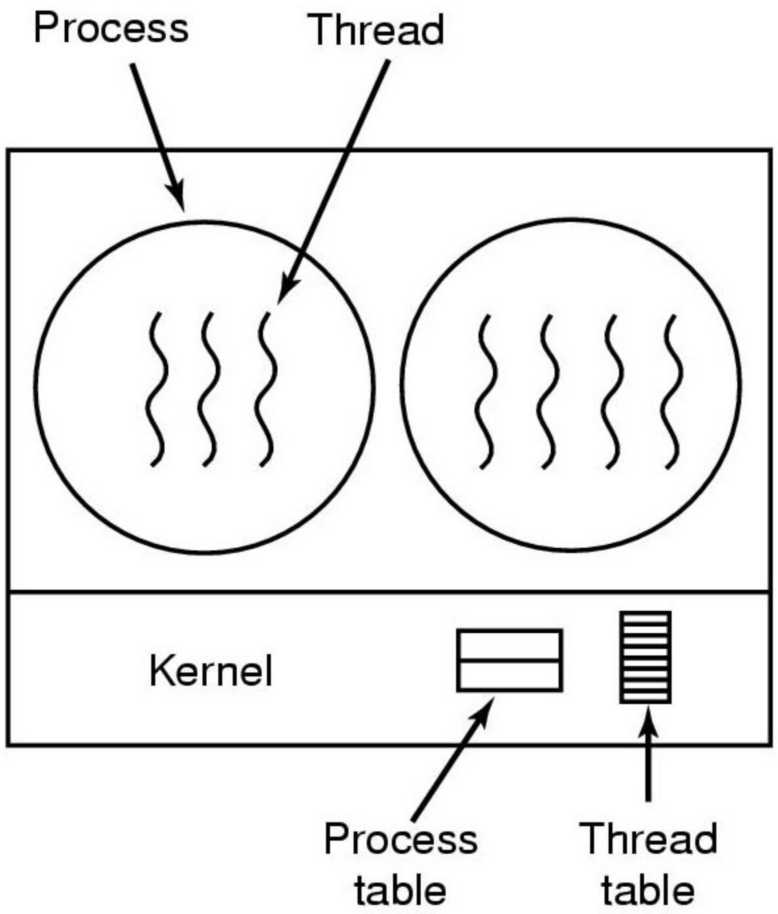
Three ways to construct a server

Implementing Threads in User Space



A user-level threads package

Implementing Threads in the Kernel



A threads package managed by the kernel

Pthreads

- POSIX standard thread model,
- Specifies the API and call semantics.
- Popular – most thread libraries are Pthreads-compatible

Preliminaries

- Include `pthread.h` in the main file
- Compile program with `-lpthread`
  - `gcc -o test test.c -lpthread`
  - may not report compilation errors otherwise but calls will fail
- Good idea to check return values on common functions

Thread creation

- Types: `pthread_t` – type of a thread
- Some calls:

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void * (*start_routine)(void *),
                   void *arg);
int pthread_join(pthread_t thread, void **status);
int pthread_detach();
void pthread_exit();
```

  - No explicit parent/child model, except main thread holds process info
  - Call `pthread_exit` in main, don't just fall through;
  - Most likely you wouldn't need `pthread_join`
    - `status` = exit value returned by joinable thread
  - Detached threads are those which cannot be joined (can also set this at creation)

Creating multiple threads

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4

void *hello (void *arg) {
      printf("Hello Thread\n");
}

main() {
  pthread_t tid[NUM_THREADS];
  for (int i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], NULL, hello, NULL);

  for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}
```

Can you find the bug here?

What is printed for myNum?

```
void *threadFunc(void *pArg) {
  int* p = (int*)pArg;
  int myNum = *p;
  printf( "Thread number %d\n", myNum);
}
. . .
// from main():
for (int i = 0; i < numThreads; i++) {
   pthread_create(&tid[i], NULL, threadFunc, &i);
}
```

Pthread Mutexes

- Type: `pthread_mutex_t`

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- Attributes: for shared mutexes/condition vars among processes, for priority inheritance, etc.
  - use defaults
- Important: Mutex scope must be visible to all threads!

Spinlock vs Mutex

Lab #1

- Basic synchronization
- http://www.cs.utexas.edu/~rossbach/cs378/lab/lab0.html

- ***Start early!!!***

Questions?