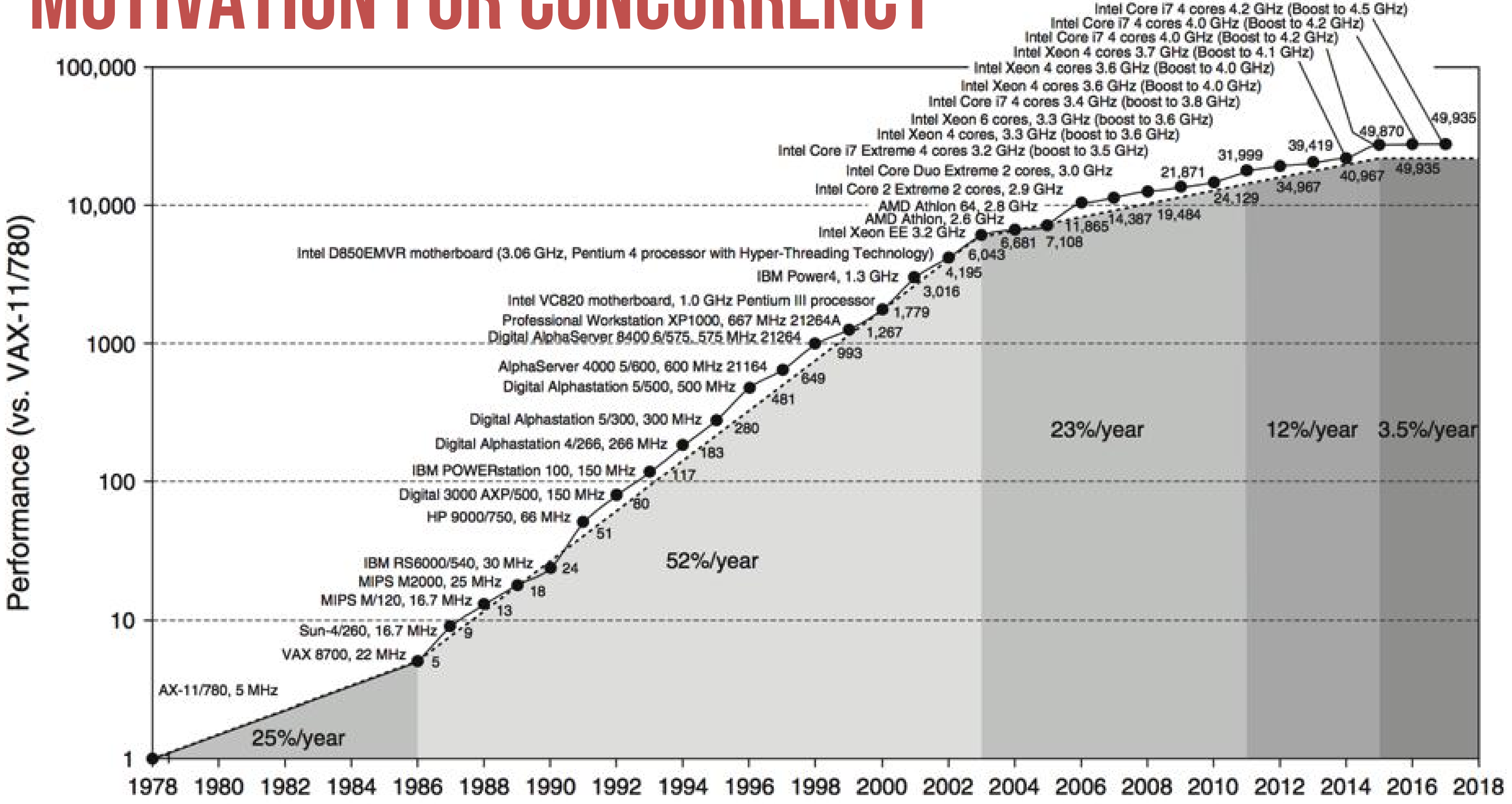
INTRODUCTION TO CONCURRENCY

Kai Mast
CS 537
Fall 2022

MOTIVATION FOR CONCURRENCY



MOTIVATION

CPU Trend:

- Little improvement in clock speed, but massive improvements the number of transistors per chip
- Achieve performance improvement using multiple cores

Option 0: Run many different (competing) applications on one machine

  - OS scheduler ensures cores are used efficiently

Our goal instead: Write single applications that fully utilize many cores

  - Improve performance of single application

OPTION 1: USE MULTIPLE PROCESSES

Option 1: Build apps from many communicating **processes**

- Example: Most modern browsers (dedicated process per tab)
- Communicate via `pipe()` or other message passing primitives

Pros?

- Don't need new abstractions; good for security and fault-tolerance

Cons?

- Cumbersome programming; all data sharing has to be explicit
- High communication overheads (usually copy data)
- Expensive context switching (why expensive?)

OPTION 2: DIVIDE PROCESS INTO THREADS

New abstraction: Threads

Threads are like processes, except:

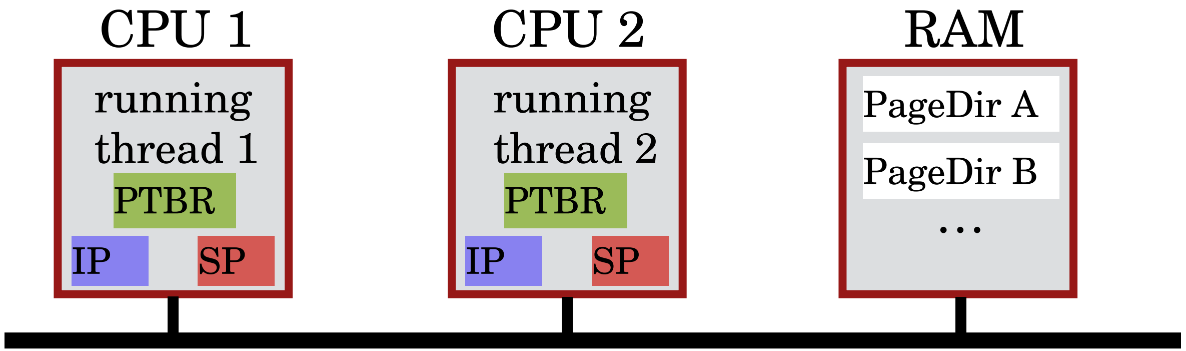
multiple threads of same process share an address space

Divide large task across several cooperative threads
Communicate through shared address space

What state do threads share?

Do threads share code?

Do threads share Instruction Pointers?



Share code, but each thread may be executing different code at the same time
=> Different Instruction Pointers

Do threads share stack and/or stack pointer?

Threads executing different functions need different stacks
(But, stacks are in same address space, so trusted to be cooperative)

Thread vs. Process

Multiple threads within a single process share:

- Process ID (PID)
- Address space: Code (instructions), Most data (heap)
- Open file descriptors
- Current working directory (environment variables)
- User and group id (permissions, limits)

Each thread has its own

- Thread ID (TID)
- Set of registers, including Program counter and Stack pointer
- Stack for local variables and return addresses
  - There are now multiple stacks in same address space

Thread API

Variety of thread systems exist

- e.g., POSIX pthreads

Common thread operations

- `create` (specify function to start at)
  - How is this different than fork()?
- `exit`
- `join` (instead of `wait()` for processes)

COMMON PROGRAMMING MODELS

Multi-threaded programs might follow one of these patterns:

Producer/consumer: Multiple producer threads create data (or work) handled by one of multiple consumer threads

Pipeline: Task is divided into series of sub-tasks, each handled in series by a different thread

Defer work with background thread: One thread performs non-critical work in the background (e.g., when CPU idle)

OS SUPPORT: NONE

User-level threads: Many-to-one thread mapping

- Thread operations implemented by user-level runtime libraries
- OS is not aware of user-level threads

  OS thinks each process contains only a single thread of control

Advantages

- Does not require OS support; Portable
- Can tune scheduling policy to meet application demands (simplify)
- Lower overhead thread operations since no system call

Disadvantages?

- Cannot use multiprocessors (i.e., more than one CPU core)
- Entire process blocks when one thread blocks

OS Support: Kernel Threads

Kernel-level threads: One-to-one thread mapping

- OS provides each user-level thread with a kernel thread
- Each kernel thread is scheduled independently by OS
- Thread operations (creation, scheduling, synchronization) performed by OS

Advantages

- Each kernel-level thread can run in parallel on a multiprocessor
- When one thread blocks, other threads from process can still be scheduled

Disadvantages

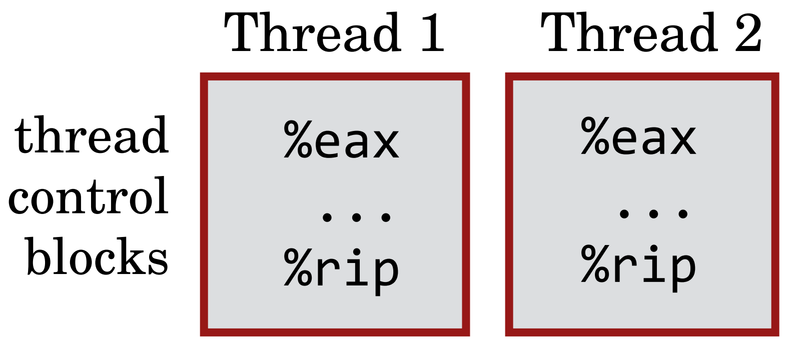
- Higher overhead for thread operations
- OS must scale well with increasing number of threads

EXAMPLE: CONCURRENT INCREMENT

Data
```
balance at 0x9cd4 = 100
```

Code (C)
```
balance = balance + 1;
```

Code (Assembly)
```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```



Registers are virtualized by OS;
Each thread "thinks" it has own

- Both threads run the above code
- What possible **execution timelines** could there be?

POSSIBLE TIMELINE #1

Thread 1	Thread 2	Balance (@0x9cd4)	EAX (T1)	EAX (T2)
	`mov 0x9cd4, %eax`			
	`add %0x1, %eax`			
`mov 0x9cd4, %eax`				
`add %0x1, %eax`				
`mov %eax, 0x123`				
	`mov %eax, 0x9cd4`			

How much is added?

POSSIBLE TIMELINE #2

Thread 1	Thread 2	Balance	EAX (T1)	EAX (T2)
`mov 0x9cd4, %eax`				
`add %0x1, %eax`				
	`mov 0x9cd4, %eax`			
`mov %eax, 0x9cd4`				
	`add %0x1, %eax`			
	`mov %eax, 0x9cd4`			

How much is added?

POSSIBLE TIMELINE #3

Thread 1	Thread 2	Balance	EAX (T1)	EAX (T2)
`mov 0x9cd4, %eax`				
`add %0x1, %eax`				
`mov %eax, 0x9cd4`				
	`mov 0x9cd4, %eax`			
	`add %0x1, %eax`			
	`mov %eax, 0x9cd4`			

How much is added?

NON-DETERMINISM

Concurrency leads to non-deterministic results

- with same inputs

Race conditions: Specific type of bug

-
- Whether bug manifests depends on CPU schedule!
- Even very unlikely events will occur eventually if repeated billions of times…

How to write code safe of race conditions?

WHAT DO WE WANT?

Want 3 instructions to execute as an uninterruptible group

Want them to be **atomic**:

```
mov 0x9cd4, %eax
add %0x1, %eax
mov %eax, 0x9cd4
```

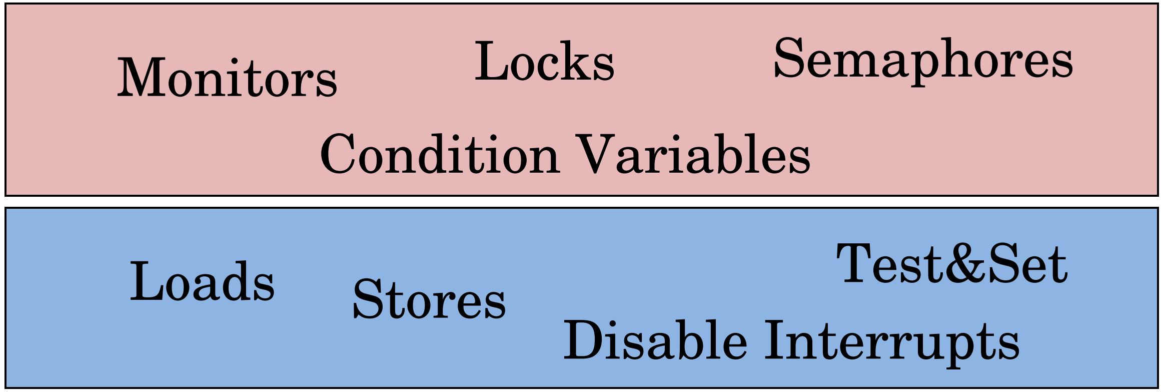
More general: Need **mutual exclusion** for **critical sections** of code

(okay if other threads do unrelated work)

SYNCHRONIZATION

- Build higher-level synchronization primitives in OS
- Operations that ensure correct ordering of instructions across threads
- Use help from hardware

Motivation: Build them once and get them right



LOCKS

Goal: Provide mutual exclusion (Mutex)

Allocate and Initialize

- `pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;`

Acquire

- Acquire exclusion access to lock;
- Wait if lock is not available (some other process in critical section)
- Spin or block (relinquish CPU) while waiting
- `pthread_mutex_lock(&mylock);`

Release

- Release exclusive access to lock; let another process enter critical section
- `pthread_mutex_unlock(&mylock);`

CONCURRENCY SUMMARY

Concurrency is needed for high performance when using multiple cores

Threads

- Multiple execution streams within a single process or address space
  - Share PID and address space
  - Each has their own registers and stack

- Context switches (or running thread on different core simultaneously) within a critical section can lead to non-deterministic bugs

- Synchronization is needed to prevent bugs and unexpected behavior

VIRTUALIZATION REVIEW

QUESTION 1: SCHEDULING

Assume we have three jobs that enter a system and need to be scheduled. The first job that enters is called A, and it needs 10 seconds of CPU time. The second, which arrives just after A, is called B, and it needs 15 seconds of CPU time. The third, C, arrives just after B, and needs 10 seconds of CPU time.

Question 2 from Fall 08 Midterm

QUESTION 1: SCHEDULING, PART 1

Assume we have three jobs that enter a system and need to be scheduled. The first job that enters is called A, and it needs 10 seconds of CPU time. The second, which arrives just after A, is called B, and it needs 15 seconds of CPU time. The third, C, arrives just after B, and needs 10 seconds of CPU time.

Assuming a **shortest-job-first (SJF)** policy, **at what time does B finish?**

Answer

- Order of completion: A, C, B
- B finishes last, after 35 seconds

QUESTION 1: SCHEDULING, PART 2

Assume we have three jobs that enter a system and need to be scheduled. The first job that enters is called A, and it needs 10 seconds of CPU time. The second, which arrives just after A, is called B, and it needs 15 seconds of CPU time. The third, C, arrives just after B, and needs 10 seconds of CPU time.

Assume a **round-robin policy** (with a time slice length of 1 second).

When does **job A finish**? When does **job B finish**?
Assume there is no overhead due to context switching.

Answers

- A finishes first at T=28
- B finishes last at T=35

QUESTION 1: SCHEDULING, PART 2

Time		2		4		6		8		10		12		14		16		18		20		22		24		26		28		30		32		34		36
Job A	S			🟪			🟪			🟪			🟪			🟪			🟪			🟪			🟪			E								
Job B		🟪			🟪			🟪			🟪			🟪			🟪			🟪			🟪			🟪			🟪		🟪	🟪	🟪	🟪	E	
Job C			🟪			🟪			🟪			🟪			🟪			🟪			🟪			🟪			🟪			E						

Job	**Arrival**	**Duration**	**First Time Scheduled**	**Finish Time**
A	0	10	0	28
B	0 (after A)	15	1	35
C	0 (after B)	10	2	30

QUESTION 1: SCHEDULING, PART 3

Of course, SJF is unrealistic, because usually the OS doesn't know how long jobs are. In this system, though, the **user gives the OS an estimate**. The problem is that the users aren't so good at estimation. In fact, if they tell you a job will last N seconds, it **might last anywhere between N – 5 and N + 5 second**s. But, being a nice, trusting OS, the OS assumes the user is exactly right.

Assuming SJF, what estimates (by the user) will lead the OS to make the worst decisions for these jobs in terms of achieving the **lowest average response time**?

Job	**Arrival**	**Duration**
A	0	10
B	0 (after A)	15
C	0 (after B)	10

Answer: Anything that makes B run first

QUESTION 1: SCHEDULING, PART 3

In this case (B runs first, no preemption), what would the **average response time** for the jobs be?

What would the average **turnaround time be**?

Job	**Arrival**	**Duration**	**Start**	**End**	**Response Time**	**Turnaround Time**
A	0	10	15	25	15	25
B	0 (after A)	15	0	15	0	15
C	0 (after B)	10	25	35	25	35

Avg. Response Time: (15+0+25)/3 = 13 1/3
Avg. Turnaround Time: (25+15+35)/3 = 25

QUESTION 2: SEGMENTATION, PART 1

Basics

What is the base register for? *Beginning of the segment in physical memory*

What is the bounds register for?
Size of the segment (needed for fault-tolerance and security)

What type of address is in the base register (physical or virtual)?
Physical (points to the location of the segment in phy. memory)

How many segments should the hardware support? (Why?)
At least two (one for code+heap and one for the stack)

Question 2 from Spring 09 Midterm

QUESTION 2: SEGMENTATION, PART 2

There are two segments supported by the hardware.

Address spaces are small (1KB), and the amount of physical memory on the system is 16KB. Assume that the segment-0 base register has the value 1KB, and its bounds (size) is set to 300 bytes; this segment grows upward.

Assume the segment 1-base register has the value 5KB in it, and its bound is also 300; this segment grows downward (the negative direction).

Assume the following program

```
void *ptr = 20;
while (ptr <= 1024) {
    int x = (int *) *ptr; // LINE 1: read what is at address ’ptr’
    ptr = ptr + 20;
    // LINE 2: increment ’ptr’ to a new address
}
```

QUESTION 2: SEGMENTATION, PART 2

Virtual AS Size: 1024

```
void *ptr = 20;
while (ptr <= 1024) {
    int x = (int *) *ptr;
    ptr = ptr + 20;
}
```

Seg.	Base	Bounds	Grows
0	1024	300	Positive
1	5120	300	Negative

What are the programs (virtual) addresses generated at **line 3**?
Starts at 20; increments by 20 until 1020

How long will this program run before crashing (due to a segmentation violation)?
14 iterations. Access to 300 exceeds the bounds of segment 0 and thus will cause an exception

What physical addresses will be generated by dereferencing ptr before the program crashes?
1324 (1024+300) will cause crash

QUESTION 2: SEGMENTATION, PART 3

Virtual AS Size: 1024

```
void *ptr = 20;
while (ptr <= 1024) {
    int x = (int *) *ptr;
    ptr = ptr + 20;
}
```

Seg.	Base	Bounds	Grows
0	1024	300	Positive
1	5120	300	Negative

What legal physical addresses could have been generated by dereferencing ptr, if the programmer had been more careful to avoid crashing?
(if we skipped invalid addresses)

The last 300 bytes of the AS are also legal (below 1024 starting at 724)

The first legal address in this range would be 740 which translates to 4836. Thus, every virtual address between 4836 and 5116 (inclusive) could have been legally accessed.

THAT'S ALL

- **Reminder:** Send me more questions you want to talk about by Monday
- Feel free to schedule office hours (via email) to talk over other questions