CONCURRENCY: INTRODUCTION

Shivaram Venkataraman

CS 537, Fall 2024

ADMINISTRIVIA

- Project 3 done?!

- Code review: Sign up?

- Midterm 1 details: Piazza

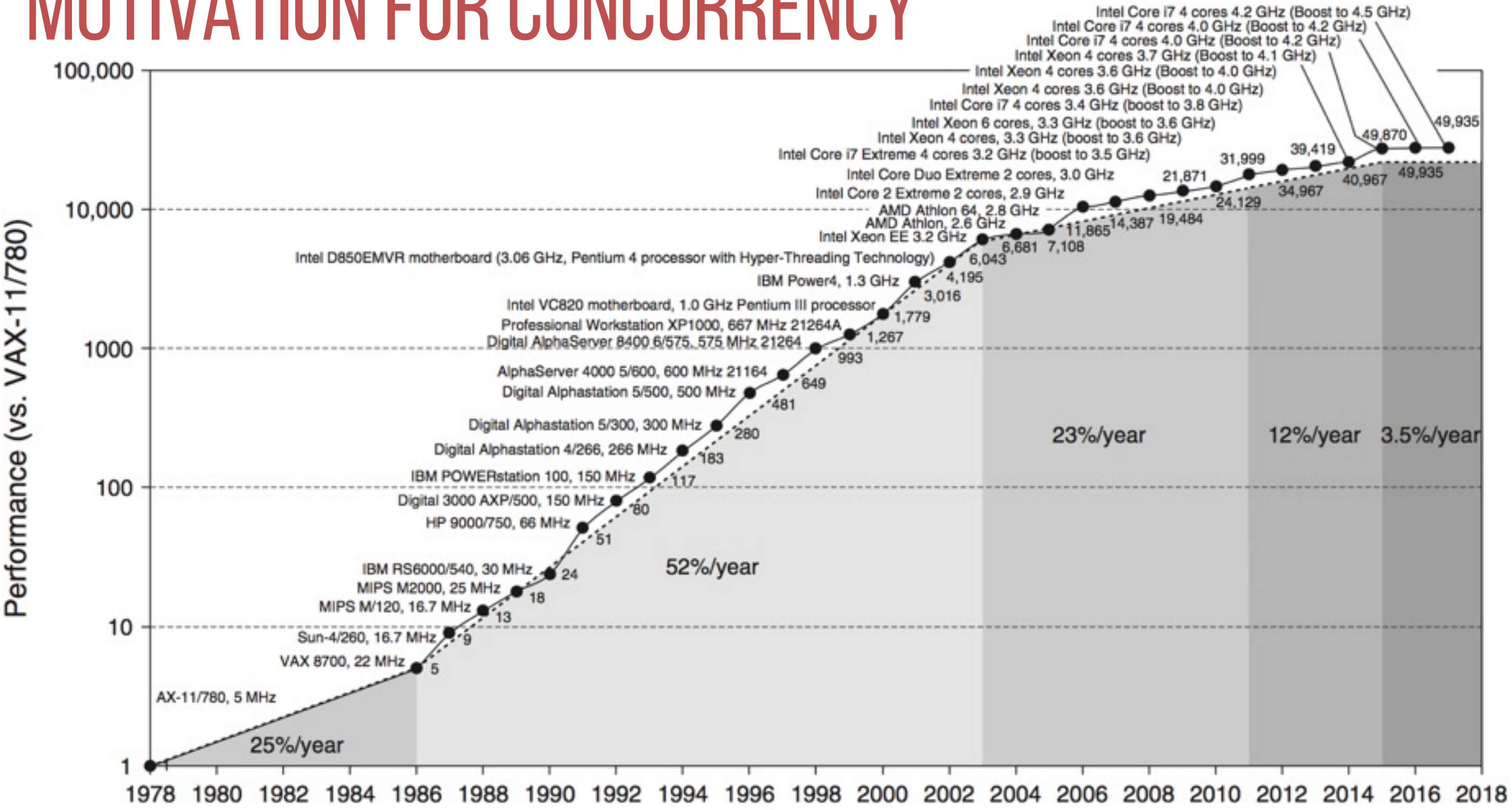
AGENDA / LEARNING OUTCOMES

Concurrency

- What is the motivation for concurrent execution?
- What are some of the challenges?

CONCURRENCY

MOTIVATION FOR CONCURRENCY



MOTIVATION

CPU Trend: Same speed, but multiple cores

Goal: Write applications that fully utilize many cores

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

- Example: Chrome (process per tab)
- Communicate via pipe() or similar

Pros?

- Don't need new abstractions; good for security

Cons?

- Cumbersome programming
- High communication overheads
- Expensive context switching (why expensive?)

CONCURRENCY: OPTION 2

New abstraction: thread

Threads are like processes, except:

multiple threads of same process share an address space

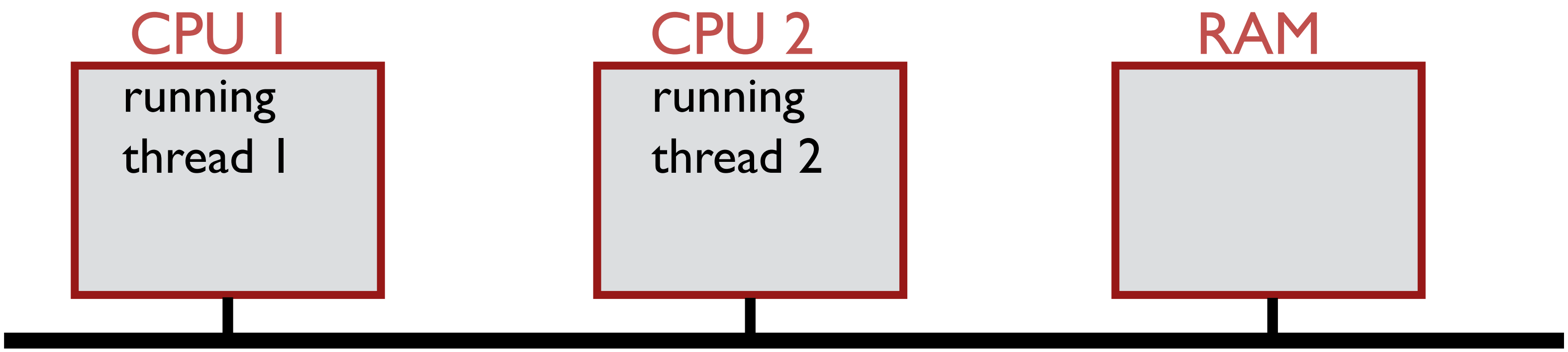
Divide large task across several cooperative threads

Communicate through shared address space

COMMON PROGRAMMING MODELS

Multi-threaded programs tend to be structured as:

- **Producer/consumer**
  Multiple producer threads create data (or work) that is handled by one of the multiple consumer threads
- **Pipeline**
  Task is divided into series of subtasks, each of which is handled in series by a different thread
- **Defer work with background thread**
  One thread performs non-critical work in the background (when CPU idle)



What state do threads share?

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
- User and group id

Each thread has its own

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

OS SUPPORT: APPROACH 1

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

- Implemented by user-level runtime libraries

  Create, schedule, synchronize threads at user-level

- 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
- Lower overhead thread operations since no system call

Disadvantages?

- Cannot leverage multiprocessors
- Entire process blocks when one thread blocks

OS SUPPORT: APPROACH 2

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

- OS provides each user-level thread with a kernel thread
- Each kernel thread scheduled independently
- 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 be scheduled

Disadvantages

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

THREAD SCHEDULE

```
volatile int balance = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
      balance++;
    }
    pthread_exit(NULL);
}
```

```
int main(int argc, char *argv[]) {
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", balance);
    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", balance);
    return 0;
}
```

```
» ./threads 100000
Initial value : 0
Final value   : 162901
```

THREAD SCHEDULE #1

```
balance = balance + 1;
balance at 0x9000
```

State:

```
0x9000: 100
%eax:
%rip = 0x195
```

```
0x195  mov 0x9000, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9000
```

thread control blocks:

	Thread 1	Thread 2
	%eax:	%eax:
	%rip:	%rip:

THREAD SCHEDULE #2

```
balance = balance + 1;
balance at 0x9cd4
```

State:

```
0x9000: 100
%eax:
%rip = 0x195
```

```
0x195  mov 0x9000, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9000
```

thread control blocks:

	Thread 1	Thread 2
%eax:		
%rip:		

TIMELINE VIEW

Thread 1

mov 0x123, %eax

add %0x1, %eax

mov %eax, 0x123

Thread 2

mov 0x123, %eax

add %0x2, %eax

mov %eax, 0x123

QUIZ 9

https://tinyurl.com/cs537-fa24-q9

Process A with threads TA1 and TA2 and process B with a thread TB1.

1. With respect to TA1 and TA2 which of the following are true?

2. Which of the following are true with respect to TA1 and TB1?

Thread 1	Thread 2
mov 0x123, %eax	
add %0x1, %eax	
	mov 0x123, %eax
mov %eax, 0x123	
	add %0x2, %eax
	mov %eax, 0x123

Thread 1	Thread 2
	mov 0x123, %eax
mov 0x123, %eax	
	add %0x2, %eax
add %0x1, %eax	
	mov %eax, 0x123
mov %eax, 0x123	

Thread 1	Thread 2
	mov 0x123, %eax
	add %0x2, %eax
	mov %eax, 0x123
mov 0x123, %eax	
add %0x1, %eax	
mov %eax, 0x123	

NON-DETERMINISM

Concurrency leads to non-deterministic results

- Different results even with same inputs
- race conditions

Whether bug manifests depends on CPU schedule!

How to program: imagine scheduler is malicious?!

WHAT DO WE WANT?

Want 3 instructions to execute as an uninterruptable group

That is, we want them to be atomic

```
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123
```

More general: Need mutual exclusion for critical sections

if thread A is in critical section C, thread B isn't

(okay if other threads do unrelated work)

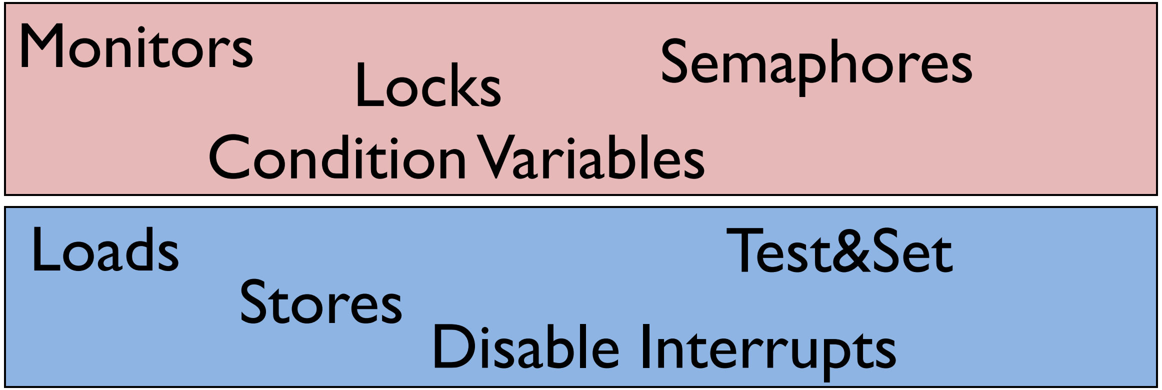
SYNCHRONIZATION

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

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);

LOCK IMPLEMENTATION GOALS

Correctness

- *Mutual exclusion*

  Only one thread in critical section at a time
- *Progress* (deadlock-free)

  If several simultaneous requests, must allow one to proceed
- *Bounded* (starvation-free)

  Must eventually allow each waiting thread to enter

Fairness: Each thread waits for same amount of time

Performance: CPU is not used unnecessarily

IMPLEMENTING SYNCHRONIZATION

Atomic operation: No other instructions can be interleaved

Approaches

- Disable interrupts
- Locks using loads/stores
- Using special hardware instructions

IMPLEMENTING LOCKS: W/ INTERRUPTS

Turn off interrupts for critical sections

- Prevent dispatcher from running another thread
- Code between interrupts executes atomically

```
void acquire(lockT *l) {
    disableInterrupts();
}
```

```
void release(lockT *l) {
    enableInterrupts();
}
```

Disadvantages?

Only works on uniprocessors

Process can keep control of CPU for arbitrary length

Cannot perform other necessary work

IMPLEMENTING LOCKS: W/ LOAD+STORE

Code uses a single **shared** lock variable

```
// shared variable
boolean lock = false;
void acquire(Boolean *lock) {
    while (*lock) /* wait */ ;
    *lock = true;
}
```

```
void release(Boolean *lock) {
    *lock = false;
}
```

Does this work? What situation can cause this to not work?

RACE CONDITION WITH LOAD AND STORE

`*lock == 0 initially`

Thread 1	Thread 2
`while(*lock == 1)`	
	`while(*lock == 1)`
	`*lock = 1`
`*lock = 1`	

Both threads grab lock!

Problem: Testing lock and setting lock are not atomic

XCHG: ATOMIC EXCHANGE OR TEST-AND-SET

How do we solve this ? Get help from the hardware!

```
// xchg(int *addr, int newval)
// return what was pointed to by addr
// at the same time, store newval into addr
int xchg(int *addr, int newval) {
  int old = *addr;
  *addr = newval;
  return old;
}
```

```
movl 4(%esp), %edx
movl 8(%esp), %eax
xchgl (%edx), %eax
ret
```

LOCK IMPLEMENTATION WITH XCHG

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    lock->flag = ??;
}

void acquire(lock_t *lock) {
    ????;
    // spin-wait (do nothing)
}

void release(lock_t *lock) {
    lock->flag = ??;
}
```

```
int xchg(int *addr, int newval)
```

OTHER ATOMIC HW INSTRUCTIONS

```
int CompareAndSwap(int *addr, int expected, int new) {
  int actual = *addr;
  if (actual == expected)
    *addr = new;
  return actual;
}
```

```
void acquire(lock_t *lock) {
    while(CompareAndSwap(&lock->flag,  ,  ) ==  ) ;
    // spin-wait (do nothing)
}
```

NEXT STEPS

Midterm 1: Next week

Next class: More about locks!