CS 423
Operating System Design: Concurrency: Locks
Mar 03

Ram Kesavan

Slide ack: Prof. Shivaram Venkataraman (Wisconsin)

ANNOUNCEMENTS

MP2 out; Due (3/23) 11:59 pm CT

MP1 grades are out

20% students: have been informed via email today

4Cr: next paper on “Mesa Monitors”, due 3/10

EXPECTATION: That students read OSTEP chapters

AGENDA / LEARNING OUTCOMES

Go through pop-quiz answers

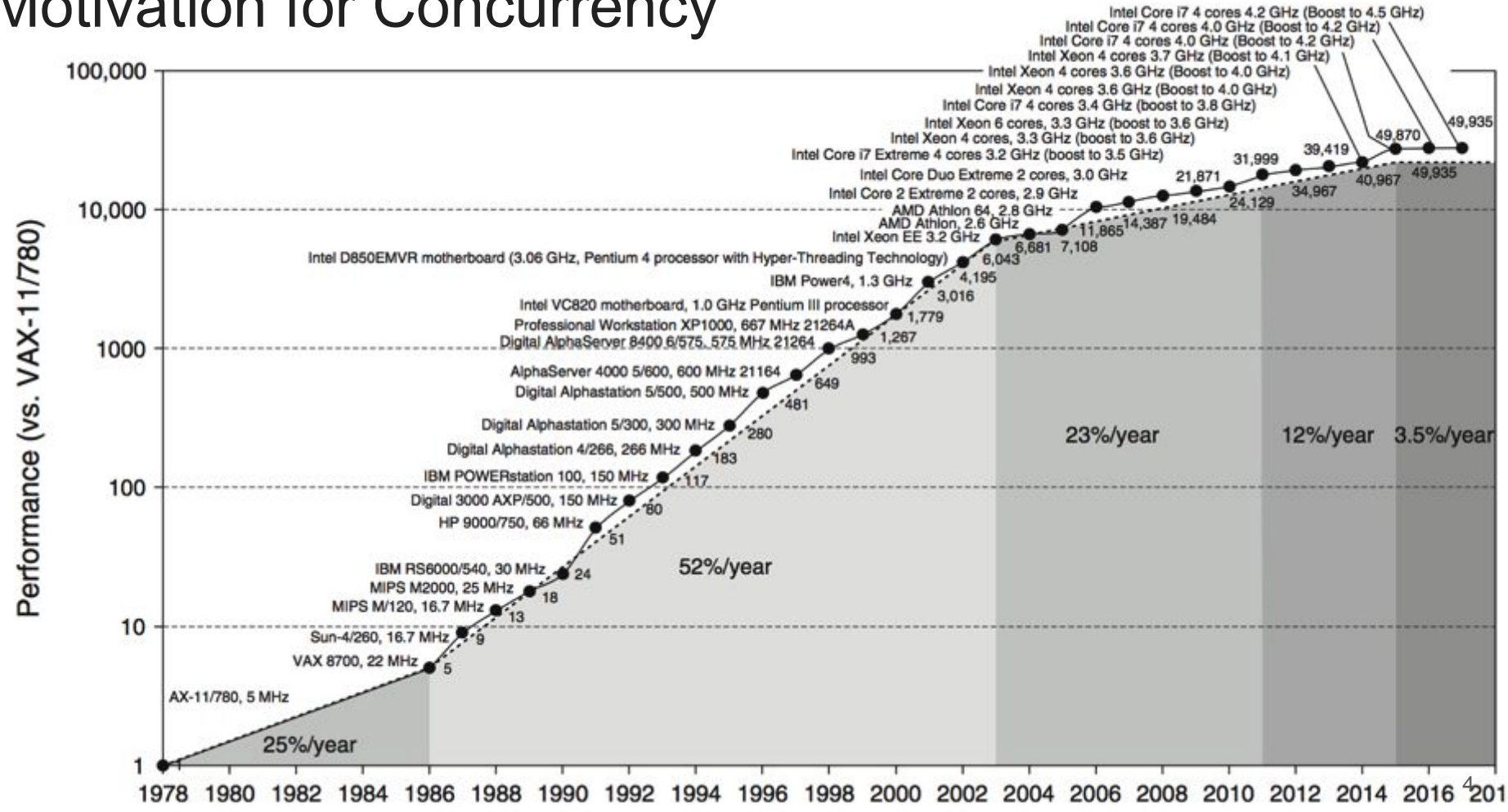
Concurrency:

- What is the motivation for concurrent execution
  - Challenges?
- Threading & Locks

For folks that have taken 341 (or its equivalent)

- Some of this will be obvious; but we'll dig deeper

Motivation for Concurrency



MOTIVATION

CPU Trend: Speed has stabilized; increasing (maybe differentiated) #cores

Goal: Application utilize more cores

Option 1: Build app with many **processes**

- Example: Browser (process per tab)

Pros:

- Don't need new abstractions
- Good for security

Cons:

- High communication overhead
- Expensive context switching

Option 2: Build app with many **threads**

- Example: Word processors, video games

Pros:

- Shared address space
- Cheaper: fast context switch

Cons:

- Data corruption

Many modern apps are hybrid: use both options

THREADING

Threads (of same process) share address space

Each thread has its “own” stack

“own” -> but not really protected from other threads

Question: Walk through eg. of a bug where a thread corrupts another thread’s stack.

THREADING

New abstraction: thread

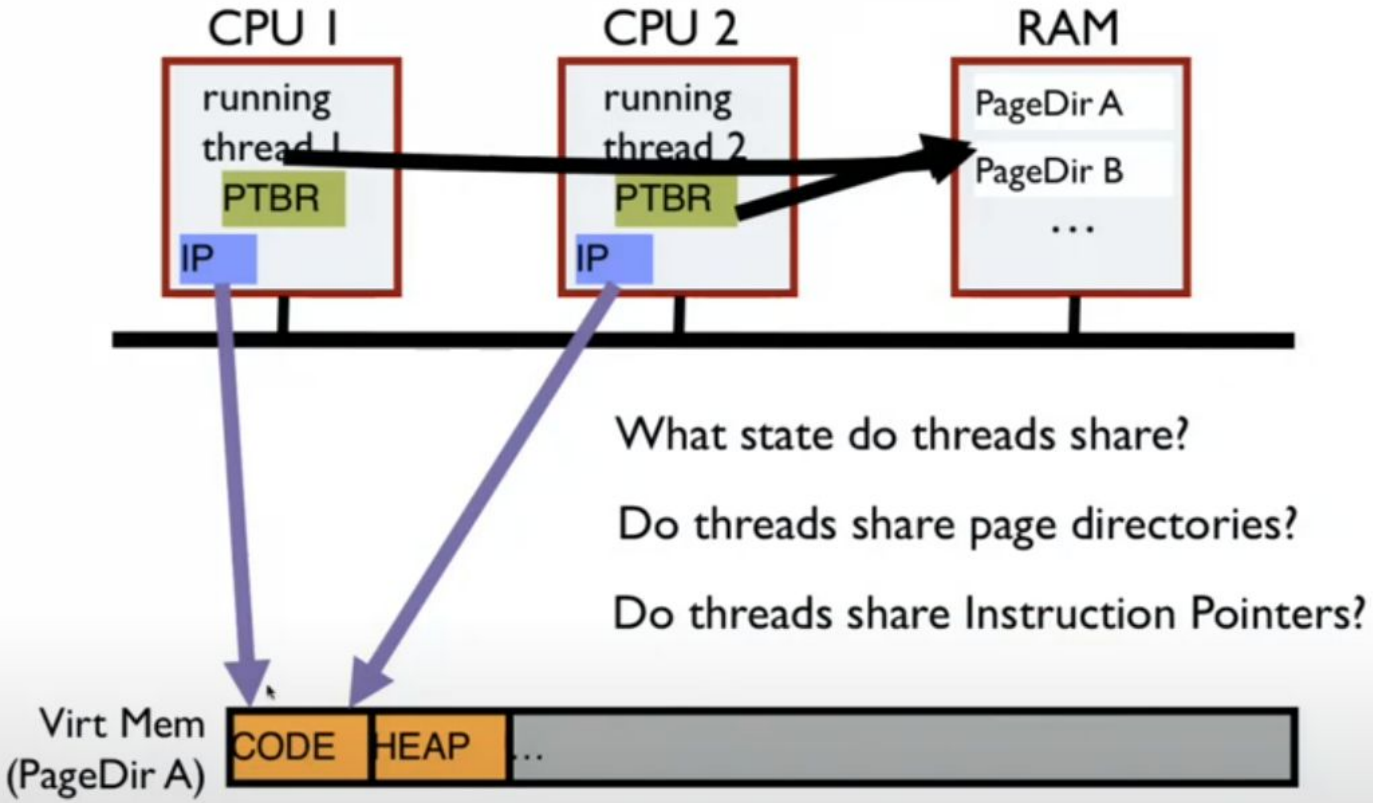
Threads are like “lightweight processes”, except:

- **Multiple threads (of same process) share the address space**
- **Each thread has its “own” stack**
  - “own” -> but not really protected from other threads

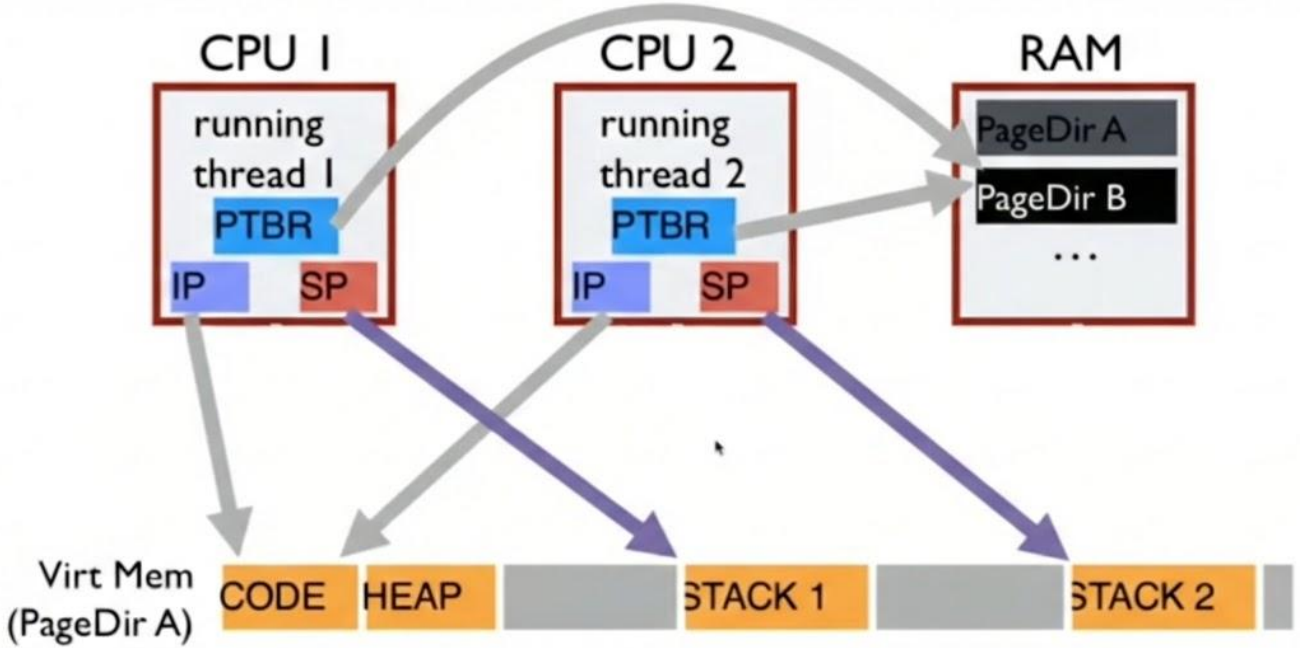
Inter-thread communication is cheaper (than inter-process communication)

- Don’t need to send the actual data because the heap is shared

In this course: POSIX thread library, aka **pthreads**



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



Do threads share stack pointer?

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

THREADS WITHIN A PROCESS

All threads **share**:

- Process ID (PID)
- Page Table (PTBR), Address space: code (instructions), data (heap)
- Open file descriptors
- Current working directory
- User and group id

Each thread has its own **unique**:

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

Threading: Approach 1

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

- Implemented by user-level runtime libraries
- OS unaware of the existence of user-level threads
- OS believes each process contains only a single thread of control

Pros:

- No OS requirements; Completely portable
- Can tune policies to suit application
- Lower overhead switching; no system call to switch threads

Cons:

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

Threading: Approach 2

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

- User-level runtime lib, but each user-level thread gets a kernel thread
- Each kernel thread schedulable independently
- OS performs thread operations (create, schedule, synchronize)

Pros:

- Multiple threads can run in parallel
- When a thread blocks, other threads are unaffected

Cons:

- Higher overhead for thread operations (compared to Approach 1)

THREAD SCHEDULE

```
volatile int balance = 0;
int loops = 1000000; // 1 million

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

```
int main(int argc, char *argv[]) {
    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;
}
```

EXPECTATION: Students will teach themselves pthreads

THREAD SCHEDULE #1

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

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

State:
0x9cd4: 100
%eax: ?
%rip = 0x195

process control blocks:

Thread 1	Thread 2
%eax: ? %rip: 0x195	%eax: ? %rip: 0x195


```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

THREAD SCHEDULE #1

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2

process control blocks:

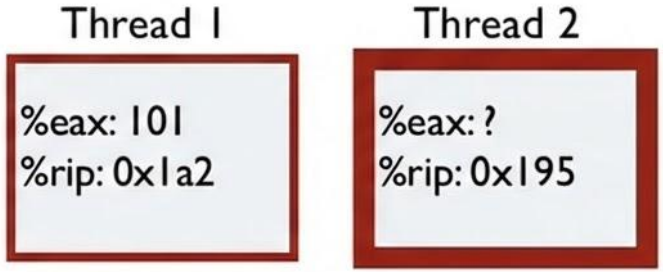

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

T1

THREAD SCHEDULE #1

State:
0x9cd4: 101
%eax: ?
%rip = 0x195

process control blocks:

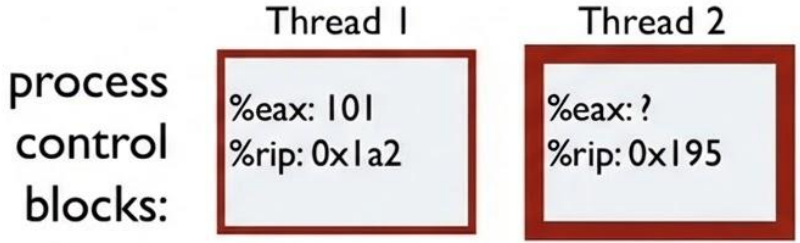


T2

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

THREAD SCHEDULE #1

State:
0x9cd4: 102
%eax: 102
%rip = 0x1a2




```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

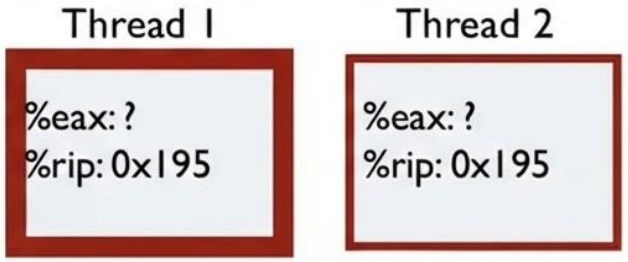
T2 ➡

Desired Result!

THREAD SCHEDULE #2

State:
0x9cd4: 100
%eax: 101
%rip = 0x19d

process control blocks:



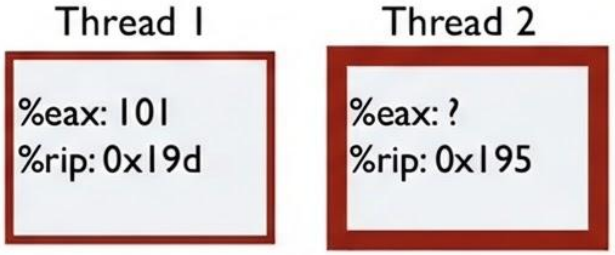

```
        0x195   mov 0x9cd4, %eax
        0x19a   add $0x1, %eax
T1      0x19d   mov %eax, 0x9cd4
```

Thread Context Switch before T1 executes 0x19d

THREAD SCHEDULE #2

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2

process control blocks:

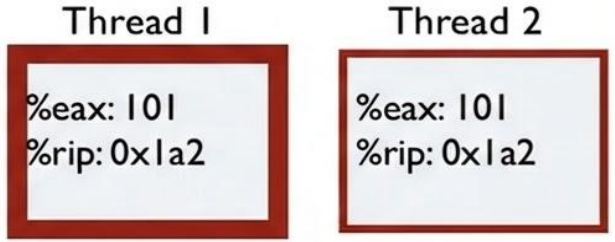



```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

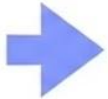
T2 

THREAD SCHEDULE #2

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2




```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

T1 

WRONG Result! Final value of balance is 101

TIMELINE VIEW

Thread 1

mov 0x123, %eax

add $0x1, %eax

mov %eax, 0x123

Thread 2

mov 0x123, %eax

add $0x2, %eax

mov %eax, 0x123

TIMELINE VIEW

Thread 1	Thread 2	Balance	T1 %eax	T2 %eax
mov 0x123, %eax		0	0	
add $0x1, %eax		0	1	
mov %eax, 0x123		1	1	
	mov 0x123, %eax	1		1
	add $0x2, %eax			3
	mov %eax, 0x123	3		3

3 gets added to balance => correct behavior

TIMELINE VIEW

Thread 1	Thread 2	Balance	T1 %eax	T2 %eax
mov 0x123, %eax		0	0	
add $0x1, %eax		0	1	
	mov 0x123, %eax	0	1	
mov %eax, 0x123		1	1	0
	add $0x2, %eax			2
	mov %eax, 0x123	2		2

2 gets added to balance => incorrect behavior

2 MORE EXAMPLES

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

What happens in each case? Assume balance = 0 at the start.

NON-DETERMINISM

Concurrency leads to non-deterministic results

- Different results even with same inputs
- Race conditions

Whether a bug manifests depends on the CPU schedule.

How to program: imagine scheduler is behaving maliciously.

General principle: Assume the worst-case schedule!

WHAT DO WE WANT?

Want 3 instructions to execute as an un-interruptable group

That is, we want them to be **atomic**

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

More generally: need mutual exclusion for critical sections

If thread A is in the critical section, all other threads cannot
(okay if other threads do unrelated work)

Common Programming Models

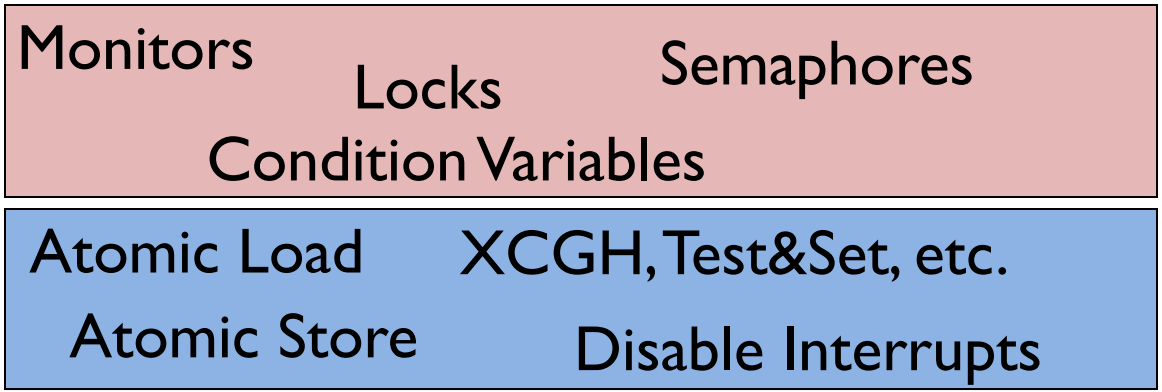
Typical programming models for threaded programs:

- **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
- **Deferred work done in the background**
  One thread performs non-critical work in the background (when CPU idle)

Synchronization

Generic synchronization: for all models from previous slides

Goal: Learn how synchronization primitives are built (lib + OS)
With a little help from our friend…the hardware



Next Class

LOCKS