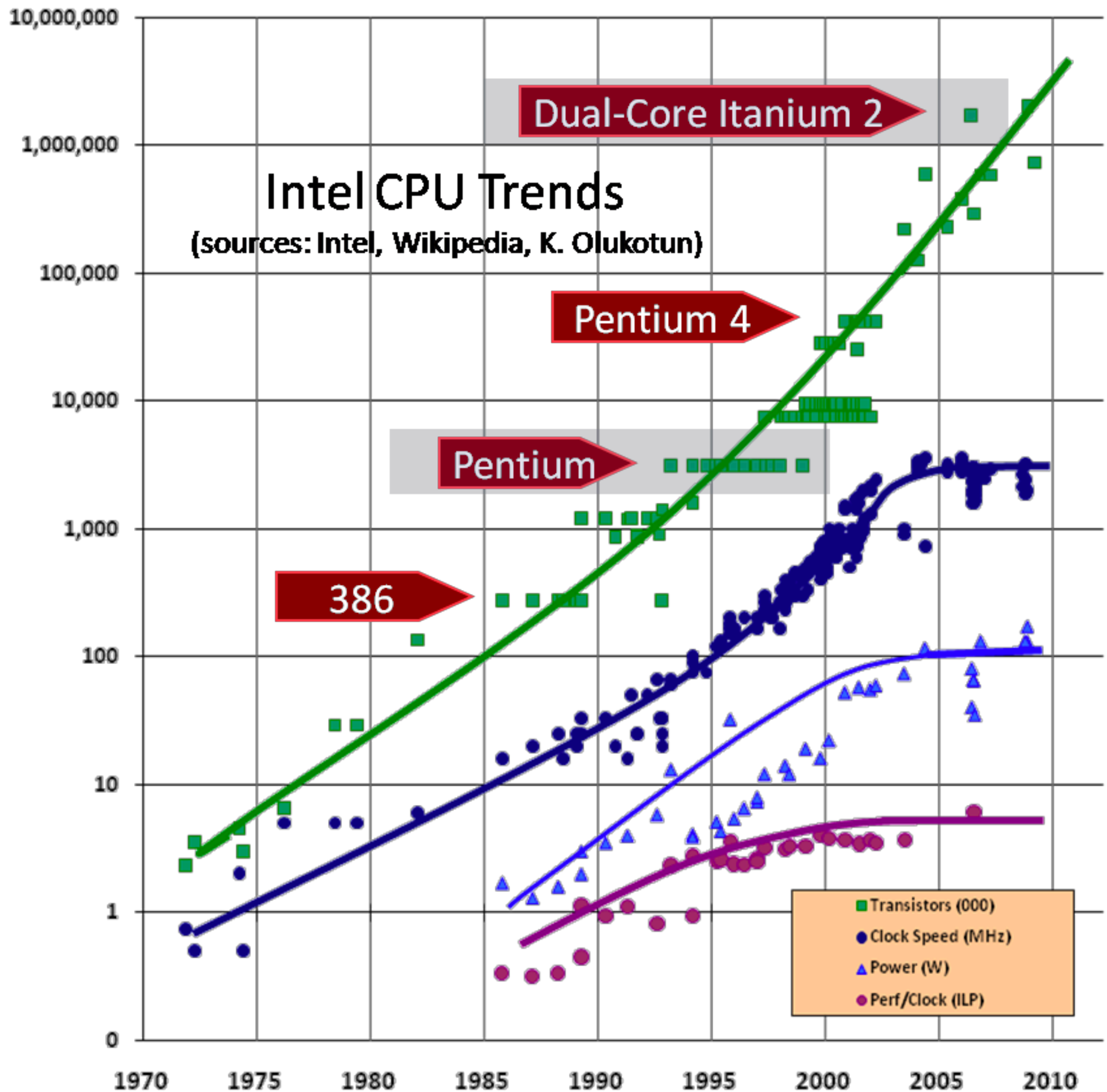
Operating Systems

Lecture 18: Concurrency - Locks

Nipun Batra

Sep 25, 2018



Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)
Dual-Core Itanium 2
Pentium 4
Pentium
386
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
0
1970
1975
1980
1985
1990
1995
2000
2005
2010
Transistors (000)
Clock Speed (MHz)
Power (W)
Perf/Clock (ILP)

Hyper threading v/s Multi-core

Hyper threading v/s Multi-core

1. Hyper-threading: Coordination between arm and mouth to increase speed of eating!

Hyper threading v/s Multi-core

1. Hyper-threading: Coordination between arm and mouth to increase speed of eating!
2. Multi-core: Having more than one mouth to eat!

Strategy 1

Strategy 1

1. Build applications from many communicating processes

Strategy 1

1. Build applications from many communicating processes
    1. Eg - Google Chrome?!

Strategy 1

1. Build applications from many communicating processes
    1. Eg - Google Chrome?!
2. Communicate using IPC:

Strategy 1

1. Build applications from many communicating processes
    1. Eg - Google Chrome?!
2. Communicate using IPC:
    1. files?!

Strategy 1

1. Build applications from many communicating processes
    1. Eg - Google Chrome?!
2. Communicate using IPC:
    1. files?!
    2. Pipes

Strategy 1

1. Build applications from many communicating processes
    1. Eg - Google Chrome?!
2. Communicate using IPC:
    1. files?!
    2. Pipes
    3. …

Strategy 1

1. Build applications from many communicating processes
    1. Eg - Google Chrome?!
2. Communicate using IPC:
    1. files?!
    2. Pipes
    3. …
3. Multiple copies of address space!

Strategy 2

Strategy 2

1. Use Threads : just like processes, but, share the address space (i.e. PT)

Strategy 2

1. Use Threads : just like processes, but, share the address space (i.e. PT)
2. Do not need to communicate using IPC:

Strategy 2

1. Use Threads : just like processes, but, share the address space (i.e. PT)
2. Do not need to communicate using IPC:
    1. Shared data

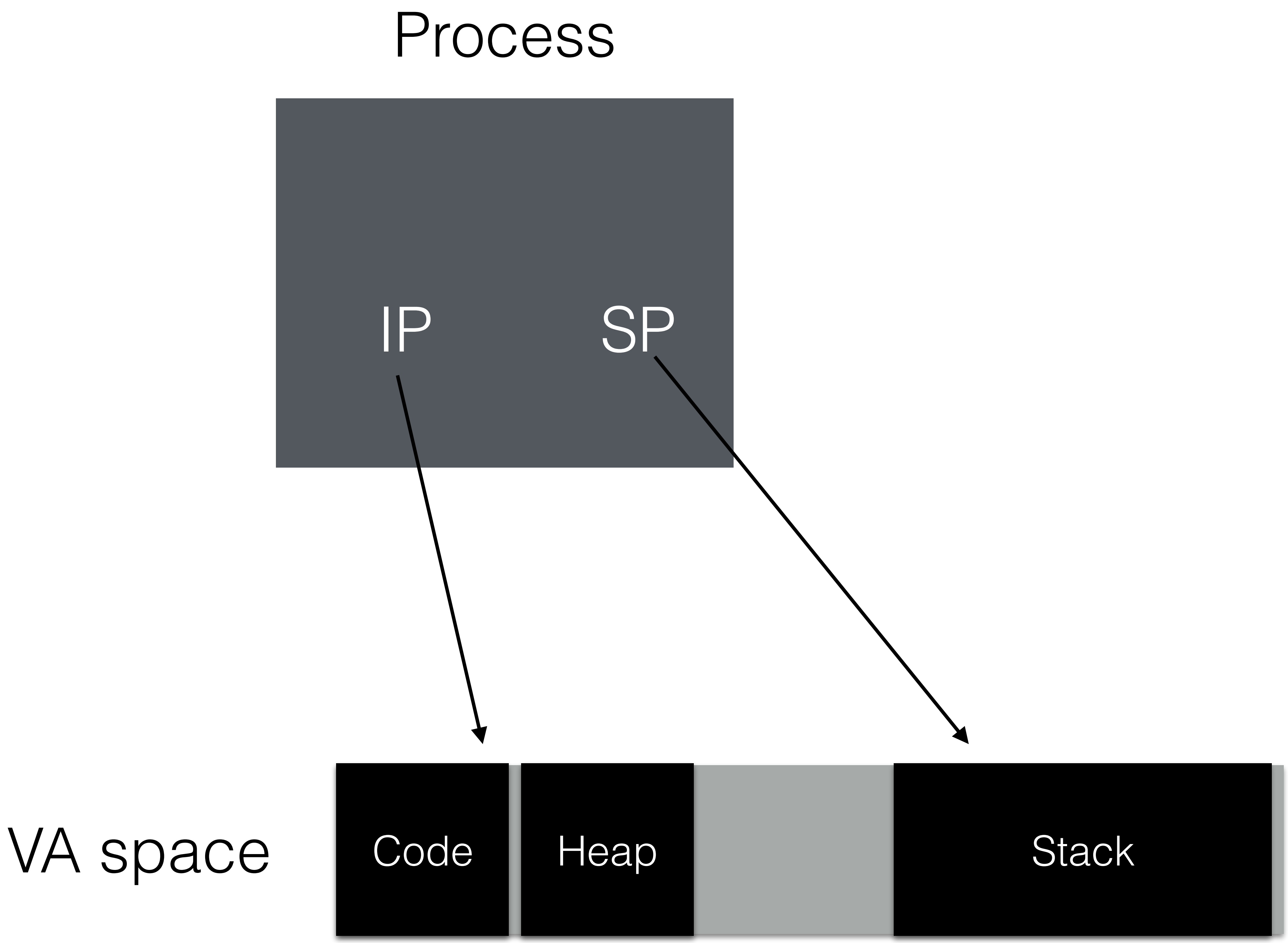
Strategy 2

1. Use Threads : just like processes, but, share the address space (i.e. PT)
2. Do not need to communicate using IPC:
    1. Shared data
3. Single copy of address space!

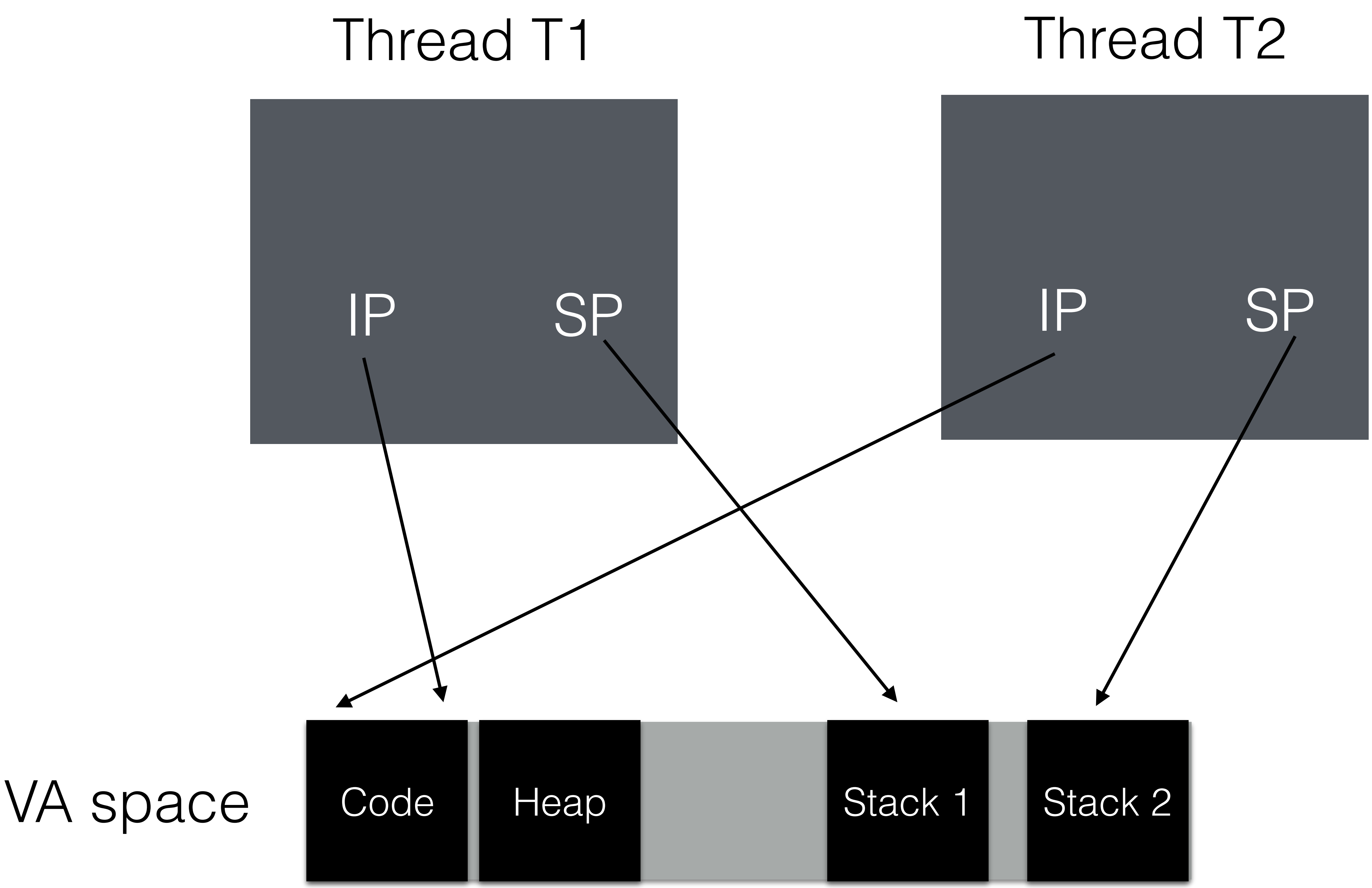
Threading

VA space

Threading



Threading


Thread T1
Thread T2
IP
SP
IP
SP
VA space
Code
Heap
Stack 1
Stack 2

Scheduling Problems …

State

0x20135f: 0
%eax: ?

Thread 1

%eax: ?
IP: cc7

Thread 2

%eax: ?
IP: ?

T1

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

Scheduling Problems …

State

0x20135f: 0
%eax: 0

Thread 1

%eax: 0
IP: ccd

Thread 2

%eax: ?
IP: ?

T1

```
cc7:  mov   0x20135f,%eax
ccd:  add   $0x1,%eax
cd0:  mov   %eax,0x20135f
```

Scheduling Problems …

State

0x20135f: 0
%eax: 1

Thread 1

%eax: 1
IP: cd0

Thread 2

%eax: ?
IP: ?

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

T1

Scheduling Problems …

State

0x20135f: 1
%eax: 1

Thread 1

%eax: 1
IP: cd4

Thread 2

%eax: ?
IP: ?

```
cc7:  mov   0x20135f,%eax
ccd:  add   $0x1,%eax
cd0:  mov   %eax,0x20135f
```

T1

Scheduling Problems …

Context Switch

State

0x20135f: 1
%eax: 1

Thread 1	Thread 2
%eax: 1 IP: cd4	%eax: ? IP: ?

```
cc7:  mov   0x20135f,%eax
ccd:  add   $0x1,%eax
cd0:  mov   %eax,0x20135f
```

T1 →

Scheduling Problems …

Context Switch

State

0x20135f: 1
%eax: 1

Thread 1

%eax: 1
IP: cd4

Thread 2

%eax: ?
IP: ?

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

T1 →

Scheduling Problems …

State

0x20135f: 1
%eax: ?

Thread 1

%eax: 1
IP: cd4

Thread 2

%eax: ?
IP: cc7

T2

```
cc7:  mov   0x20135f,%eax
ccd:  add   $0x1,%eax
cd0:  mov   %eax,0x20135f
```

Scheduling Problems …

State

0x20135f: 1
%eax: 1

Thread 1

%eax: 1
IP: cd4

Thread 2

%eax: 1
IP: ccd

```
     cc7:  mov    0x20135f,%eax
T2 → ccd:  add    $0x1,%eax
     cd0:  mov    %eax,0x20135f
```

Scheduling Problems …

State

0x20135f: 1
%eax: 2

Thread 1

%eax: 1
IP: cd4

Thread 2

%eax: 2
IP: cd0

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

T2 →

Scheduling Problems …

State

0x20135f: 2
%eax: 2

Thread 1

%eax: 1
IP: cd4

Thread 2

%eax: 2
IP: cd4

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

T2 →

Scheduling Problems …(Order #2)

State

0x20135f: 0
%eax: ?

Thread 1

%eax: ?
IP: cc7

Thread 2

%eax: ?
IP: ?

T1 →

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

Scheduling Problems …(Order #2)

State

0x20135f: 0
%eax: 0

Thread 1

%eax: 0
IP: ccd

Thread 2

%eax: ?
IP: ?

T1

```
cc7:  mov   0x20135f,%eax
ccd:  add   $0x1,%eax
cd0:  mov   %eax,0x20135f
```

Scheduling Problems …(Order #2)

State

0x20135f: 0
%eax: 1

Thread 1

%eax: 1
IP: cd0

Thread 2

%eax: ?
IP: ?

```
     cc7: mov  0x20135f,%eax
     ccd: add  $0x1,%eax
T1 → cd0: mov  %eax,0x20135f
```

Scheduling Problems …(Order #2)

Context Switch

State
0x20135f: 0
%eax: 1

Thread 1	Thread 2
%eax: 1	%eax: ?
IP: cd0	IP: ?

	cc7:	mov	0x20135f,%eax
	ccd:	add	$0x1,%eax
T1 →	cd0:	mov	%eax,0x20135f

Scheduling Problems …(Order #2)

Context Switch

State

0x20135f: 0
%eax: ?

Thread 1

%eax: 1
IP: cd0

Thread 2

%eax: ?
IP: cc7

T2 →

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

Scheduling Problems …(Order #2)

State

0x20135f: 0
%eax: 0

Thread 1

%eax: 1
IP: cd0

Thread 2

%eax: 0
IP: ccd

T2 →

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

Scheduling Problems …(Order #2)

State

0x20135f: 0
%eax: 1

Thread 1

%eax: 1
IP: cd0

Thread 2

%eax: 1
IP: cd0

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

T2 →

Scheduling Problems …(Order #2)

State

0x20135f:1
%eax: 1

Thread 1

%eax: 1
IP: cd0

Thread 2

%eax: 1
IP: cd4

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

T2 →

Scheduling Problems …(Order #2)

Context Switch

State

0x20135f:1
%eax: 1

Thread 1

%eax: 1
IP: cd0

Thread 2

%eax: 1
IP: cd4

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

T2 →

Scheduling Problems …(Order #2)

Context Switch

State

0x20135f:1
%eax: 1

Thread 1

%eax: 1
IP: cd0

Thread 2

%eax: 1
IP: cd4

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

T1 →

Scheduling Problems …(Order #2)

Context Switch

State

0x20135f:1
%eax: 1

Thread 1

%eax: 1
IP: cd4

Thread 2

%eax: 1
IP: cd4

```
cc7:  mov   0x20135f,%eax
ccd:  add   $0x1,%eax
cd0:  mov   %eax,0x20135f
```

T1 →

Timeline View

Thread 1

```
cc7: mov    0x20135f,%eax
ccd: add    $0x1,%eax
cd0: mov    %eax,0x20135f
```

Thread 2

```
cc7: mov    0x20135f,%eax
ccd: add    $0x1,%eax
cd0: mov    %eax,0x20135f
```

Timeline View

Thread 1

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

Thread 2

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

What's the value at 0x20135f?

Timeline View

Thread 1

```
cc7: mov    0x20135f,%eax
ccd: add    $0x1,%eax
```

Thread 2

```
cc7: mov    0x20135f,%eax
ccd: add    $0x1,%eax
```

Thread 1

```
cd0: mov    %eax,0x20135f
```

Thread 2

```
cd0: mov    %eax,0x20135f
```

Timeline View

Thread 1

Thread 2

cc7: mov 0x20135f,%eax

ccd: add $0x1,%eax

cc7: mov 0x20135f,%eax

ccd: add $0x1,%eax

cd0: mov %eax,0x20135f

cd0: mov %eax,0x20135f

What's the value at 0x20135f?

Timeline View

Thread 1

```
cc7: mov    0x20135f,%eax
ccd: add    $0x1,%eax
cd0: mov    %eax,0x20135f
```

Thread 2

```
cc7: mov    0x20135f,%eax
ccd: add    $0x1,%eax
cd0: mov    %eax,0x20135f
```

Timeline View

Thread 1

Thread 2

```
cc7: mov    0x20135f,%eax
```

```
                    cc7: mov    0x20135f,%eax
```

```
ccd: add    $0x1,%eax
```

```
                    ccd: add    $0x1,%eax
```

```
cd0: mov    %eax,0x20135f
```

```
                    cd0: mov    %eax,0x20135f
```

What's the value at 0x20135f?

Timeline View

Thread 1	Thread 2
cc7: mov 0x20135f,%eax	
ccd: add $0x1,%eax	
	cc7: mov 0x20135f,%eax
cd0: mov %eax,0x20135f	
	ccd: add $0x1,%eax
	cd0: mov %eax,0x20135f

Timeline View

Thread 1	Thread 2
cc7: mov 0x20135f,%eax	
ccd: add $0x1,%eax	
	cc7: mov 0x20135f,%eax
cd0: mov %eax,0x20135f	
	ccd: add $0x1,%eax
	cd0: mov %eax,0x20135f

What's the value at 0x20135f?

Atomicity

```
cc7:  mov    0x20135f,%eax
ccd:  add    $0x1,%eax
cd0:  mov    %eax,0x20135f
```

Atomicity

1. Want **all or none** of the following instructions to execute —> atomic instruction

```
cc7:  mov   0x20135f,%eax
ccd:  add   $0x1,%eax
cd0:  mov   %eax,0x20135f
```

Atomicity

1. Want **all or none** of the following instructions to execute —> atomic instruction

```
cc7: mov  0x20135f,%eax
ccd: add  $0x1,%eax
cd0: mov  %eax,0x20135f
```

Atomicity

1. Want **all or none** of the following instructions to execute —> atomic instruction

```
cc7: mov   0x20135f,%eax
ccd: add   $0x1,%eax
cd0: mov   %eax,0x20135f
```

Critical section

Atomicity

1. Want **all or none** of the following instructions to execute —> atomic instruction
2. Want **mutual execution** for **critical section** (if T1 runs, T2 can not, and vice versa)

```
cc7: mov    0x20135f,%eax
ccd: add    $0x1,%eax
cd0: mov    %eax,0x20135f
```

Critical section

Practice Questions …

1. Let's examine a simple program, "loop.s". First, just read and understand it. Then, run it with these arguments (`./x86.py -p loop.s -t 1 -i 100 -R dx`) This specifies a single thread, an interrupt every 100 instructions, and tracing of register `%dx`. What will `%dx` be during the run? Use the `-c` flag to check your answers; the answers, on the left, show the value of the register (or memory value) *after* the instruction on the right has run.
2. Same code, different flags: (`./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx`) This specifies two threads, and initializes each %dx to 3. What values will `%dx` see? Run with `-c` to check. Does the presence of multiple threads affect your calculations? Is there a race in this code?
3. Run this: `./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx` This makes the interrupt interval small/random; use different seeds (`-s`) to see different interleavings. Does the interrupt frequency change anything?
4. Now, a different program, `looping-race-nolock.s`, which accesses a shared variable located at address 2000; we'll call this variable `value`. Run it with a single thread to confirm your understanding: `./x86.py -p looping-race-nolock.s -t 1 -M 2000` What is `value` (i.e., at memory address 2000) throughout the run? Use `-c` to check.
5. Run with multiple iterations/threads: `./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000` Why does each thread loop three times? What is final value of `value`?
6. Run with random interrupt intervals: `./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0` with different seeds (`-s 1`, `-s 2`, etc.) Can you tell by looking at the thread interleaving what the final value of `value` will be? Does the timing of the interrupt matter? Where can it safely occur? Where not? In other words, where is the critical section exactly?

Practice Questions …

6. Run with random interrupt intervals: `./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0` with different seeds (`-s 1`, `-s 2`, etc.) Can you tell by looking at the thread interleaving what the final value of `value` will be? Does the timing of the interrupt matter? Where can it safely occur? Where not? In other words, where is the critical section exactly?
7. Now examine fixed interrupt intervals: `./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1` What will the final value of the shared variable `value` be? What about when you change `-i 2`, `-i 3`, etc.? For which interrupt intervals does the program give the "correct" answer?
8. Run the same for more loops (e.g., set `-a bx=100`). What interrupt intervals (`-i`) lead to a correct outcome? Which intervals are surprising?
9. One last program: `wait-for-me.s`. Run: `./x86.py -p wait-for-me.s -a ax=1,ax=0 -R ax -M 2000` This sets the `%ax` register to 1 for thread 0, and 0 for thread 1, and watches `%ax` and memory location 2000. How should the code behave? How is the value at location 2000 being used by the threads? What will its final value be?
10. Now switch the inputs: `./x86.py -p wait-for-me.s -a ax=0,ax=1 -R ax -M 2000` How do the threads behave? What is thread 0 doing? How would changing the interrupt interval (e.g., `-i 1000`, or perhaps to use random intervals) change the trace outcome? Is the program efficiently using the CPU?

Locks

Thread 1

```
cc7: mov    0x20135f,%eax
ccd: add    $0x1,%eax
cd0: mov    %eax,0x20135f
```

Thread 2

```
cc7: mov    0x20135f,%eax
ccd: add    $0x1,%eax
cd0: mov    %eax,0x20135f
```

Locks

Thread 1

Thread 2

- Thread 1 checks if lock is free

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

Locks

Thread 1

Thread 2

- Thread 1 checks if lock is free
- Lock is free, Thread 1 acquires the lock

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

Locks

Thread 1

Thread 2

- Thread 1 checks if lock is free
- Lock is free, Thread 1 acquires the lock
- Thread 2 checks if lock is free

Thread 1:

```
cc7: mov    0x20135f,%eax
ccd: add    $0x1,%eax
cd0: mov    %eax,0x20135f
```

Thread 2:

```
cc7: mov    0x20135f,%eax
ccd: add    $0x1,%eax
cd0: mov    %eax,0x20135f
```

Locks

Thread 1

Thread 2

- Thread 1 checks if lock is free
- Lock is free, Thread 1 acquires the lock
- Thread 2 checks if lock is free
- Is not free; does not execute till lock free

Thread 1:

```
cc7: mov    0x20135f,%eax
ccd: add    $0x1,%eax
cd0: mov    %eax,0x20135f
```

Thread 2:

```
cc7: mov    0x20135f,%eax
ccd: add    $0x1,%eax
cd0: mov    %eax,0x20135f
```

Locks

Thread 1

Thread 2

- Thread 1 checks if lock is free
- Lock is free, Thread 1 acquires the lock
- Thread 2 checks if lock is free
- Is not free; does not execute till lock free

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

cc7:	mov	0x20135f,%eax
ccd:	add	$0x1,%eax
cd0:	mov	%eax,0x20135f

- Thread 1 executes

Locks

Thread 1

Thread 2

- Thread 1 checks if lock is free
- Lock is free, Thread 1 acquires the lock
- Thread 2 checks if lock is free
- Is not free; does not execute till lock free

Thread 1:

```
cc7: mov    0x20135f,%eax
ccd: add    $0x1,%eax
cd0: mov    %eax,0x20135f
```

Thread 2:

```
cc7: mov    0x20135f,%eax
ccd: add    $0x1,%eax
cd0: mov    %eax,0x20135f
```

- Thread 1 executes
- Thread 1 Unlocks

Locks

Thread 1

Thread 2

- Thread 1 checks if lock is free
- Lock is free, Thread 1 acquires the lock
- Thread 2 checks if lock is free
- Is not free; does not execute till lock free

Thread 1:

```
cc7: mov    0x20135f,%eax
ccd: add    $0x1,%eax
cd0: mov    %eax,0x20135f
```

Thread 2:

```
cc7: mov    0x20135f,%eax
ccd: add    $0x1,%eax
cd0: mov    %eax,0x20135f
```

- Thread 1 executes
- Thread 1 Unlocks
- Thread 2 checks (keeps on doing so) for lock being free

Locks

Thread 1 | Thread 2

- Thread 1 checks if lock is free
- Lock is free, Thread 1 acquires the lock
- Thread 2 checks if lock is free
- Is not free; does not execute till lock free

Thread 1:

```
cc7: mov    0x20135f,%eax
ccd: add    $0x1,%eax
cd0: mov    %eax,0x20135f
```

Thread 2:

```
cc7: mov    0x20135f,%eax
ccd: add    $0x1,%eax
cd0: mov    %eax,0x20135f
```

- Thread 1 executes
- Thread 1 Unlocks
- Thread 2 checks (keeps on doing so) for lock being free
- Thread 2 executes and unlocks

Goals of a Lock

Goals of a Lock

- Mutual exclusion: Only a single thread can run the critical section at a time

Goals of a Lock

- Mutual exclusion: Only a single thread can run the critical section at a time
- Fairness: Each thread should get a fair chance of running the critical section. No starvation.

Goals of a Lock

- Mutual exclusion: Only a single thread can run the critical section at a time
- Fairness: Each thread should get a fair chance of running the critical section. No starvation.
- Performance: Low time overhead

Goals of a Lock

- Mutual exclusion: Only a single thread can run the critical section at a time
- Fairness: Each thread should get a fair chance of running the critical section. No starvation.
- Performance: Low time overhead
  - Performance overhead when:

Goals of a Lock

- Mutual exclusion: Only a single thread can run the critical section at a time
- Fairness: Each thread should get a fair chance of running the critical section. No starvation.
- Performance: Low time overhead
    - Performance overhead when:
        - Single thread, no contention

Goals of a Lock

- Mutual exclusion: Only a single thread can run the critical section at a time
- Fairness: Each thread should get a fair chance of running the critical section. No starvation.
- Performance: Low time overhead
    - Performance overhead when:
        - Single thread, no contention
        - Multiple threads, single CPU

Goals of a Lock

- Mutual exclusion: Only a single thread can run the critical section at a time
- Fairness: Each thread should get a fair chance of running the critical section. No starvation.
- Performance: Low time overhead
  - Performance overhead when:
    - Single thread, no contention
    - Multiple threads, single CPU
    - Multiple threads, multiple CPU

Building a Lock - Disable Interrupts

Void lock()
{ Disable Interrupts}

Critical Section

Void unlock()
{ Enable Interrupts}

Building a Lock - Disable Interrupts

Building a Lock - Disable Interrupts

Pros

Building a Lock - Disable Interrupts

Pros

1. Simple and works!

Building a Lock - Disable Interrupts

Pros

1. Simple and works!

Cons

Building a Lock - Disable Interrupts

Pros

1. Simple and works!

Cons

1. Threads are given a lot of trust

Building a Lock - Disable Interrupts

Pros

1. Simple and works!

Cons

1. Threads are given a lot of trust
    1. Call lock() at starting of program and run infinitely

Building a Lock - Disable Interrupts

Pros

1. Simple and works!

Cons

1. Threads are given a lot of trust
    1. Call lock() at starting of program and run infinitely
2. Does not work on multiprocessors

Building a Lock - Disable Interrupts

Pros

1. Simple and works!

Cons

1. Threads are given a lot of trust
    1. Call lock() at starting of program and run infinitely
2. Does not work on multiprocessors
    1. Each processor will have own interrupts?!

Building a Lock - Disable Interrupts

Pros

1. Simple and works!

Cons

1. Threads are given a lot of trust
    1. Call lock() at starting of program and run infinitely
2. Does not work on multiprocessors
    1. Each processor will have own interrupts?!
3. Loss of interrupts

Building a Lock - Disable Interrupts

Pros

1. Simple and works!

Cons

1. Threads are given a lot of trust
    1. Call lock() at starting of program and run infinitely
2. Does not work on multiprocessors
    1. Each processor will have own interrupts?!
3. Loss of interrupts
4. Inefficient - Interrupt routines can be slow

Building a Lock - Load/Store or Flag

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section
- Thread calls lock before entering critical section

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section
- Thread calls lock before entering critical section
    - Is flag set? (some other thread has critical section control)

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section
- Thread calls lock before entering critical section
  - Is flag set? (some other thread has critical section control)
    - Yes - Spin waiting

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section
- Thread calls lock before entering critical section
  - Is flag set? (some other thread has critical section control)
    - Yes - Spin waiting
    - No

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section
- Thread calls lock before entering critical section
  - Is flag set? (some other thread has critical section control)
    - Yes - Spin waiting
    - No
      - set flag, execute critical section

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section
- Thread calls lock before entering critical section
  - Is flag set? (some other thread has critical section control)
    - Yes - Spin waiting
    - No
      - set flag, execute critical section
      - After completion of critical section, unset flag

Building a Lock - Load/Store or Flag

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section
- Thread calls lock before entering critical section

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section
- Thread calls lock before entering critical section
    - Is flag set? (some other thread has critical section control)

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section
- Thread calls lock before entering critical section
  - Is flag set? (some other thread has critical section control)
    - Yes - Spin waiting

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section
- Thread calls lock before entering critical section
  - Is flag set? (some other thread has critical section control)
    - Yes - Spin waiting
    - No

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section
- Thread calls lock before entering critical section
    - Is flag set? (some other thread has critical section control)
        - Yes - Spin waiting
        - No
            - set flag, execute critical section

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section
- Thread calls lock before entering critical section
  - Is flag set? (some other thread has critical section control)
    - Yes - Spin waiting
    - No
      - set flag, execute critical section
      - After completion of critical section, unset flag

Building a Lock - Load/Store or Flag

Building a Lock - Load/Store or Flag

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

Building a Lock - Load/Store or Flag

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

```
void init(lock_t *mutex)
   { // 0 -> lock is available, 1 -> held
   mutex->flag = 0; }
```

Building a Lock - Load/Store or Flag

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

```
void init(lock_t *mutex)
   { // 0 -> lock is available, 1 -> held
   mutex->flag = 0; }
```

```
void lock(lock_t *mutex) {
   while (mutex->flag == 1);
   // spin-wait (do nothing)
   mutex->flag = 1; // now SET it!
   }
```

Building a Lock - Load/Store or Flag

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

```
void init(lock_t *mutex)
   { // 0 -> lock is available, 1 -> held
   mutex->flag = 0; }
```

```
void lock(lock_t *mutex) {
   while (mutex->flag == 1);
   // spin-wait (do nothing)
   mutex->flag = 1; // now SET it!
   }
```

```
void unlock(lock_t *mutex) { mutex->flag = 0; }
```

Building a Lock - Load/Store or Flag

Thread 1	Thread 2

Building a Lock - Load/Store or Flag

Thread 1	Thread 2
Call Lock()	

Building a Lock - Load/Store or Flag

Thread 1	Thread 2
Call Lock()	
Lock held by some other thread	

Building a Lock - Load/Store or Flag

Thread 1	Thread 2
Call Lock()	
Lock held by some other thread	
while(flag ==1) // Busy spinning	

Building a Lock - Load/Store or Flag

Thread 1	Thread 2
Call Lock()	
Lock held by some other thread	
while(flag ==1) // Busy spinning	
Other thread unlocks —> flag = 0	

Building a Lock - Load/Store or Flag

Thread 1	Thread 2
Call Lock()	
Lock held by some other thread	
while(flag ==1) // Busy spinning	
Other thread unlocks —> flag = 0	

Context Switch

Building a Lock - Load/Store or Flag

Thread 1	Thread 2
Call Lock()	
Lock held by some other thread	
while(flag ==1) // Busy spinning	
Other thread unlocks —> flag = 0	

Context Switch

	Call Lock()

Building a Lock - Load/Store or Flag

Thread 1	Thread 2
Call Lock()	
Lock held by some other thread	
while(flag ==1) // Busy spinning	
Other thread unlocks —> flag = 0	

Context Switch

	Call Lock()
	while(flag ==1)

Building a Lock - Load/Store or Flag

Thread 1	Thread 2
Call Lock()	
Lock held by some other thread	
while(flag ==1) // Busy spinning	
Other thread unlocks —> flag = 0	

Context Switch

Thread 1	Thread 2
	Call Lock()
	while(flag ==1)
	flag = 1

Building a Lock - Load/Store or Flag

Thread 1	Thread 2
Call Lock()	
Lock held by some other thread	
while(flag ==1) // Busy spinning	
Other thread unlocks —> flag = 0	

Context Switch

	Call Lock()
	while(flag ==1)
	flag = 1

Context Switch

Building a Lock - Load/Store or Flag

Thread 1	Thread 2
Call Lock()	
Lock held by some other thread	
while(flag ==1) // Busy spinning	
Other thread unlocks —> flag = 0	

Context Switch

	Call Lock()
	while(flag ==1)
	flag = 1

Context Switch

flag = 1	