CONCURRENCY:
THREADS

Questions answered in this lecture:
Why is concurrency useful?
What is a thread and how does it differ from processes?
What can go wrong if scheduling of critical sections is not atomic?

ANNOUNCEMENTS

P2: Due next Friday
• Test scripts released soon
• Purpose of graph is to demonstrate scheduler is working correctly

1st Exam: Congratulations for completing!
• Grades will be posted to Learn@UW
• Return individual sheets next week
• Exam with answers will be posted to course web page

Read as we go along!
• Chapter 26
REVIEW: EASY PIECE 1

Virtualization

CPU

Context Switch

Schedulers

Memory

Allocation

Segmentation

Paging

TLBs

Multilevel

Swapping

MOTIVATION FOR CONCURRENCY

http://cacm.acm.org/magazines/2012/4/147359-cpu-db-recording-microprocessor-history/fulltext
**MOTIVATION**

CPU Trend: Same speed, but multiple cores

Option 0: Run many different applications on one machine

Goal: Write applications that fully utilize many cores

Option 1: Build applications 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 same address space

Approach
• 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?

<table>
<thead>
<tr>
<th>CPU 1</th>
<th>CPU 2</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>running thread 1</td>
<td>running thread 2</td>
<td></td>
</tr>
</tbody>
</table>
What state do threads share?

Do threads share page directories?

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 pointer?

threads executing different functions need different stacks
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)

THREAD API

Variety of thread systems exist
- POSIX Pthreads

Common thread operations
- Create
- Exit
- Join (instead of wait() for processes)
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 single thread of control

Advantages
- Does not require OS support; Portable
- Can tune scheduling policy to meet application demands
- 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
DEMO: BASIC THREADS

THREAD SCHEDULE #1

balance = balance + 1; balance at 0x9cd4

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

Thread 1

process control blocks:
%eax: ?
%rip: 0x195

Thread 2

%eax: ?
%rip: 0x195

T1 → 
0x195 mov 0x9cd4, %eax
0x19a add $0x1, %eax
0x19d mov %eax, 0x9cd4A

What is state after instruction 0x195 completes?

Registers are virtualized by OS;
Each thread thinks it has own
**THREAD SCHEDULE #1**

<table>
<thead>
<tr>
<th>State:</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x9cd4: 100</td>
<td>%eax: ?</td>
<td>%eax: ?</td>
</tr>
<tr>
<td>%eax: 100</td>
<td>%rip: 0x195</td>
<td>%rip: 0x195</td>
</tr>
<tr>
<td>%rip = 0x19a</td>
<td>process</td>
<td>blocks:</td>
</tr>
</tbody>
</table>

T1 →

0x195 mov 0x9cd4, %eax
0x19a add $0x1, %eax
0x19d mov %eax, 0x9cd4A

What is state after instruction 0x19a completes?

---

**THREAD SCHEDULE #1**

<table>
<thead>
<tr>
<th>State:</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x9cd4: 100</td>
<td>%eax: ?</td>
<td>%eax: ?</td>
</tr>
<tr>
<td>%eax: 101</td>
<td>%rip: 0x195</td>
<td>%rip: 0x195</td>
</tr>
<tr>
<td>%rip = 0x19d</td>
<td>process</td>
<td>blocks:</td>
</tr>
</tbody>
</table>

T1 →

0x195 mov 0x9cd4, %eax
0x19a add $0x1, %eax
0x19d mov %eax, 0x9cd4A

What is state after instruction 0x19d completes?
THREAD SCHEDULE #1

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

Thread 1
%eax: ?
%rip: 0x195

Thread 2
%eax: ?
%rip: 0x195

0x195 mov 0x9cd4, %eax
0x19a add $0x1, %eax
0x19d mov %eax, 0x9cd4A

Thread Context Switch
New contents of PCB and %eax and %rip?

What is state after instruction 0x195 completes?

T1

T2
**Thread Schedule #1**

**State:**
- 0x9cd4: 101
- %eax: 101
- %rip = 0x19a

**Thread 1**
- %eax: 101
- %rip: 0x1a2

**Thread 2**
- %eax: ?
- %rip: 0x195

T2 →
- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4A

What is state after instruction 0x19a completes?

**Thread Schedule #1**

**State:**
- 0x9cd4: 101
- %eax: 102
- %rip = 0x19d

**Thread 1**
- %eax: 101
- %rip: 0x1a2

**Thread 2**
- %eax: ?
- %rip: 0x195

T2 →
- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4A

What is state after instruction 0x19d completes?
**THREAD SCHEDULE #1**

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

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax: 101</td>
<td>%eax: ?</td>
</tr>
<tr>
<td>%rip: 0xa1a2</td>
<td>%rip: 0x195</td>
</tr>
</tbody>
</table>

0x195    mov 0x9cd4, %eax
0x19a    add $0x1, %eax
0x19d    mov %eax, 0x9cd4A

T2

Desired Result!

**ANOTHER SCHEDULE**
**THREAD SCHEDULE #2**

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

**Thread 1**
- process:
- control:
- blocks:

**Thread 2**
- %eax: ?
- %rip: 0x195

**T1**

0x195 mov 0x9cd4, %eax
0x19a add $0x1, %eax
0x19d mov %eax, 0x9cd4A

---

**THREAD SCHEDULE #2**

**State:**
- 0x9cd4: 100
- %eax: 100
- %rip = 0x19a

**Thread 1**
- process:
- control:
- blocks:

**Thread 2**
- %eax: ?
- %rip: 0x195

**T1**

0x195 mov 0x9cd4, %eax
0x19a add $0x1, %eax
0x19d mov %eax, 0x9cd4A
### THREAD SCHEDULE #2

<table>
<thead>
<tr>
<th>State:</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x9cd4: 100</td>
<td>%eax: ?</td>
<td>%eax: ?</td>
</tr>
<tr>
<td>%eax: 101</td>
<td>%rip: 0x195</td>
<td>%rip: 0x195</td>
</tr>
<tr>
<td>%rip = 0x19d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Thread Context Switch**

0x195 mov 0x9cd4, %eax
0x19a add $0x1, %eax
0x19d mov %eax, 0x9cd4A

---

### THREAD SCHEDULE #2

<table>
<thead>
<tr>
<th>State:</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x9cd4: 100</td>
<td>%eax: 101</td>
<td>%eax: ?</td>
</tr>
<tr>
<td>%eax: ?</td>
<td>%rip: 0x19d</td>
<td>%rip: 0x195</td>
</tr>
<tr>
<td>%rip = 0x195</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Thread Context Switch**

0x195 mov 0x9cd4, %eax
0x19a add $0x1, %eax
0x19d mov %eax, 0x9cd4A
# Thread Schedule #2

**State:**
- **Thread 1**
  - 0x9cd4: 100
  - %eax: 101
  - %rip = 0x19d

- **Thread 2**
  - 0x9cd4: 100
  - %eax: ?
  - %rip: 0x195

**Process Control Blocks:**
- T2

**0x195**
- mov 0x9cd4, %eax

**0x19a**
- add $0x1, %eax

**0x19d**
- mov %eax, 0x9cd4A

**T2**
- mov 0x9cd4, %eax
- add $0x1, %eax
- mov %eax, 0x9cd4A
### THREAD SCHEDULE #2

<table>
<thead>
<tr>
<th>State:</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x9cd4: 101</td>
<td>%eax: 101</td>
<td>%eax: ?</td>
</tr>
<tr>
<td>%eax: 101</td>
<td>%rip: 0x19d</td>
<td>%rip: 0x195</td>
</tr>
<tr>
<td>%rip = 0x1a2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Thread Context Switch**

<table>
<thead>
<tr>
<th></th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>%eax: 101</td>
<td>%eax: 1</td>
</tr>
<tr>
<td></td>
<td>%rip: 0x19d</td>
<td>%rip: 0x1a2</td>
</tr>
</tbody>
</table>

#### T2
- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4A

#### T1
- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4A
**THREAD SCHEDULE #2**

<table>
<thead>
<tr>
<th>State:</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x9cd4: 101</td>
<td>%eax: 101</td>
<td>%eax: 101</td>
</tr>
<tr>
<td>%eax: 101</td>
<td>%rip: 0x1a2</td>
<td>%rip: 0x1a2</td>
</tr>
<tr>
<td>%rip = 0x1a2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T1

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

**WRONG Result! Final value of balance is 101**
### Timeline View

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov 0x123, %eax</td>
<td>mov 0x123, %eax</td>
</tr>
<tr>
<td>add %0x1, %eax</td>
<td>add %0x2, %eax</td>
</tr>
<tr>
<td>mov %eax, 0x123</td>
<td>mov %eax, 0x123</td>
</tr>
</tbody>
</table>

How much is added to shared variable? 3: correct!

### Timeline View

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov 0x123, %eax</td>
<td>mov 0x123, %eax</td>
</tr>
<tr>
<td>add %0x1, %eax</td>
<td>add %0x2, %eax</td>
</tr>
<tr>
<td>mov %eax, 0x123</td>
<td>mov %eax, 0x123</td>
</tr>
</tbody>
</table>

How much is added? 2: incorrect!
## TIMELINE VIEW

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov 0x123, %eax</td>
<td>mov 0x123, %eax</td>
</tr>
<tr>
<td>mov 0x123, %eax</td>
<td>add %0x2, %eax</td>
</tr>
<tr>
<td>add %0x1, %eax</td>
<td>mov %eax, 0x123</td>
</tr>
<tr>
<td>mov %eax, 0x123</td>
<td>mov %eax, 0x123</td>
</tr>
</tbody>
</table>

How much is added? 1: incorrect!

## TIMELINE VIEW

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov 0x123, %eax</td>
<td>mov 0x123, %eax</td>
</tr>
<tr>
<td>mov 0x123, %eax</td>
<td>add %0x2, %eax</td>
</tr>
<tr>
<td>add %0x1, %eax</td>
<td>mov %eax, 0x123</td>
</tr>
<tr>
<td>mov %eax, 0x123</td>
<td>mov %eax, 0x123</td>
</tr>
</tbody>
</table>

How much is added? 3: correct!
TIMELINE VIEW

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

Thread 2
- mov 0x123, %eax
- add %0x2, %eax
- mov %eax, 0x123

How much is added? 2: incorrect!

NON-DETERMINISM

Concurrency leads to non-deterministic results
- Not deterministic result: different results even with same inputs
- Race conditions

Whether bug manifests depends on CPU schedule!
- Passing tests means little

How to program well for concurrency?
- Imagine scheduler is malicious
- Assume scheduler will pick bad ordering at some point...
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 Ci and Cj
• if process A is in critical section Ci, process B can’t execute Cj
  (okay if other processes do unrelated work)

Specific: Any code that modifies “balance” variable

BREAK

• What is your spirit animal?

• Did you have a favorite pet growing up?

• If you could have any type of pet, what would it be?
SYNCHRONIZATION

Build higher-level synchronization primitives in OS

- Operations that ensure correct ordering of instructions across threads

Motivation: Build them once and get them right

<table>
<thead>
<tr>
<th>Monitors</th>
<th>Locks</th>
<th>Semaphores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Condition Variables</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Loads</td>
<td>Stores</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Test&amp;Set</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disable Interrupts</td>
</tr>
</tbody>
</table>

LOCKS

Goal: Provide mutual exclusion (mutex)

Three common operations:

- 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 (implementation later)
  - `Pthread_mutex_lock(&mylock);`

- Release
  - Release exclusive access to lock; let another process enter critical section
  - `Pthread_mutex_unlock(&mylock);`
MORE DEMOS
MAIN-THREAD-1C
MAIN-THREAD-2C

LESSONS FROM DEMOS

Mutex interface is very easy to use
Tricky to get best performance; trade-off…

Acquiring and releasing locks has significant overhead
• Implication: Don't want to do “too often”

Shorter critical sections mean more concurrency
• Utilize more cores effectively
• Implication: Put locks around smallest portion of code possible

Extreme scenarios for correctness:
• Single big lock around all code; poor performance but works!
Conclusions

Concurrency is needed to obtain high performance by utilizing multiple cores.

Threads are multiple execution streams within a single process or address space:
- Share PID and address space
- Separate registers and stack

Context switches within a critical section can lead to non-deterministic bugs (race conditions)

Use locks to provide mutual exclusion.