Concurrency: Threads

Questions Answered in this Lecture:
• Why is concurrency useful?
• What is a thread and how does it differ from a process?
• What can go wrong if we don’t enforce mutual exclusion for critical sections?
Announcements

• P2a due tomorrow! Don’t expect us to stay up until midnight on Piazza ;)

• I have office hours today! Come get help!

• P1b grades looking good. A handful of you managed to not turn in your info.txt
What is concurrency?

• A more general form of *parallelism*
• The *illusion of multiple execution contexts making progress*
• Execution context = process/thread/etc.
• Does not *require* multiple CPU cores, processors, or machines
• But often involves them
• We’ve already seen concurrency with CPU virtualization! (multiprogramming of processes)
What is parallelism?

• *Special case* of concurrency

• *Two execution contexts* execute *simultaneously*

• Always requires more hardware (*more cores, more processors, more vector units, more machines, etc.*)
Why parallelism?
The Switching Equation

\[ P_d = \alpha CV^2 f \]

Increasing clock frequency is great for performance, but it increases power consumption (and thus heat generated).

We can’t do this forever! At some point clock frequency levels out.
Trends

• Can’t keep ramping up frequency due to power (and thus heat) consumption
  • But we can keep shrinking transistors
  • What to do with all those extra transistors?
  • More cores!
• Challenge: make good use of these cores
Remember...

• One of the roles of the OS is to *provide abstractions to the hardware*
• Or a “*hardware API*” if you like
• What’s the right one for multiple cores?
Why concurrency?

• Increase interactivity (doesn’t really help with performance)
  • The *illusion* of true parallelism

• *latency hiding* (don’t wait for long-running operations)

• Overlapping activities (you probably do this every day)
How to make it happen?

• Option 1: Communicating processes
  • Example: Chrome (process per tab)
  • Example: Windowing system (process for server, one process per client)

• How do we coordinate processes?
  • pipe() (buffer shared between producer proc and consumer proc)
  • messages (message queues)
Pros?

• Don’t need new abstractions
• Good for isolation/security
Cons?

• Hard to program!
• Communication overheads are high
• Context switching is expensive
Option 2: Threads

• Like a process, less state attached
• Namely, threads share an address space (they share the page table(s))
• Divide your task into parts, one thread works on each part
  • Communication is via *shared memory*
Concurrent programming models

- **Producer/consumer**: some threads/procs create work, others process work
- **Client/server**: one thread/proc fields requests from multiple consumers
- **Pipeline**: one thread/proc per task, each passes work to the next thread/proc
- **Daemon**: work gets queued to a background thread
- A lot of others, take CS451 and/or CS546!
What state do threads share?
What threads share page directories?
CPU 1
running thread 1
PTBR

CPU 2
running thread 2
PTBR

RAM
PageDir A
PageDir B
...

PageDir A
PageDir B
Do threads share Instruction Pointer?
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, Qthreads, Cilk, etc.

• Common thread operations
  • create()
  • exit()
  • join(thethread) (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 a 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
Thread Schedule #1

balance = balance + 1; balance at 0x9cd4

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

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

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

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

balance = balance + 1; balance at 0x9cd4
Thread Schedule #1

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

**Thread 1**
- %eax: ?
- %rip: 0x195

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

---

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x195</td>
<td>mov 0x9cd4, %eax</td>
</tr>
<tr>
<td>0x19a</td>
<td>add $0x1, %eax</td>
</tr>
<tr>
<td>0x19d</td>
<td>mov %eax, 0x9cd4A</td>
</tr>
</tbody>
</table>
Thread Schedule #1

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

Thread 1
%eax: ?
%rip: 0x195

Thread 2
%eax: ?
%rip: 0x195

Process control blocks:

0x195 mov 0x9cd4, %eax
0x19a add $0x1, %eax
T1 0x19d mov %eax, 0x9cd4A
Thread Schedule #1

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

process control blocks:
- Thread 1:
  - %eax: ?
  - %rip: 0x195
- Thread 2:
  - %eax: ?
  - %rip: 0x195

T1

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

Thread Context Switch
Thread Schedule #1

State:

- \(0x9cd4\): 101
- \%eax\): ?
- \%rip = 0x195

Thread 1

- \%eax: 101
- \%rip: 0x1a2

Thread 2

- \%eax: ?
- \%rip: 0x195

\(T2\) \[0x195\] mov \(0x9cd4\), \%eax
\[0x19a\] add $0x1, \%eax
\[0x19d\] mov \%eax, \(0x9cd4\)A
Thread Schedule #1

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

Thread 1
%eax: 101
%rip: 0x1a2

Thread 2
%eax: ?
%rip: 0x195

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

T2
Thread Schedule #1

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

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

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

<table>
<thead>
<tr>
<th>%rip</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x195</td>
<td>mov 0x9cd4, %eax</td>
</tr>
<tr>
<td>0x19a</td>
<td>add $0x1, %eax</td>
</tr>
<tr>
<td>0x19d</td>
<td>mov %eax, 0x9cd4A</td>
</tr>
</tbody>
</table>
Thread Schedule #1

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

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

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

```assembly
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```
Thread Schedule #1

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

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

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

**Process control blocks:**

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

T2

**Desired result!**
Another schedule
Thread Schedule #2

balance = balance + 1; balance at 0x9cd4

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

---

**Thread 1**
- %eax: ?
- %rip: 0x195

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

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

**Process Control Blocks:**
- T1
  - 0x195
  - mov 0x9cd4, %eax
- 0x19a
  - add $0x1, %eax
- 0x19d
  - mov %eax, 0x9cd4A
Thread Schedule #2

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

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

Thread Context Switch
Thread Schedule #2

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

**Thread 1**
- %eax: 101
- %rip: 0x19d

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

**T2**
- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4A
Thread Schedule #2

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

**Thread 1**
- $%eax$: 101
- $%rip$: 0x19d

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

**process control blocks:**
- $0x195$
- $0x19a$
- $0x19d$

**T2**
- $0x195$: mov $0x9cd4, $%eax$
- $0x19a$: add $0x1, $%eax$
- $0x19d$: mov $%eax, 0x9cd4A
Thread Schedule #2

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

Thread 1
- %eax: 101
- %rip: 0x19d

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

Process control blocks:

0x195 mov 0x9cd4, %eax
0x19a add $0x1, %eax
T2 0x19d mov %eax, 0x9cd4A
Thread Schedule #2

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

**Thread 1**
- %eax: 101
- %rip: 0x19d

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

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

T2 ➔ Thread Context Switch
Thread Schedule #2

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

**Thread 1**
- %eax: 101
- %rip: 0x19d

**Thread 2**
- %eax: 101
- %rip: 0x1a2

T1

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

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

Thread 1
%eax: 101
%rip: 0x1a2

Thread 2
%eax: 101
%rip: 0x1a2

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

WRONG RESULT! Final balance value is 101
Timeline View: Interleaving #1

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

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

How much is added to shared variable? 3: correct!
Timeline View: Interleaving #2

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

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

How much is added?

2: incorrect!
Timeline View: Interleaving #3

Thread 1

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

Thread 2

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

How much is added?

1: incorrect!
### Timeline View: Interleaving #4

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

**How much is added?**

3: correct!
Timeline View: Interleaving #5

Thread 1

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! (*heisenbug*)
• Passing tests means little
• How to program: assume 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 an *atomic unit*

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

More general:
Need mutual exclusion for critical sections
• if process A is in critical section C, process B can’t be
  (okay if other processes do unrelated work)
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
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
  - `pthread_mutex_lock(&mylock);`

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

• **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, own registers and stack)
• **Context switches** within a critical section **can lead to non-deterministic bugs** (race conditions)
• Use locks to provide mutual exclusion
Implementing Synchronization

• To implement, *need atomic operations*

• Atomic operation: guarantees no other instructions can be interleaved

• Examples of atomic operations
  • Code between interrupts on uniprocessors
    • Disable timer interrupts, don’t do any I/O
  • Loads and stores of words
    • Load r1, B
    • Store r1, A
  • Special hardware instructions
    • *atomic* test & set
    • *atomic* compare & swap
Implementing Locks: Attempt #1

**Turn off interrupts for critical sections**
- Prevent dispatcher from running another thread
- Code executes atomically

```c
void acquire(lock_t *l) {
    disable_interrupts();
}

void release(lock_t *l) {
    enable_interrupts();
}
```

*Disadvantages??*
Implementing Locks: Attempt #2

Code uses a single shared lock variable

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

```cpp
void release() {
    lock = false;
}
```

Why doesn’t this work?