CS 423
Operating System Design: Concurrency (more Threads)

Professor Adam Bates
Fall 2018
Goals for Today

- **Learning Objectives:**
  - Dive yet further into concurrency and threading
- **Announcements:**
  - MP1 out on Friday!

**Reminder:** Please put away devices at the start of class
HW1 Wrap-Up

• Average Score: 88.8%…

• Three questions of medium difficulty (<80%):
  • “Hold-and-wait” situations?
  • which code snippets are wrong?
  • UNIX shell?
Which of the following systems may never exhibit a "hold-and-wait" situation? You may assume that the only blocking that occurs in these systems occurs on mutexes.

a. Systems that ensure that all resources needed for an application are locked in a single atomic operation that either succeeds and locks all requested resources or fails and locks none.
b. Systems with only one mutex.
c. Systems where all mutexes are numbered. A user cannot lock a mutex with a lower number, X, after they have locked a mutex with a larger number, Y > X.
d. Systems of type (a) and (b) only
e. Systems of type (a), (b), or (c)
Which of the following code snippets are wrong?

Case 1
```c
int *p;
*p = 10;
```

Case 2
```c
char a[2];
strcpy(a, "Hi");
```

Case 3
```c
int b[10];
*b = 11;
```
Which of the following best describes a UNIX shell?

- a. Part of the UNIX kernel that executes user commands
- b. A process forked off at UNIX initialization to accept inputs from a user
- c. A system call executed by the UNIX startup routine to accept commands from users
- d. A library that implements various UNIX commands
- e. The UNIX keyboard device driver that interprets keyboard input
Why Concurrency?

• Servers
  – Multiple connections handled simultaneously

• Parallel programs
  – To achieve better performance

• Programs with user interfaces
  – To achieve user responsiveness while doing computation

• Network and disk bound programs
  – To hide network/disk latency
• **Thread**: A single execution sequence that represents a separately schedulable task.

  • *Single execution sequence*: intuitive and familiar programming model

  • *separately schedulable*: OS can run or suspend a thread at any time.

  • Schedulers operate over threads/tasks, both kernel and user threads.

  • *Does the OS protect all threads from one another?*
• Infinite number of processors

• Threads execute with variable speed

The Thread Abstraction

<table>
<thead>
<tr>
<th>Programmer Abstraction</th>
<th>Physical Reality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>Threads</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Running Threads          Ready Threads
### Programmer View

<table>
<thead>
<tr>
<th>Programmer’s View</th>
<th>Possible Execution #1</th>
<th>Possible Execution #2</th>
<th>Possible Execution #3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = x + 1;</td>
<td>x = x + 1;</td>
<td>x = x + 1;</td>
<td>x = x + 1;</td>
</tr>
<tr>
<td>y = y + x;</td>
<td>y = y + x;</td>
<td></td>
<td>y = y + x;</td>
</tr>
<tr>
<td>z = x + 5y;</td>
<td>z = x + 5y;</td>
<td></td>
<td>z = x + 5y;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Thread is suspended.</td>
<td>Thread is suspended.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other thread(s) run.</td>
<td>Other thread(s) run.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Thread is resumed.</td>
<td>Thread is resumed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Variable Speed:** Program must anticipate all of these possible executions
Possible Executions

Processor View

One Execution

Thread 1
Thread 2
Thread 3

Another Execution

Thread 1
Thread 2
Thread 3

Another Execution

Thread 1
Thread 2
Thread 3

Something to look forward to when we discuss scheduling!
Thread Ops

- `thread_create(thread, func, args)`
  Create a new thread to run `func(args)`
- `thread_yield()`
  Relinquish processor voluntarily
- `thread_join(thread)`
  In parent, wait for forked thread to exit, then return
- `thread_exit`
  Quit thread and clean up, wake up joiner if any
#define NTHREADS 10
thread_t threads[NTHREADS];
main() {
    for (i = 0; i < NTHREADS; i++)  thread_create(&threads[i], &go, i);
    for (i = 0; i < NTHREADS; i++) {
        exitValue = thread_join(threads[i]);
        printf("Thread %d returned with %ld\n", i, exitValue);
    }
    printf("Main thread done.\n");
}
void go (int n) {
    printf("Hello from thread %d\n", n);
    thread_exit(100 + n);
    // REACHED?
Ex: threadHello output

```
bash-3.2$ ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```

- Must “thread returned” print in order?
- What is maximum # of threads running when thread 5 prints hello?
- Minimum?
Fork/Join Concurrency

• Threads can create children, and wait for their completion
• Data only shared before fork/after join
• Examples:
  • Web server: fork a new thread for every new connection
    • As long as the threads are completely independent
  • Merge sort
  • Parallel memory copy
void blockzero (unsigned char *p, int length) {
    int i, j;
    thread_t threads[NTHREADS];
    struct bzeroparams params[NTHREADS];

    // For simplicity, assumes length is divisible by NTHREADS.
    for (i = 0, j = 0; i < NTHREADS; i++, j += length/NTHREADS) {
        params[i].buffer = p + i * length/NTHREADS;
        params[i].length = length/NTHREADS;
        thread_create_p(&threads[i], &go, &params[i]);
    }
    for (i = 0; i < NTHREADS; i++) {
        thread_join(threads[i]);
    }
}
Thread Data Structures

<table>
<thead>
<tr>
<th>Shared State</th>
<th>Thread 1’s Per-Thread State</th>
<th>Thread 2’s Per-Thread State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>Thread Control Block (TCB)</td>
<td>Thread Control Block (TCB)</td>
</tr>
<tr>
<td></td>
<td>Stack Information</td>
<td>Stack Information</td>
</tr>
<tr>
<td></td>
<td>Saved Registers</td>
<td>Saved Registers</td>
</tr>
<tr>
<td></td>
<td>Thread Metadata</td>
<td>Thread Metadata</td>
</tr>
<tr>
<td>Global Variables</td>
<td>Stack</td>
<td>Stack</td>
</tr>
<tr>
<td>Heap</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thread Lifecycle

- **Init**
  - Thread Creation
  - `sthread_create()`

- **Ready**
  - Scheduler Resumes Thread
  - `sthread_exit()`

- **Running**
  - Thread Yield/Scheduler Suspends Thread
  - `sthread_yield()`

- **Waiting**
  - Event Occurs
  - Other Thread Calls
  - `sthread_join()`

- **Finished**
  - Thread Exit
  - `sthread_exit()`
Thread Implementations

- Kernel threads
  - Thread abstraction only available to kernel
  - To the kernel, a kernel thread and a single threaded user process look quite similar
- Multithreaded processes using kernel threads
  - Kernel thread operations available via syscall
- User-level threads
  - Thread operations without system calls
Implementing Threads

- Thread_fork(func, args)
  - Allocate thread control block
  - Allocate stack
  - Build stack frame for base of stack (stub)
  - Put func, args on stack
  - Put thread on ready list
  - Will run sometime later (maybe right away!)

- stub(func, args):
  - Call (*func)(args)
  - If return, call thread_exit()
Implementing Threads

- Thread.Exit
  - Remove thread from the ready list so that it will never run again
  - Free the per-thread state allocated for the thread
How do we switch out thread state? (i.e., ctx switch)

# Save caller’s register state
# NOTE: %eax, etc. are ephemeral
pushl %ebx
pushl %ebp
pushl %esi
pushl %edi

# Get offsetof (struct thread, stack)
mov thread_stack_ofs, %edx
# Save current stack pointer to old thread's stack, if any.
movl SWITCH_CUR(%esp), %eax
movl %esp, (%eax,%edx,1)

# Change stack pointer to new thread's stack
# this also changes currentThread
movl SWITCH_NEXT(%esp), %ecx
movl (%ecx,%edx,1), %esp

# Restore caller's register state.
popl %edi
pobl %esi
pobl %ebp
pobl %ebx
ret
A subtlety

• Thread_create puts new thread on ready list
• When it first runs, some thread calls switchframe
  • Saves old thread state to stack
  • Restores new thread state from stack
• Set up new thread’s stack as if it had saved its state in switchframe
  • “returns” to stub at base of stack to run func
Ex: Two Threads call Yield

Thread 1’s instructions
“return” from thread_switch
into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 1 state to TCB
load thread 2 state

Thread 2’s instructions
“return” from thread_switch
into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 2 state to TCB
load thread 1 state

Processor’s instructions
“return” from thread_switch
into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 1 state to TCB
load thread 2 state
“return” from thread_switch
into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 2 state to TCB
load thread 1 state
return from thread_switch
return from thread_yield
call thread_yield
choose another thread
call thread_switch

...
Take 1:
• User thread = kernel thread (Linux, MacOS)
  • System calls for thread fork, join, exit (and lock, unlock, ...)
  • Kernel does context switch
• Simple, but a lot of transitions between user and kernel mode
Multi-threaded User Processes

Take 1:

Kernel

User-Level Processes
Multi-threaded User Processes

Take 2:

• Green threads (early Java)
  • User-level library, within a single-threaded process
  • Library does thread context switch
  • Preemption via upcall/UNIX signal on timer interrupt
  • Use multiple processes for parallelism
  • Shared memory region mapped into each process
Take 3:

- Scheduler activations (Windows 8):
  - Kernel allocates processors to user-level library
  - Thread library implements context switch
  - Thread library decides what thread to run next
- Upcall whenever kernel needs a user-level scheduling decision:
  - Process assigned a new processor
  - Processor removed from process
  - System call blocks in kernel
Take 3: (What’s old is new again)

M:N model multiplexes N user-level threads onto M kernel-level threads

Good idea? Bad Idea?
Question

Compare event-driven programming with multithreaded concurrency. Which is better in which circumstances, and why?