CS 423
Operating System Design: Concurrency

Professor Adam Bates
Fall 2018
• **Learning Objectives:**
  - Understand different primitives for concurrency at the operating system layer

• **Announcements:**
  - C4 weekly summaries! **Due Friday (any time zone)**
  - **MP1 is out! Due Feb 20 (any time zone)**
  - CS Instructional Cloud is back online

**Reminder:** Please put away devices at the start of class
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?*
The Thread Abstraction

- Infinite number of processors
- Threads execute with variable speed

<table>
<thead>
<tr>
<th>Programmer Abstraction</th>
<th>Physical Reality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td></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>
<tr>
<td>Processors</td>
<td></td>
</tr>
</tbody>
</table>

Running Threads  Ready Threads
## Programmer vs. Processor View

### 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>x = x + 1;</td>
<td>x = x + 1;</td>
<td>x = x + 1;</td>
</tr>
<tr>
<td></td>
<td>y = y + x;</td>
<td>y = y + x;</td>
<td>y = y + x;</td>
</tr>
<tr>
<td></td>
<td>z = x + 5y;</td>
<td>z = x + 5y;</td>
<td>z = x + 5y;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example:

```
x = x + 1;
y = y + x;
z = x + 5y;
```

### Variable Speed: Program must anticipate all of these possible executions
### Processor View

#### One Execution

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
</table>

#### Another Execution

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
</table>

### Another Execution

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
</table>

### 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?
- Why aren’t any messages interrupted mid-string?
• 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

Shared State
- Code
- Global Variables
- Heap

Thread 1’s Per–Thread State
- Thread Control Block (TCB)
  - Stack Information
  - Saved Registers
  - Thread Metadata

Thread 2’s Per–Thread State
- Thread Control Block (TCB)
  - Stack Information
  - Saved Registers
  - Thread Metadata

Global Variables
- Head
- Code
- Stack
Thread Lifecycle

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

- **Ready**
  - Scheduler Resumes Thread
  - Thread Yield/Scheduler Suspends Thread: `sthread_yield()`

- **Running**
  - Thread Exit: `sthread_exit()`
  - Thread Waits for Event: `sthread_join()`
  - Other Thread Calls `sthread_join()`

- **Finished**

- **Waiting**
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
Multithreaded OS Kernel

Kernel:
- Code
-Globals
-Heap

Kernel Threads:
- Kernel Thread 1
- Kernel Thread 2
- Kernel Thread 3

Process 1:
- PCB 1
- Stack

Process 2:
- PCB 2
- Stack

User-Level Processes:
- Process 1 Thread
  - Stack
  - Code
  - Globals
  - Heap

- Process 2 Thread
  - Stack
  - Code
  - Globals
  - Heap
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

<table>
<thead>
<tr>
<th>Thread 1’s instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>“return” from thread_switch into stub</td>
</tr>
<tr>
<td>call go</td>
</tr>
<tr>
<td>call thread_yield</td>
</tr>
<tr>
<td>choose another thread</td>
</tr>
<tr>
<td>call thread_switch</td>
</tr>
<tr>
<td>save thread 1 state to TCB</td>
</tr>
<tr>
<td>load thread 2 state</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 2’s instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>“return” from thread_switch into stub</td>
</tr>
<tr>
<td>call go</td>
</tr>
<tr>
<td>call thread_yield</td>
</tr>
<tr>
<td>choose another thread</td>
</tr>
<tr>
<td>call thread_switch</td>
</tr>
<tr>
<td>save thread 2 state to TCB</td>
</tr>
<tr>
<td>load thread 1 state</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor’s instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>“return” from thread_switch into stub</td>
</tr>
<tr>
<td>call go</td>
</tr>
<tr>
<td>call thread_yield</td>
</tr>
<tr>
<td>choose another thread</td>
</tr>
<tr>
<td>call thread_switch</td>
</tr>
<tr>
<td>save thread 1 state to TCB</td>
</tr>
<tr>
<td>load thread 2 state</td>
</tr>
<tr>
<td>“return” from thread_switch into stub</td>
</tr>
<tr>
<td>call go</td>
</tr>
<tr>
<td>call thread_yield</td>
</tr>
<tr>
<td>choose another thread</td>
</tr>
<tr>
<td>call thread_switch</td>
</tr>
<tr>
<td>save thread 2 state to TCB</td>
</tr>
<tr>
<td>load thread 1 state</td>
</tr>
</tbody>
</table>

Figure 4.15: Interleaving of instructions when two threads loop and call thread_yield().
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:
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
M:N model multiplexes N user-level threads onto M kernel-level threads

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