Concurrency
Motivation

• Operating systems (and application programs) often need to be able to handle multiple things happening at the same time
  – Process execution, interrupts, background tasks, system maintenance
• Humans are not very good at keeping track of multiple things happening simultaneously
• Threads are an abstraction to help bridge this gap
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
Déjà vu?

• Didn’t we learn all about concurrency in CSE 332/333?
  – More practice
    • Realistic examples, especially in the project
  – Design patterns and pitfalls
    • Methodology for writing correct concurrent code
  – Implementation
    • How do threads work at the machine level?
  – CPU scheduling
    • If multiple threads to run, which do we do first?
Definitions

• A thread is a single execution sequence that represents a separately schedulable task
  – Single execution sequence: familiar programming model
  – Separately schedulable: OS can run or suspend a thread at any time

• Protection is an orthogonal concept
  – Can have one or many threads per protection domain
Threads in the Kernel and at User-Level

- Multi-threaded kernel
  - multiple threads, sharing kernel data structures, capable of using privileged instructions

- Multiprocess kernel
  - Multiple single-threaded processes
  - System calls access shared kernel data structures

- Multiple multi-threaded user processes
  - Each with multiple threads, sharing same data structures, isolated from other user processes
Thread Abstraction

- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule

![Diagram showing threads and processors]

- Programmer Abstraction
  - Threads: 1, 2, 3, 4, 5
  - Processors: 1, 2
- Physical Reality
  - Running Threads: 1, 2
  - Ready Threads: 4, 5, 3, 2, 1
Question

Why do threads execute at variable speed?
# Programmer vs. Processor 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>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thread is suspended.</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>
<tr>
<td></td>
<td>y = y + x;</td>
<td>z = x + 5y;</td>
<td>z = x + 5y;</td>
</tr>
</tbody>
</table>
Possible Executions

One Execution

Thread 1
Thread 2
Thread 3

Another Execution

Thread 1
Thread 2
Thread 3

Another Execution

Thread 1
Thread 2
Thread 3
Thread Operations

- **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?
}
threadHello: Example Output

- Why must “thread returned” print in order?
- What is maximum # of threads running when thread 5 prints hello?
- Minimum?

```
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.
```
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
bzero with fork/join concurrency

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
- Stack

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

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

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

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

- **Waiting**

- **Finished**
Implementing Threads: Roadmap

• 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 (Linux, MacOS, Windows)
  – Kernel thread operations available via syscall

• User-level threads (Windows)
  – Thread operations without system calls
Multithreaded OS Kernel

Kernel

Kernel Thread 1

Kernel Thread 2

Kernel Thread 3

Process 1

Process 2

Code

Globals

Stack

Heap

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()
Thread Stack

• What if a thread puts too many procedures on its stack?
  – What happens in Java?
  – What happens in the Linux kernel?
  – What happens in xk?
  – What should happen?
Thread Context Switch

• Voluntary
  – Thread_yield
  – Thread_join (if child is not done yet)

• Involuntary
  – Interrupt or exception
  – Some other thread is higher priority
Voluntary thread context switch

• Save registers on old stack
• Switch to new stack, new thread
• Restore registers from new stack
• Return
• Exactly the same with kernel threads or user threads
  – xk: switch is always between kernel threads
x86 switch_threads

# Save caller’s register state
pushl %ebx
pushl %ebp
pushl %esi
pushl %edi

# NOTE: %eax, etc. are ephemeral
pushl %ebp
pushl %esi
pushl %edi

# Get offset of struct thread.stack
mov thread_stack_stack_ofs, %edx

# Save current stack pointer
movl SWITCH_CUR(%esp), %eax

# Change stack pointer;
# stack points to new TCB
movl SWITCH_NEXT(%esp), %ecx
movl (%ecx,%edx,1), %esp

# Restore caller's register state.
popl %edi
popl %esi
popl %ebp
popl %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
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_switch
return from thread_yield
call thread_yield
choose another thread
call thread_switch

return from thread_switch
return from thread_switch
return from thread_yield
call thread_yield
choose another thread
call thread_switch

...
Involuntary Thread/Process Switch

- Timer or I/O interrupt
  - Tells OS some other thread should run
- Simple version (OS/161)
  - End of interrupt handler calls switch()
  - When resumed, return from handler resumes kernel thread or user process
  - Thus, processor context is saved/restored twice (once by interrupt handler, once by thread switch)
Faster Thread/Process Switch

• What happens on a timer (or other) interrupt?
  – Interrupt handler saves state of interrupted thread
  – Decides to run a new thread
  – Throw away current state of interrupt handler!
  – Instead, set saved stack pointer to trapframe
  – Restore state of new thread
  – On resume, pops trapframe to restore interrupted thread
Multithreaded User Processes (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
Multithreaded User Processes (Take 1)
Multithreaded 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
Multithreaded User Processes (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
Question

• Compare event-driven programming with multithreaded concurrency. Which is better in which circumstances, and why?
OS/161 switchframe_switch

/* a0: pointer to old thread control block
 * a1: pointer to new thread control block */
/* Allocate stack space for 10 registers. */
addi sp, sp, -40
/* Save the registers */
sw ra, 36(sp)
sw gp, 32(sp)
sw s8, 28(sp)
sw s6, 24(sp)
sw s5, 20(sp)
sw s4, 16(sp)
sw s3, 12(sp)
sw s2, 8(sp)
sw s1, 4(sp)
sw s0, 0(sp)
/* Store old stack pointer in old thread */
sw sp, 0(a0)

/* Get new stack pointer from new thread */
lw sp, 0(a1)
nop /* delay slot for load */
/* Now, restore the registers */
lw s0, 0(sp)
lw s1, 4(sp)
lw s2, 8(sp)
lw s3, 12(sp)
lw s4, 16(sp)
lw s5, 20(sp)
lw s6, 24(sp)
lw s8, 28(sp)
lw gp, 32(sp)
lw ra, 36(sp)
nop /* delay slot for load */
j ra /* and return. */
addi sp, sp, 40 /* in delay slot */