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?
  – 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

• Single threaded user program  
  – one thread, one protection domain

• Multi-threaded user program  
  – multiple threads, sharing same data structures, 
    isolated from other user programs

• Multiprocess kernel  
  – Multiple processes, sharing kernel data structures

• Multi-threaded kernel  
  – multiple threads, sharing kernel data structures, 
    capable of using privileged instructions
Thread Abstraction

• Infinite number of processors
• Threads execute with variable speed
  – Programs must be designed to work with any schedule
## 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></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Possible Execution #1:
- `x = x + 1;`
- `y = y + x;`
- `z = x + 5y;`

Possible Execution #2:
- `x = x + 1;`
- `y = y + x;`
- `z = x + 5y;`
- Thread is suspended
- Other thread(s) run
- Thread is resumed

Possible Execution #3:
- `x = x + 1;`
- `y = y + x;`
- `z = x + 5y;`
- Thread is suspended
- Other thread(s) run
- Thread is resumed
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)
  - OS/161: thread_fork
- **thread_yield()**
  - Relinquish processor voluntarily
  - OS/161: thread_yield
- **thread_join(thread)**
  - In parent, wait for forked thread to exit, then return
  - OS/161: tbd
- **thread_exit**
  - Quit thread and clean up, wake up joiner if any
  - OS/161: thread_exit
Example: threadHello

#define NTHREADS 10
thread_t threads[NTHREADS];

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);
    // Not reached
}
threadHello: Example Output

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

```bash
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
Thread Lifecycle

Init

Thread Creation
stthread_create()

Ready

Scheduler Resumes Thread

Running

Thread Exit
stthread_exit()

Finished

Thread Yield/Scheduler Suspends Thread
stthread_yield()

Waiting

Event Occurs
(Other Thread Calls
stthread_join())

Thread Waits for Event
stthread_join()
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)
  – Kernel thread operations available via syscall

• User-level threads
  – Thread operations without system calls
NOTE: this picture has an error; there should be an exception stack in the kernel for each process, and no separate kernel thread on the right.
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):** OS/161 mips_threadstart
  - 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 OS/161?
  - 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
  – OS/161: thread switch is always between kernel threads, not between user process and kernel thread
x86 switch_threads (oldT, nextT) (interrupts disabled)

# Save caller’s register state
# NOTE: %eax, etc. are ephemeral
# This stack frame must match the one set up by thread_create()
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
popp %esi
popp %ebp
popp %ebx
ret
Two threads call yield

<table>
<thead>
<tr>
<th>Thread 1’s instructions</th>
<th>Thread 2’s instructions</th>
<th>Processor’s instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>call thread_yield</td>
<td>call thread_yield</td>
<td>call thread_yield</td>
</tr>
<tr>
<td>save state to stack</td>
<td>save state to stack</td>
<td>save state to stack</td>
</tr>
<tr>
<td>save state to TCB</td>
<td>save state to TCB</td>
<td>save state to TCB</td>
</tr>
<tr>
<td>choose another thread</td>
<td>choose another thread</td>
<td>choose another thread</td>
</tr>
<tr>
<td>load other thread state</td>
<td>load other thread state</td>
<td>load other thread state</td>
</tr>
<tr>
<td>return thread_yield</td>
<td>return thread_yield</td>
<td>return thread_yield</td>
</tr>
<tr>
<td>call thread_yield</td>
<td>call thread_yield</td>
<td>call thread_yield</td>
</tr>
<tr>
<td>save state to stack</td>
<td>save state to stack</td>
<td>save state to stack</td>
</tr>
<tr>
<td>save state to TCB</td>
<td>save state to TCB</td>
<td>save state to TCB</td>
</tr>
<tr>
<td>choose another thread</td>
<td>choose another thread</td>
<td>choose another thread</td>
</tr>
<tr>
<td>load other thread state</td>
<td>load other thread state</td>
<td>load other thread state</td>
</tr>
<tr>
<td>return thread_yield</td>
<td>return thread_yield</td>
<td>return thread_yield</td>
</tr>
<tr>
<td>call thread_yield</td>
<td>call thread_yield</td>
<td>call thread_yield</td>
</tr>
<tr>
<td>save state to stack</td>
<td>save state to stack</td>
<td>save state to stack</td>
</tr>
<tr>
<td>save state to TCB</td>
<td>save state to TCB</td>
<td>save state to TCB</td>
</tr>
<tr>
<td>choose another thread</td>
<td>choose another thread</td>
<td>choose another thread</td>
</tr>
<tr>
<td>load other thread state</td>
<td>load other thread state</td>
<td>load other thread state</td>
</tr>
<tr>
<td>return thread_yield</td>
<td>return thread_yield</td>
<td>return thread_yield</td>
</tr>
<tr>
<td>call thread_yield</td>
<td>call thread_yield</td>
<td>call thread_yield</td>
</tr>
<tr>
<td>save state to stack</td>
<td>save state to stack</td>
<td>save state to stack</td>
</tr>
<tr>
<td>save state to TCB</td>
<td>save state to TCB</td>
<td>save state to TCB</td>
</tr>
<tr>
<td>choose another thread</td>
<td>choose another thread</td>
<td>choose another thread</td>
</tr>
<tr>
<td>load other thread state</td>
<td>load other thread state</td>
<td>load other thread state</td>
</tr>
<tr>
<td>return thread_yield</td>
<td>return thread_yield</td>
<td>return thread_yield</td>
</tr>
<tr>
<td>call thread_yield</td>
<td>call thread_yield</td>
<td>call thread_yield</td>
</tr>
<tr>
<td>save state to stack</td>
<td>save state to stack</td>
<td>save state to stack</td>
</tr>
<tr>
<td>save state to TCB</td>
<td>save state to TCB</td>
<td>save state to TCB</td>
</tr>
<tr>
<td>choose another thread</td>
<td>choose another thread</td>
<td>choose another thread</td>
</tr>
<tr>
<td>load other thread state</td>
<td>load other thread state</td>
<td>load other thread state</td>
</tr>
<tr>
<td>return thread_yield</td>
<td>return thread_yield</td>
<td>return thread_yield</td>
</tr>
</tbody>
</table>

Figure 4.13: Interleaving of instructions when two threads loop and call thread_yield().
Involuntary Thread Switch

• Timer or I/O interrupt
  – Tells OS some other thread should run

• Simple version (OS/161)
  – End of interrupt handler calls schedule()
  – When resumed, return from handler resumes kernel thread or user process

• Faster version (Linux)
  – Interrupt handler returns to saved state in TCB
  – Could be kernel thread or user process
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
NOTE: this picture has an error; there should be an exception stack in the kernel for each user thread, and no separate kernel thread on the right.
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 action requires (user-level) scheduling decision
  • Process assigned a new processor
  • Processor removed from process
  • System call blocks in kernel
Question

• Compare event-driven programming (333) with multithreaded concurrency (451). Which is better in which circumstances, and why?