Main Points

- Thread abstraction
  - What are threads and what is the thread abstraction?
- Thread life cycle
  - What states does a thread go through?
- Thread Implementation
  - How does kernel implement the thread abstraction?
Motivation

- Operating systems (and applications) often need 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
Concurrent Tasks:
Earth Visualizer Example
Why Concurrency?

• Expressing logically concurrent tasks
  • To simulate real-world applications

• Exploiting multiple processors
  • To achieve better performance

• Running work in the background
  • To improve user responsiveness while doing computation

• Managing I/O devices
  • To hide network/disk latency
Definition

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

• Is a kernel interrupt handler a thread?
  • Hint: is it independently schedulable?
Threads in Kernel and at User-Level

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

- Multi-process 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
Question

• Why do threads execute at unpredictable and 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></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>
<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>$y = y + x;$</td>
<td>$y = y + x;$</td>
</tr>
<tr>
<td>$z = x + 5y;$</td>
<td>$z = x + 5y;$</td>
<td>$...............$</td>
<td>$...............$</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>$y = y + x;$</td>
<td>$...............$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z = x + 5y;$</td>
<td>$z = x + 5y;$</td>
</tr>
</tbody>
</table>
# Possible Interleavings

## One Execution

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

## Another Execution

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

## Another Execution

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cooperative vs. Preemptive Multi-threading

• Cooperative threads
  • Run without interruption
  • Explicitly relinquish processor to another thread
  • Long-running threads can monopolize processor
    • Starvation and non-responsiveness
  • Less often used today

• Preemptive threads
  • Can be switched at any time
  • What we mean by the term “multi-threading”
Simple Thread API

- **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 the 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);
}
threadHello: Example Output

- Why 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]);
  }
}

void go (struct bzeroparams *p) {
  memset(p->buffer, 0, p->length);
}
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

Heap

Stack

Stack
Thread Lifecycle

- **Init**: Thread Creation
  - `thread_create()`
- **Ready**: Scheduler Resumes Thread
  - SchedulerSuspends Thread
  - `thread_yield()`
- **Running**: Event Occurs
  - Other Thread Calls
  - `thread_exit()`
- **Waiting**: Thread Waits for Event
  - `thread_join()`
- **Finished**: Thread Exit
  - `thread_exit()`
Question

• For the threadHello program, what is the minimum and maximum number of times that the main thread enters the ready state on a uniprocessor?
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

• Multi-threaded processes using kernel threads
  • E.g., Linux, MacOS
  • Kernel thread operations available via syscall

• User-level threads
  • Thread operations without system calls
Multi-threaded OS Kernel

Kernel

- Code
- Global
- Stack
- Heap

Kernel Thread 1
- TCB 1
- Stack
- Process 1
- Stack
- Code
- Global
- Heap

Kernel Thread 2
- TCB 2
- Stack
- Process 2
- Stack
- Code
- Global
- Heap

Kernel Thread 3
- TCB 3
- Stack
- Process 1
- Stack
- Code
- Global
- Heap

User-Level Processes

Process 1
- Thread
- Stack
- Code
- Global
- Heap

Process 2
- Thread
- Stack
- Code
- Global
- Heap
Implementing threads

- thread_create(func, args)
  - Allocate TCB
  - Allocate stack
  - Build stack frame for base of stack (stub)
  - Push func, args on stack
  - Put thread on ready list
  - Create a dummy stack frame (needed by thread_switch)
  - 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 should happen?
Thread Context Switch

- Voluntary
  - `thread_yield`
  - `thread_exit`
  - `thread_join` (if child is not done yet)

- Involuntary
  - Interrupt or exception
  - Some other thread is higher priority
Voluntary Thread Context Switch

- Mask interrupts (why?)
- Choose another TCB from the ready list
  - Go back to original thread if nothing else to run
- Move original thread onto ready list
- Call thread_switch to switch to the new thread
- Unmask interrupts
```c
void thread_switch(oldThreadTCB, newThreadTCB) {
    pushad; // Push general register values onto old stack
    oldThreadTCB->sp = %esp; // Save the old thread’s stack pointer
    %esp = newThreadTCB->sp; // Switch to the new stack
    popad; // Pop register values from the new stack
    return;
}
```
A Subtlety

• Thread_create puts new thread on ready list
• When it first runs, some thread calls `thread_switch`
  • 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 `thread_switch`
  • “returns” to stub at base of stack to run `func`

```c
*(tcb->sp) = stub; // Return to the beginning of stub.
tcb->sp--;
tcb->sp -= SizeOfPopad;
```
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
```
Involuntary Thread/Process Switch

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

• Simple version
  • End of interrupt handler calls thread_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 trap frame
  • Restore state of new thread
  • On resume, pops trap frame to restore interrupted thread
Multi-threaded User Processes (Take 1)

- User thread = kernel thread (Linux, MacOS)
  - System calls for thread create, 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 cont.)
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
Multi-threaded 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?
Acknowledgment

• This lecture is a slightly modified version of the one prepared by Tom Anderson