Recall: Use of Threads

- Version of program with Threads (loose syntax):

  ```
  main() {
    ThreadFork(ComputePI, "pi.txt");
    ThreadFork(PrintClassList, "classList.txt");
  }
  ```

  - What does `ThreadFork()` do?
    - Start independent thread running given procedure
  - What is the behavior here?
    - Now, you would actually see the class list
      - This *should* behave as if there are two separate CPUs

```
CPU1 CPU2 CPU1 CPU2 CPU1 CPU2
```

Time

Recall: Memory Footprint: Two-Threads

- If we stopped this program and examined it with a debugger, we would see
  - Two sets of CPU registers
  - Two sets of Stacks

- Questions:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks?
  - What happens if threads violate this?
  - How might you catch violations?

Recall: Dispatch Loop

- Conceptually, the dispatching loop of the operating system looks as follows:

  ```
  Loop {
    RunThread();
    newTCB = ChooseNextThread();
    SaveStateOfCPU(curTCB);
    LoadStateOfCPU(newTCB);
  }
  ```

  - This is an *infinite* loop
    - One could argue that this is all that the OS does
  - Should we ever exit this loop??
    - When would that be?
Saving/Restoring state (often called “Context Switch”)

```c
switch(tCur,tNew) {
    /* Unload old thread */
    TCB[tCur].regs.r7 = CPU.r7;
    ...
    TCB[tCur].regs.r0 = CPU.r0;
    TCB[tCur].regs.sp = CPU.sp;
    TCB[tCur].regs.retpc = CPU.retpc; /*return addr*/

    /* Load and execute new thread */
    CPU.r7 = TCB[tNew].regs.r7;
    ...
    CPU.r0 = TCB[tNew].regs.r0;
    CPU.sp = TCB[tNew].regs.sp;
    CPU.retpc = TCB[tNew].regs.retpc;
    return; /* Return to CPU.retpc */
}
```

Switch Details (continued)

- What if you make a mistake in implementing switch?
  - Suppose you forget to save/restore register 32
  - Get intermittent failures depending on when context switch occurred and whether new thread uses register 32
  - System will give wrong result without warning

- Can you devise an exhaustive test to test switch code?
  - No! Too many combinations and inter-leavings

- Cautionary tale:
  - For speed, Topaz kernel saved one instruction in switch()
  - Carefully documented! Only works as long as kernel size < 1MB
  - What happened?
    » Time passed, People forgot
    » Later, they added features to kernel (no one removes features!)
    » Very weird behavior started happening
  - Moral of story: Design for simplicity

Some Numbers

- Frequency of performing context switches: 10-100ms
- Context switch time in Linux: 3-4 µsecs (Intel i7 & E5)
  - Thread switching faster than process switching (100 ns)
  - But switching across cores ~2x more expensive than within-core

- Context switch time increases sharply with size of working set*
  - Can increase 100x or more

*The working set is subset of memory used by process in a time window

- Moral: context switching depends mostly on cache limits and the process or thread’s hunger for memory

Some Numbers

- Many process are multi-threaded, so thread context switches may be either within-process or across-processes
What happens when thread blocks on I/O?

- What happens when a thread requests a block of data from the file system?
  - User code invokes a system call
  - Read operation is initiated
  - Run new thread/switch
- Thread communication similar
  - Wait for Signal/Join
  - Networking

External Events

- What happens if thread never does any I/O, never waits, and never yields control?
  - Could the ComputePI program grab all resources and never release the processor?
    » What if it didn’t print to console?
  - Must find way that dispatcher can regain control!
- Answer: utilize external events
  - Interrupts: signals from hardware or software that stop the running code and jump to kernel
  - Timer: like an alarm clock that goes off every some milliseconds
- If we make sure that external events occur frequently enough, can ensure dispatcher runs

Example: Network Interrupt

- An interrupt is a hardware-invoked context switch
  - No separate step to choose what to run next
  - Always run the interrupt handler immediately

Use of Timer Interrupt to Return Control

- Solution to our dispatcher problem
  - Use the timer interrupt to force scheduling decisions
- Timer Interrupt routine:

```
Some Routine

TimerInterrupt() {
  DoPeriodicHouseKeeping();
  run_new_thread();
  switch
}
```
Thread Abstraction

- Illusion: Infinite number of processors

Programmer vs. Processor View

Programmer’s View | Possible Execution
--- | ---
#1 | . .
 | . .
x = x + 1; y = y + x; z = x + 5y;
 | . .

Possible Execution #1

Possible Execution #2

x = x + 1; y = y + x; thread is suspended
z = x + 5y; other thread(s) run

thread is resumed

y = y + x
z = x + 5y
Programmer vs. Processor View

Programmer's View
Possible Execution
#1
#2
#3
x = x + 1;
y = y + x;
z = x + 5y;

Possible Execution
x = x + 1;
y = y + x;
z = x + 5y;

Possible Execution
x = x + 1;
y = y + x;
z = x + 5y;

x = x + 1;   x = x + 1   x = x + 1
y = y + x;   y = y + x;   y = y + x
z = x + 5y;   z = x + 5y;  thread is suspended

Possible Executions
#1
#2
#3

a) One execution
b) Another execution
c) Another execution

Thread Lifecycle

Init
Ready
Running
Finished

Thread Creation
Thread Resumes Thread
Scheduler Suspends Thread
Thread Yields/Susends Thread
Event Occurs
Scheduler Suspends Thread
Thread Waits for Event

Thread Exit
Scheduler Resumes Thread
Thread Exits

Administrivia

• Your section is your home for CS162
  – The TA needs to get to know you to judge participation
  – All design reviews will be conducted by your TA
  – You can attend alternate section by same TA, but try to keep the amount of such cross-section movement to a minimum

• First midterm: Wed Feb 28 6:30 – 8:30 PM
  – ROOM ASSIGNMENTS TBD
  – LET US KNOW DSP AND ACADEMIC CONFLICTS ASAP
  – 245 Li Ka Shing
  – 20 Barrows Hall
  – A1 Hearst Field Annex
  – 2060 Valley Life Sciences Building
  – 102 Wurster Hall
Per Thread Descriptor
(Kernel Supported Threads)

- Each Thread has a Thread Control Block (TCB)
  - Execution State: CPU registers, program counter (PC), pointer to stack (SP)
  - Scheduling info: state, priority, CPU time
  - Various Pointers (for implementing scheduling queues)
  - Pointer to enclosing process (PCB) – user threads
  - Etc (add stuff as you find a need)

- OS Keeps track of TCBs in “kernel memory”
  - In Array, or Linked List, or …
  - I/O state (file descriptors, network connections, etc)

ThreadFork(): Create a New Thread

- ThreadFork() is a user-level procedure that creates a new thread and places it on ready queue

- Arguments to ThreadFork()
  - Pointer to application routine (fcnPtr)
  - Pointer to array of arguments (fcnArgPtr)
  - Size of stack to allocate

- Implementation
  - Sanity check arguments
  - Enter Kernel-mode and Sanity Check arguments again
  - Allocate new Stack and TCB
  - Initialize TCB and place on ready list (Runnable)

How do we initialize TCB and Stack?

- Initialize Register fields of TCB
  - Stack pointer made to point at stack
  - PC return address ⇒ OS (asm) routine ThreadRoot()
  - Two arg registers (a0 and a1) initialized to fcnPtr and fcnArgPtr, respectively

- Initialize stack data?
  - No. Important part of stack frame is in registers (ra)
  - Think of stack frame as just before body of ThreadRoot() really gets started
How does Thread get started?

- Eventually, `run_new_thread()` will select this TCB and return into beginning of `ThreadRoot()`
  - This really starts the new thread

Stack growth

- A
- B(while)
- yield
- run_new_thread
- switch

New Thread

ThreadRoot stub

What does ThreadRoot() look like?

- `ThreadRoot()` is the root for the thread routine:
  ```plaintext
  ThreadRoot() {
      DoStartupHousekeeping();
      UserModeSwitch(); /* enter user mode */
      Call fcnPtr(fcnArgPtr);
      ThreadFinish();
  }
  ```
  - Startup Housekeeping
    - Includes things like recording start time of thread
    - Other statistics
- Stack will grow and shrink with execution of thread
- Final return from thread returns into `ThreadRoot()` which calls `ThreadFinish()`
  - `ThreadFinish()` wake up sleeping threads

Multithreaded Processes

- Process Control Block (PCBs) points to multiple Thread Control Blocks (TCBs):

  - Switching threads within a block is a simple thread switch
  - Switching threads across blocks requires changes to memory and I/O address tables

Examples multithreaded programs

- Embedded systems
  - Elevators, planes, medical systems, smart watches
  - Single program, concurrent operations

- Most modern OS kernels
  - Internally concurrent because have to deal with concurrent requests by multiple users
  - But no protection needed within kernel

- Database servers
  - Access to shared data by many concurrent users
  - Also background utility processing must be done
Example multithreaded programs (con’t)

• Network servers
  – Concurrent requests from network
  – Again, single program, multiple concurrent operations
  – File server, Web server, and airline reservation systems

• Parallel programming (more than one physical CPU)
  – Split program into multiple threads for parallelism
  – This is called Multiprocessing

• Some multiprocessors are actually uniprogrammed:
  – Multiple threads in one address space but one program at a time

A Typical Use Case

Client Browser
- process for each tab
- thread to render page
- GET in separate thread
- multiple outstanding GETs
- as they complete, render portion

Web Server
- fork process for each client connection
- thread to get request and issue response
- fork threads to read data, access DB, etc
- join and respond

Kernel Use Cases

• Thread for each user process
• Thread for sequence of steps in processing I/O
• Threads for device drivers
• …

Putting it Together: Process

(Unix) Process

Memory
Stack

I/O State
(e.g., file, socket contexts)

CPU state
(PC, SP, registers..)

Sequential stream of instructions

Resources

Stored in OS
Putting it Together: Processes

- Switch overhead: high
  - CPU state: low
  - Memory/IO state: high
- Process creation: high
- Protection
  - CPU: yes
  - Memory/IO: yes
- Sharing overhead: high
  (involves at least a context switch)

Putting it Together: Threads

- Switch overhead: medium
  - CPU state: low
- Thread creation: medium
- Protection
  - CPU: yes
  - Memory/IO: No
- Sharing overhead: low(ish)
  (thread switch overhead low)

Kernel versus User-Mode Threads

- We have been talking about kernel threads
  - Native threads supported directly by the kernel
  - Every thread can run or block independently
  - One process may have several threads waiting on different things
- Downside of kernel threads: a bit expensive
  - Need to make a crossing into kernel mode to schedule
- Lighter weight option: User Threads

User-Mode Threads

- Lighter weight option:
  - User program provides scheduler and thread package
  - May have several user threads per kernel thread
  - User threads may be scheduled non-preemptively relative to each other
    (only switch on yield())
  - Cheap
- Downside of user threads:
  - When one thread blocks on I/O, all threads block
  - Kernel cannot adjust scheduling among all threads
  - Option: Scheduler Activations
    » Have kernel inform user level when thread blocks…
Some Threading Models

Simple One-to-One Threading Model

Many-to-One

Many-to-Many

Threads in a Process

- Threads are useful at user-level: parallelism, hide I/O latency, interactivity
- Option A (early Java): user-level library, within a single-threaded process
  - Library does thread context switch
  - Kernel time slices between processes, e.g., on system call I/O
- Option B (SunOS, Linux/Unix variants): green threads
  - User-level library does thread multiplexing
- Option C (Windows): scheduler activations
  - Kernel allocates processors to user-level library
  - Thread library implements context switch
  - System call I/O that blocks triggers upcall
- Option D (Linux, MacOS, Windows): use kernel threads
  - System calls for thread fork, join, exit (and lock, unlock,...)
  - Kernel does context switching
  - Simple, but a lot of transitions between user and kernel mode

Putting it Together: Multi-Cores

- Switch overhead: low (only CPU state)
- Thread creation: low
- Protection
  - CPU: yes
  - Memory/I/O: No
- Sharing overhead: low (thread switch overhead low, may not need to switch at all!)

Recall: Simultaneous MultiThreading/Hyperthreading

- Hardware technique
  - Superscalar processors can execute multiple instructions that are independent
  - Hyperthreading duplicates register state to make a second “thread,” allowing more instructions to run
- Can schedule each thread as if were separate CPU
  - But, sub-linear speedup!
- Original called “Simultaneous Multithreading”
  - Intel, SPARC, Power (IBM)
  - A virtual core on AWS’ EC2 is basically a hyperthread
Putting it Together: Hyper-Threading

Process 1

- Switch overhead between hardware-threads: very-low (done in hardware)
- Contention for ALUs/FPUs may hurt performance

8 threads at a time

CPU state
CPU state
CPU state
CPU state

Thread
Mem.
Mem.
IO state
IO state

OS

Hardware threads (hyperthreading)

Process N

Classification

<table>
<thead>
<tr>
<th># threads Per AS</th>
<th># of addr spaces</th>
<th>One</th>
<th>Many</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td></td>
<td>MS/DOS, early Macintosh</td>
<td>Traditional UNIX</td>
</tr>
<tr>
<td>Many</td>
<td></td>
<td>Embedded systems (Geoworks, VxWorks, JavaOS, etc)</td>
<td>Mach, OS/2, Linux Windows 10, Win NT to XP, Solaris, HP-UX, OS X</td>
</tr>
</tbody>
</table>

- Most operating systems have either
  - One or many address spaces
  - One or many threads per address space

Summary

- Processes have two parts
  - Threads (Concurrency)
  - Address Spaces (Protection)

- Various textbooks talk about processes
  - When this concerns concurrency, really talking about thread portion of a process
  - When this concerns protection, talking about address space portion of a process

- Concurrent threads are a very useful abstraction
  - Allow transparent overlapping of computation and I/O
  - Allow use of parallel processing when available

- Concurrent threads introduce problems when accessing shared data
  - Programs must be insensitive to arbitrary interleavings
  - Without careful design, shared variables can become completely inconsistent