Recall: Lifecycle of a Process

- **new**: The process is being created
- **ready**: The process is waiting to run
- **running**: Instructions are being executed
- **waiting**: Process waiting for some event to occur
- **terminated**: The process has finished execution

Recall: Use of Threads

- Version of program with Threads (loose syntax):

```c
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

---

Internal Events: 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 many milliseconds

• If we make sure that external events occur frequently enough, can ensure dispatcher runs

Use of Timer Interrupt to Return Control

• Solution to our dispatcher problem
  – Use the timer interrupt to force scheduling decisions

  TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
  }

Thread Abstraction

• Illusion: Infinite number of processors
Thread Abstraction

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

Programmer vs. Processor View

Programmer's View

<table>
<thead>
<tr>
<th>Possible Execution</th>
<th>Possible Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>#2</td>
</tr>
<tr>
<td>x = x + 1;</td>
<td>x = x + 1;</td>
</tr>
<tr>
<td>y = y + x;</td>
<td>y = y + x;</td>
</tr>
<tr>
<td>z = x + 5y;</td>
<td>thread is suspended</td>
</tr>
<tr>
<td>other thread(s) run</td>
<td></td>
</tr>
<tr>
<td>thread is resumed</td>
<td></td>
</tr>
<tr>
<td>y = y + x</td>
<td></td>
</tr>
<tr>
<td>z = x + 5y</td>
<td></td>
</tr>
</tbody>
</table>

Possible Execution

<table>
<thead>
<tr>
<th>Program 1</th>
<th>Program 2</th>
<th>Program 3</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>z = x + 5y;</td>
<td>thread suspended</td>
<td>other thread(s) run</td>
</tr>
<tr>
<td>thread is resumed</td>
<td>thread is resumed</td>
<td>thread is resumed</td>
</tr>
<tr>
<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>z = x + 5y;</td>
</tr>
</tbody>
</table>
Possible Executions

Thread 1
Thread 2
Thread 3

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

Recall: Multithreaded stack switching

- Consider the following code blocks:
  ```
  proc A() {
    B();
  }
  proc B() {
    while(TRUE) {
      yield();
      run_new_thread
      switch
    }
  }
  ```
- Suppose we have 2 threads:
  - Threads S and T

Recall: Use of Timer Interrupt to Return Control

- Solution to our dispatcher problem
  - Use the timer interrupt to force scheduling decisions
  ```
  TimerInterrupt
  run_new_thread
  switch
  ```
- Timer Interrupt routine:
  ```
  TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
  }
  ```
**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
**What does ThreadRoot() look like?**

- **ThreadRoot()** is the root for the thread routine:
  ```c
  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

**Famous Quote wrt Scheduling: Dennis Ritchie**

Dennis Ritchie, Unix V6, slp.c:

“If the new process paused because it was swapped out, set the stack level to the last call to savu(u_ssav). This means that the return which is executed immediately after the call to aretu actually returns from the last routine which did the savu.”

“You are not expected to understand this.”

Source: Dennis Ritchie, Unix V6 slp.c (context-switching code) as per The Unix Heritage Society(tuhs.org); gif by Eddie Koehler. Included by Ali R. Butt in CS3204 from Virginia Tech

**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

**Administrivia**

- **Section assignments**
  - We tried to accommodate your needs, but had to balance both over-popular and under-popular sections

- 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

- **Project #1**: Started Monday!

- **HW1** due Monday
  - Submit via git
Examples multithreaded programs

- Embedded systems
  - Elevators, Planes, Medical systems, Wristwatches
  - 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

Sequential stream of instructions

Resources

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

CPU state (PC, SP, registers...)

• Switch overhead: high
  – Kernel entry: low (ish)
  – 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: Processes

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

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/IO: No
• Sharing overhead: low
  (thread switch overhead low, may not need to switch at all)
Putting it Together: Hyper-Threading

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

Supporting IT and MT Processes

User

System

Classification

- Real operating systems have either
  - One or many address spaces
  - One or many threads per address space
- Did Windows 95/98/ME have real memory protection?
  - No: Users could overwrite process tables/System DLLs
**Break**

---

**Perspective on ‘groking’ 162**

- Historically, OS was the most complex software
  - Concurrency, synchronization, processes, devices, communication, …
  - Core systems concepts developed there
- Today, many “applications” are complex software systems too
  - These concepts appear there
  - But they are realized out of the capabilities provided by the operating system
- Seek to understand how these capabilities are implemented upon the basic hardware
- See concepts multiple times from multiple perspectives
  - Lecture provides conceptual framework, integration, examples, …
  - Book provides a reference with some additional detail
  - Lots of other resources that you need to learn to use
    - man pages, google, reference manuals, includes (.h)
- Section, Homework and Project provides detail down to the actual code AND direct hands-on experience

---

**Operating System as Design**

- Processes
  - Thread(s) + address space
- Address Space
- Protection
- Dual Mode
- Interrupt handlers
  - Interrupts, exceptions, syscall
- File System
  - Integrates processes, users, cwd, protection
- Key Layers: OS Lib, Syscall, Subsystem, Driver
  - User handler on OS descriptors
- Process control
  - fork, wait, signal, exec
- Communication through sockets
  - Integrates processes, protection, file ops, concurrency
- Client-Server Protocol
- Concurrent Execution: Threads
- Scheduling
PintOS Projects

- Groups almost all formed
- Work as one!
- 10x homework
- P1: threads & scheduler
- P2: user process
- P3: file system

MT Kernel IT Process ala Pintos/x86

- Each user process/thread associated with a kernel thread, described by a 4KB page object containing TCB and kernel stack for the kernel thread

In User thread, w/ Kernel thread waiting

- x86 CPU holds interrupt SP in register
- During user thread execution, associated kernel thread is “standing by”

In Kernel thread

- Kernel threads execute with small stack in thread structure
- Scheduler selects among ready kernel and user threads
Thread Switch (switch.S)

- switch_threads: save regs on current small stack, change SP, return from destination threads call to switch_threads

Switch to Kernel Thread for Process

Kernel → User

- Interrupt return (iret) restores user stack and PL

User → Kernel

- Mechanism to resume k-thread goes through interrupt vector
User \rightarrow \text{Kernel via interrupt vector}

- Interrupt transfers control through the Interrupt Vector (IDT in x86)
- \text{iret} restores user stack and PL

Recall: cs61C THE STACK FRAME

Basic Structure of a Function

- **Prologue**
  - \text{entry:\ label:}
  - \text{add \$sp, \$sp, -framesize}
  - \text{sw \$ra, framesize-4(\$sp)}
  - save other regs if need be
- **Body** \ldots (call other functions...)
  - \text{lw \$ra, framesize-4(\$sp)}
  - restore other regs if need be
  - \text{add \$sp, \$sp, framesize}
  - \text{restore \$ra, if \$ra}

The Stack (review)

- Stack frame includes:
  - Return “instruction” address
  - Parameters
  - Space for other local variables
- Stack frames contiguous blocks of memory; stack pointer tells where bottom of stack frame is
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames

Pintos Interrupt Processing

intr\_NN\_stub()

- \text{intr\_entry:}
  - save regs as frame
  - set up kernel env.
  - call intr\_handler
- \text{intr\_exit:}
  - restore regs
  - \text{iret}

Pintos Interrupt Processing

intr\_NN\_stub()

- \text{intr\_entry:}
  - save regs as frame
  - set up kernel env.
  - call intr\_handler
- \text{intr\_exit:}
  - restore regs
  - \text{iret}

Wrapper for generic handler

intr\_entry: save regs as frame set up kernel env. call intr\_handler
intr\_exit: restore regs iret

Intr\_handler(*frame)

- classify
- dispatch
- ack IRQ
- maybe thread yield

intr\_NN\_stub()

- push 0x20 (int #)
  - jmp intr\_entry
- push 0x21 (int #)
  - jmp intr\_entry

intr\_entry:

- save regs as frame
- set up kernel env.
- call intr\_handler

intr\_exit:

- restore regs
- iret

Pintos intr\_handlers

interrupt.c

intr\_NN\_stub()

- push 0x20 (int #)
  - jmp intr\_entry
- push 0x21 (int #)
  - jmp intr\_entry

intr\_entry:

- save regs as frame
- set up kernel env.
- call intr\_handler

intr\_exit:

- restore regs
- iret

Pintos intr\_handlers

interrupt.c

intr\_NN\_stub()

- push 0x20 (int #)
  - jmp intr\_entry
- push 0x21 (int #)
  - jmp intr\_entry

intr\_entry:

- save regs as frame
- set up kernel env.
- call intr\_handler

intr\_exit:

- restore regs
- iret

Pintos intr\_handlers

interrupt.c

intr\_NN\_stub()

- push 0x20 (int #)
  - jmp intr\_entry
- push 0x21 (int #)
  - jmp intr\_entry

intr\_entry:

- save regs as frame
- set up kernel env.
- call intr\_handler

intr\_exit:

- restore regs
- iret

Pintos intr\_handlers

interrupt.c
Timer may trigger thread switch

- thread_tick
  - Updates thread counters
  - If quanta exhausted, sets yield flag
- thread_yield
  - On path to rtln from interrupt
  - Sets current thread back to READY
  - Pushes it back on ready_list
  - Calls schedule to select next thread to run upon iret
- Schedule
  - Selects next thread to run
  - Calls switch_threads to change regs to point to stack for thread to resume
  - Sets its status to RUNNING
  - If user thread, activates the process
  - Returns back to intr_handler

Pintos Return from Processing

Recall: Thread Abstraction

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

Multiprocessing vs Multiprogramming

- Remember Definitions:
  - Multiprocessing ≡ Multiple CPUs
  - Multiprogramming ≡ Multiple Jobs or Processes
  - Multithreading ≡ Multiple threads per Process
- What does it mean to run two threads “concurrently”?
  - Scheduler is free to run threads in any order and interleaving: FIFO, Random, ...
  - Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks
Correctness for systems with concurrent threads

- If dispatcher can schedule threads in any way, programs must work under all circumstances
  - Can you test for this?
  - How can you know if your program works?
- Independent Threads:
  - No state shared with other threads
  - Deterministic \(\Rightarrow\) Input state determines results
  - Reproducible \(\Rightarrow\) Can recreate Starting Conditions, I/O
  - Scheduling order doesn't matter (if `switch()` works!!!)
- Cooperating Threads:
  - Shared State between multiple threads
  - Non-deterministic
  - Non-reproducible
- Non-deterministic and Non-reproducible means that bugs can be intermittent
  - Sometimes called "Heisenbugs"

Interactions Complicate Debugging

- Is any program truly independent?
  - Every process shares the file system, OS resources, network, etc.
  - Extreme example: buggy device driver causes thread A to crash "independent thread" B
- You probably don't realize how much you depend on reproducibility:
  - Example: Evil C compiler
    » Modifies files behind your back by inserting errors into C program unless you insert debugging code
  - Example: Debugging statements can overrun stack
- Non-deterministic errors are really difficult to find
  - Example: Memory layout of kernel+user programs
    » Depends on scheduling, which depends on timer/other things
    » Original UNIX had a bunch of non-deterministic errors
  - Example: Something which does interesting I/O
    » User typing of letters used to help generate secure keys

Why allow cooperating threads?

- People cooperate; computers help/enhance people's lives, so computers must cooperate
  - By analogy, the non-reproducibility/non-determinism of people is a notable problem for "carefully laid plans"
- Advantage 1: Share resources
  - One computer, many users
  - One bank balance, many ATMs
    » What if ATMs were only updated at night?
  - Embedded systems (robot control: coordinate arm & hand)
- Advantage 2: Speedup
  - Overlap I/O and computation
    » Many different file systems do read-ahead
  - Multiprocessors – chop up program into parallel pieces
- Advantage 3: Modularity
  - More important than you might think
  - Chop large problem up into simpler pieces
    » To compile, for instance, `gcc calls cpp | cc1 | cc2 | as | ld`
    » Makes system easier to extend

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