Recall: Lifecycle of a Process

- As a process executes, it changes state:
  - 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):
  ```
  main() {
      ThreadFork(ComputePI("pi.txt"));
      ThreadFork(PrintClassList("clist.text"));
  }
  ```

- 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

Recall: Multithreaded stack switching

- Consider the following code blocks:
  ```
  proc A() {
      B();
  }
  ```

  ```
  proc B() {
      while(TRUE) {
          yield();
      }
  }
  ```

- Suppose we have 2 threads:
  - Threads S and T
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 many 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:

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

- I/O interrupt: same as timer interrupt except that DoHousekeeping() replaced by ServiceIO().
ThreadFork(): Create a New Thread

- **ThreadFork()** is a user-level procedure that creates a new thread and places it on ready queue
  - We called this CreateThread() earlier
- **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 \(\Rightarrow\) OS (asm) routine ThreadRoot()
- **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:
  ```
  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
Administrivia

- Group formation: should be completed by tonight!
  - Will handle stragglers tonight
- Section assignment
  - Form due tonight by midnight!
  - We will try to do final section assignment tomorrow
- 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: Released!
  - Technically starts today
  - Autograder should be up by tomorrow.
- HW1 due next Monday
  - Must be submitted via the recommended “push” mechanism through git

Famous Quote WRT Scheduling: Dennis Richie

Dennis Richie,
Unix V6, slp.c:

```
2236 /* If the new process paused because it was swapped out, set the stack level to the last call
2237 2238 # swapped out. Set the stack level to the last call
2239 2240 # tosav(u, savu). This means that the return
2241 2242 which is executed immediately after the call to aretu actually returns from the last routine which did
2243 2244 # the savu.
2245 2246 # the savu.
2247 2248 # You are not expected to understand this.
2249
```

"If the new process paused because it was swapped out, set the stack level to the last call to savu(u, savu). 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):

![Diagram of PCB and TCB relationships](image)

- 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, 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 uniprocessed:
  - 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

Some Numbers

- Frequency of performing context switches: 10-100ms
- Context switch time in Linux: 3-4 μsecs (Current Intel i7 & E5).
  - Thread switching faster than process switching (100 ns).
  - But switching across cores about 2x more expensive than within-core switching.
- Context switch time increases sharply with the size of the working set*, and can increase 100x or more.

* The working set is the subset of memory used by the 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.
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

(UIKit) Process

A(int tmp) {
  if (tmp<2)
    B();
    printf(tmp);
  }
B() {
  C();
  }
C() {
  A((2);
  }
A(1);
...

Putting it together: Processes

Process 1
Process 2
Process N

- 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: Threads

Process 1
Process N

- Switch overhead: medium
  - Kernel entry: low(ish)
  - CPU state: low
  - Memory/IO: No
- 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 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
  - User thread

- Many-to-One
  - User thread

- Many-to-Many
  - Kernel thread

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!)
Putting it together: Hyper-Threading

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

Supporting 1T and MT Processes

• 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
You are here... why?

- 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

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

- Compilers
- Word Processing
- Web Browsers
- Email
- Databases
- Web Servers

Application / Service

User

System

Portable OS Library

System Call

Interface

Portable OS Kernel

Platform support, Device Drivers

Hardware

x86

PowerPC

ARM

PCI

Graphics

Ethernet (10/100/1000)

802.11 a/b/g/n

SCSI IDE

Starting today: Pintos Projects

- Groups almost all formed
- Work as one!
- 10x homework
- P1: threads & scheduler
- P2: user process
- P3: file system
MT Kernel 1T 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/ k-thread waiting

- x86 proc holds interrupt SP high system level
- During user thread exec, associate kernel thread is "standing by"

In Kernel thread

- Kernel threads execute with small stack in thread struct
- 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

Kernel

User

Kernel->User

Kernel

User

Kernel

User

User->Kernel

Kernel

User

Kernel

User

User->Kernel via interrupt vector

Kernel

User

Kernel

User

• Switch to Kernel Thread for Process

Kernel

User

Kernel

User

Kernel

User

Kernel

User

• Mechanism to resume k-thread goes through interrupt vector

Kernel

User

Kernel

User

Kernel

User

Kernel

User

• Interrupt transfers control through the IV (IDT in x86)

Kernel

User

Kernel

User

Kernel

User

Kernel

User

• int 0 restores user stack and PL
Pintos Interrupt Processing

stubs

***

0x20

push 0x20 (int #)
jmp intr_entry

push 0x20 (int #)
jmp intr_entry

***

255

Hardware interrupt vector

Wrapper for generic handler

intr_entry:
save regs as frame
set up kernel env.
call intr_handler

intr_exit:
restore regs
iret

stubs.S

Pintos intr_handlers

Recall: cs61C THE STACK FRAME

Basic Structure of a Function

Prologue
entry_label:
add $sp,$sp,-framesize
sw $ra,framesize-4($sp)
if save $ra
save other regs if need be

Body . . . (call other functions...)

Epilogue
restore other regs if need be
lw $ra,framesize-4($sp)
add $sp,$sp,framesize
ejr $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

Timer may trigger thread switch

- thread_tick
  - Updates thread counters
  - If quanta exhausted, sets yield flag
- thread_yield
  - On path to rtn 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

- Hardware interrupt vector
- Interrupt vector
  - push 0x20 (int #)
  - jmp intr_entry
  - intr_entry:
    - save regs as frame
    - set up kernel env.
    - call intr_handler
- intr_exit:
  - restore regs
  - iret

Wrapper for generic handler
- Intr_handler(*frame)
  - classify
  - dispatch
  - ack IRQ
  - maybe thread yield
- Pintos intr_handlers
  - timer_intr(*frame)
  - tick++
  - thread_tick()
- timer.c

Resume Some Thread

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 ⇒ Input state determines results
  - Reproducible ⇒ 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”

Thread Abstraction

- programmer abstraction
- physical reality
- threads
- ready
- running

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

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

Multithreading

- Multiprocessing
- Multiprogramming

9/16/15 Kubiatowicz CS162 ©UCB Fall 2015
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

High-level Example: Web Server

- Server must handle many requests
- Non-cooperating version:
  ```c
  serverLoop() {
    con = AcceptCon();
    ProcessFork(ServiceWebPage(),con);
  }
  ```

Threaded Web Server

- Now, use a single process
- Multithreaded (cooperating) version:
  ```c
  serverLoop() {
    connection = AcceptCon();
    ThreadFork(ServiceWebPage(),connection);
  }
  ```

- Looks almost the same, but has many advantages:
  - Can share file caches kept in memory, results of CGI scripts, other things
  - Threads are much cheaper to create than processes, so this has a lower per-request overhead
- Question: would a user-level (say one-to-many) thread package make sense here?
  - When one request blocks on disk, all block...
- What about Denial of Service attacks or digg / Slash-dot effects?
Thread Pools

- Problem with previous version: Unbounded Threads
  - When website becomes too popular - throughput sinks
- Instead, allocate a bounded “pool” of worker threads, representing the maximum level of multiprogramming

```
master() {
    allocThreads(worker,queue);
    while(TRUE) {
        con=AcceptCon();
        Enqueue(queue,con);
        wakeUp(queue);
    }
}
worker(queue) {
    while(TRUE) {
        con=Dequeue(queue);
        if (con==null)
            sleepOn(queue);
        else
            ServiceWebPage(con);
    }
}
```

Summary (1 of 2)

- Processes have two parts
  - Threads (Concurrency)
  - Address Spaces (Protection)
- Concurrency accomplished by multiplexing CPU Time:
  - Unloading current thread (PC, registers)
  - Loading new thread (PC, registers)
  - Such context switching may be voluntary (yield(), I/O operations) or involuntary (timer, other interrupts)
- Protection accomplished restricting access:
  - Memory mapping isolates processes from each other
  - Dual-mode for isolating I/O, other resources
- 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

Summary (2 or 2)

- 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
- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives
- Showed how to protect a critical section with only atomic load and store \(\Rightarrow\) pretty complex!