CS 3204
Operating Systems
Lecture 3
Godmar Back

Announcements
• Submit prerequisite forms by end of class
• TA office hours posted
• Reading assignment is Chapter 1-3
  • Project 0 due Sep 3, 11:59pm
  • Project 1 help sessions next week, time TBA

Project 0
• Implement User-level Memory Allocator
  – Use address-ordered first-fit

Intermezzo
Just enough on concurrency to get through Project 0
A lot more later.

Concurrency
• Access to shared resources must be mediated
  – Specifically shared (non-stack) variables
• Will hear a lot more about this
• For now, simplest way to protection is mutual exclusion via locks (aka mutexes)
• For Project 0, concurrency is produced by using PThreads (POSIX Threads), so must use PThread’s mutexes.
  – Just an API, idea is the same everywhere

pthread_mutex example
/* Define a mutex and initialize it. */
static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
static int counter = 0; /* A global variable to protect. */
/* Function executed by each thread. */
static void increment(void *)
{
    int i;
    for (i = 0; i < 1000000; i++)
    {
        pthread_mutex_lock(&lock);
        counter++;        
        pthread_mutex_unlock(&lock);
    }
}
Core OS Functions

• Hardware abstraction through interfaces
• Protection:
  – Preemption
  – Interposition
  – Privilege (user/kernel mode)
• Resource Management
  – Virtualizing of resources
  – Scheduling of resources

OS and Performance

• Time spent inside OS code is wasted, from user’s point of view
  – In particular, applications don’t like it if OS does B in addition to A when they’re asking for A, only
  – Must minimize time spend in OS – how?
• Provide minimal abstractions
• Efficient data structures & algorithms
  – Example: O(1) schedulers
• Exploit application behavior
  – Caching, Replacement, Prefetching

Common Performance Tricks

• Caching
  – Pareto-Principle: 80% of time spent in 20% of the code; 20% of memory accessed 80% of the time.
  – Keep close what you predict you’ll need
  – Requires replacement policy to get rid of stuff you don’t
• Use information from past to predict future
  – Decide what to evict from cache: monitor uses, use least-recently-used policies (or better)
• Prefetch: Think ahead/speculate:
  – Application asks for A now, will it ask for A+1 next?

Final thought: OS aren’t perfect

• Still way too easy to crash an OS
• Example 1: “fork bomb”
  – main() { for(;;) fork(); } stills brings down most Unixes
• Example 2: livelock
  – Can be result of denial-of-service attack
  – OS spends 100% of time servicing (bogus) network requests
• Example 3: buffer overflows
  – Either inside OS, or in critical system components – read most recent Microsoft bulletin.

Things to get out of this class

• (Hopefully) deep understanding of OS
• Understanding of how OS interacts with hardware
• Understanding of how OS kernel interacts with applications
• Kernel Programming Experience
  – Applies to Linux, Windows, Mac OS-X
  – Debugging skills
• Experience with concurrent programming
  – Useful in many other contexts (Java, C#, …)

Processes & Threads
Overview

- Definitions
- How does OS execute processes?
  - How do kernel & processes interact
  - How does kernel switch between processes
  - How do interrupts fit in
- What's the difference between threads/processes
- Process States
- Priority Scheduling

Process

- These are all possible definitions:
  - A program in execution
  - An instance of a program running on a computer
  - Schedulable entity (*)
  - Unit of resource ownership
  - Unit of protection
  - Execution sequence (*) + current state (*) + set of resources

(*) can be said of threads as well

Alternative definition

- Thread:
  - Execution sequence + CPU state (registers + stack)
- Process:
  - n Threads + Resources shared by them (specifically: accessible heap memory, global variables, file descriptors, etc.)
- In most contemporary OS, n >= 1.
- In Pintos, n=1: a process is a thread – as in traditional Unix.
  - Following discussion applies to both threads & processes.

Context Switching

- Multiprogramming: switch to another process if current process is (momentarily) blocked
- Time-sharing: switch to another process periodically to make sure all process make equal progress
  - this switch is called a context switch.
- Must understand how it works
  - how it interacts with user/kernel mode switching
  - how it maintains the illusion of each process having the CPU to itself (process must not notice being switched in and out)

Aside: Kernel Threads

Most OS (including Pintos) support kernel threads that never run in user mode – in fact, in Project 1, all Pintos threads run like that.
Mode Switching

• User → Kernel mode
  – For reasons external or internal to CPU
• External (aka hardware) interrupt:
  – timer/clock chip, I/O device, network card, keyboard, mouse
  – asynchronous (with respect to the executing program)
• Internal interrupt (aka software interrupt, trap, or exception)
  – are synchronous
  – can be intended: for system call (process wants to enter kernel to obtain services)
  – or unintended (usually): fault/exception (division by zero, attempt to execute privileged instruction in user mode)
• Kernel → User mode switch on iret instruction

Context vs Mode Switching

• Mode switch guarantees kernel gains control when needed
  – To react to external events
  – To handle error situations
  – Entry into kernel is controlled
• Not all mode switches lead to context switches
  – Kernel code’s logic decides when – subject of scheduling
• Mode switch always hardware supported
  – Context switch (typically) not – this means many options for implementing it!