CSE 120—Principles of Operating Systems

July 11, 2006—Day 2

Processes
Threads

Instructor: Neil Rhodes

Processes/Threads

Multiprocessing:
- Multiprogramming: One CPU switching quickly between various programs
- Multiprocessor: Multiple CPUs each running a program simultaneously

All Software Organized as Processes
- Program + state information
  - Code + register values, stack, memory, …
- Analogy
  - Program = Class
  - Process = Object

Kernel switches between processes
- Cooperative: Each process yields when willing to give up control
- Preemptive: Timer goes off every so often. (How often?)
- Context Switch: switching from one process to another:
  - Swap registers
  - including Program Counter (PC) and Stack Pointer (SP)
  - Change memory map

Each process has the illusion that it runs continuously, independent of other processes
- At times, the programmer must be aware that it’s just an illusion

Advantages of Multiprogramming
- Easier to organize/abstract
  - Imagine if entirety of GNU/Linux was one giant program!
- Better Throughput
  - If process A is waiting for a read from disk to complete, process B can profitably use the CPU

Potential Problems
- Process that have real-time requirements (deadlines)
  - Video, Audio good examples

Memory Model of a Process

Text: The code

Data:
- The global variables
- The dynamically allocated memory (heap)

Stack
- activation records (stack frames). One per in-progress subroutine
  - Parameters
  - Saved program counter
  - Pointer to previous activation record
  - Any saved registers
  - Local variables

All are memory addresses in the process’s address space
How Processes are Created

The first process is created by hand (at boot time)

Remaining processes are spawned from existing processes
- Unix: fork()
- Windows: CreateProcess(…)

Creating Processes

When process A creates a process B
- Process A is the parent process
- Process B is the child process
- Two possibilities:
  - Process A continues running concurrently with B
  - Process A stops until B is finished
- Two possibilities with respect to address space
  - Child process has a duplicate of the parent process (same code, data, stack, etc.)
  - Child process has new program loaded into it.

Process Creation on Unix

```c
int main()
{
  int pid;
  pid = fork();
  if (pid < 0) {
    // error occurred
    exit(1);
  } else if (pid == 0) // child process
    exec("/bin/ls", "ls", NULL);
  else { // parent process
    wait(NULL); // wait until child completes
  }
}
```

Sharing the CPU

Three processes: their illusion

Three processes: the reality

Each process gets a slice of time (the quantum or timeslice)
How Processes are Destroyed

Normal exit—voluntary
  • return from main()
Error exit—voluntary
  • exit(error_code)
Fatal error—involuntary
  • Divide-by-zero
  • Segmentation violation
  • Bus error
  • ...
Killed by another process—involuntary
  • % kill -SIGKILL 1234 # sends SIGKILL signal (9) to process ID 1234
  • or, kill(SIGKILL, 1234)

How Time is Shared (Cooperative)

Cooperative multiprogramming
Process A calls Yield()
  • next process in list of waiting process now gets to run.

What does Yield do?
  • Must do Context Switch
  • Switch registers
  • Switch memory map

Yield must cause execution to begin in Process B
  • Where?

When Process B calls yield, Process A runs
  • From A’s point of view, it called Yield, which just returned (even though B executed in the meantime).

Details of Context Switch

How does Yield work?

One implementation
  • Yield()
    {
      volatile int magic = 0;
      Save context of current process (memory map, general registers, SP, PC)
      if (magic == 1)
        return;
      magic = 1;
      restore context of next process (memory map, general registers, SP, PC)
    }

Context-switch Example

A:
  int y = 1;
  void main() {
    Yield();
    y = 2;
  }

B:
  int x = 10;
  void main() {
    Yield();
    x = 11;
    Yield();
    x = 12;
  }

Text:
  Data:
  Stack:

Text:
  Data:
  Stack:

Saved PC
SP

Saved PC
SP
How is Time Shared? (Preemption)

Preemptive
- Advantages:
  - Doesn't require cooperation from all applications
  - Doesn't require special code in an application
- Disadvantages:
  - OS might not have as much information as application
  - Context switch occurs anytime, rather than at well-defined locations in code

How implemented?
- Kernel starts hardware timer
- If process finishes before timer expires, no problem
  - Makes blocking I/O call
  - Gives up CPU explicitly (yield)
  - Process destroyed
- Otherwise, timer interrupt occurs
  - Hardware dispatches to interrupt handler
  - Interrupt handler saves state of process A
  - Figures out which is the next eligible process
  - Restores state of process B
  - When process A runs again, it'll continue from where it left off

Possible States of a Process

Running
- Actually executing on the CPU
- Only one process can be in this state

Ready
- Ready to use the CPU (in a queue)
- What is the ordering of the queue?

Blocked
- Not able to use the CPU (waiting for something)

Context-switching/Scheduling

Context-switching
- Switching from one process to another
- Has overhead because
  - Requires system call
  - Change memory map
  - May flush cache

Scheduling
- Deciding which process should next run
- We’ll see more about this next class

Process table

Kernel stores information about each process in a table

Entries in a table: Process Control Block (PCB)
- Program Counter
- Stack Pointer
- Other registers
- Open files/sockets
- Scheduling information
- Memory map information
- Priority
- History
- Accounting information
  - CPU time spent by this process
  - CPU time spent by the kernel on behalf of this process
Kernel vs. Process

Kernel supports processes (is not a process itself)

Well-defined entry to the kernel
- Trap: system call by application
- Interrupt: generated by hardware (timer, I/O device, etc.)

How does kernel get control?
- Hardware (on trap or interrupt)
  - Switches from user mode to supervisor mode
  - Hardware determines new PC using a vector (Stored at predefined location)
    - 68K: 16 traps. x86: 256 software traps
- Software (trap or interrupt handler)
  - Stores state
  - Calls appropriate Kernel routine (based on specific trap or interrupt)
  - Restores state (if context switch, does not restore state).

User mode/supervisor mode
- Usually stored in Processor Status Word (PSW)
- In user mode, some instructions not allowed
  - Write PSW (Read PSW?)
  - Change memory map
  - Others that could breach security

Inter-Process Communication

Cooperating processes must communicate

Need to communicate
- Transfer information
- Synchronize

Different types of IPC
- Shared Memory
- Synchronization carried out via semaphores or monitors
- Message passing
- Remote Procedure Call (RPC)

Message Passing

One process sends a message, another receives it
- Variable-size / fixed-size message
- Direct or indirect communication
  - Directly specify other process
    - Symmetric: Send(P, message) / Receive(Q, &message)
    - Asymmetric: Send(P, message) / Receive(&processID, &message)
  - Indirect: use mailbox or port
    - send(MboxA, message) / Receive(MboxA, &message)
- Synchronous or asynchronous
  - Blocking send
  - Nonblocking send
  - Blocking receive
  - Nonblocking receive
  - Combo of blocking send and blocking receive yields a rendezvous
- Buffering: zero, bounded, or unbounded
  - Zero: sender blocks until receiver receives message
  - Bounded: If the buffer is full, sender blocks
  - Unbounded: sender never blocks

Remote Procedure Call (RPC)

Make a procedure call that invokes procedure in remote process
- Could be process on the same machine, or process across a network

How it works:
- Local stub marshals parameters
- May need to convert to machine-independent format
- Sends message with procedure ID and parameters
- Waits for receipt of message with return result

Remote Method Invocation (RMI)
- Used in Java as a way to cause a method to be executed on a remote object

Corba (Common Object Request Broker Architecture)
- Language- and platform-neutral way to execute methods on objects
Threads

Threads are like lightweight processes

Processes have their own:
- Address space
- Open Files

Threads are within a given process
- Share the same address space (memory)
- Share open files
- Share other resources

Can be cooperative or preemptive

User-level/Kernel-level threads

User-level (sometimes called fibers)
- Kernel knows nothing about threads
- Advantages:
  - Can be added to any OS
  - Lightweight context-switch between threads (no kernel call)
  - Ability to have process-specific scheduling algorithm
  - Context-switch happens at well-defined times; can reduce race conditions
- Disadvantages
  - If a thread blocks on an I/O call, all threads in the process are blocked
  - Must wait on a thread to yield before another thread in the process runs
    - Unless roll own preemption with alarms

Kernel-level
- Kernel knows about threads, will schedule/context-switch, etc.
- Advantages:
  - Can schedule preemptively within a process
  - Can schedule another thread within a process if one thread blocks
- Disadvantages:
  - Heavier-weight context switch
  - If many threads, kernel must store lots of state information

Thread table
- Per-thread saved information:
  - Registers (including PC)
  - Stack

Multithreading Models

Many-to-one

One-to-one

Many-to-many

Common Thread Primitives

thread_id = Thread_create(…, procedure)
- New thread executes procedure

Thread_join(thread_id)
- blocks calling thread until thread with given ID exits

Thread_exit(exit_value)
- Kills current thread

Thread_yield()
- Switches execution from this thread to another thread
Multithreaded Process

OS with kernel-level threads:
- Process starts with one thread

If process spawns another thread, it is now multithreaded.

Issues of synchronization between threads/process and Inter-Process Communication (IPC)