CMSC 412
Fall 2004
Processes and Threads

Announcements

- Project #1
  - Due Friday

- Reading
  - Chapter 4 (parts), 5 (parts)
  - Chapter 7 (for Monday)
Processes

- What is a process?
  - A program in execution
    - Either sequentially or with multiple “threads of control.”

- What’s not a process?
  - A program on a disk - a process is an active object, but a program is just a file

Computation Abstractions

A dual-processor computer
Processes vs. Threads

Processes do not share data

int x;
foo() {
    ...x...
}

Threads share data within a process

int x;
foo() {
    ...x...
}

More on threads later …

Process State

- Processes switch between different states based on internal and external events
- Each process is in exactly one state at a time
- Typical States of Processes (varies with OS)
  - New: just been created
  - Running: instructions are being executed
    - only one process per processor may be running
  - Waiting: waiting for an event to occur
    - examples: I/O events, signals
  - Ready: waiting to be assigned the CPU
  - Terminated: finished execution
Components of a Process

- **Memory Segments**
  - Program - often called the text segment
  - Data - global variables
  - Stack - contains activation records
  - Heap - contains dynamically-allocated data

- **Processor Registers**
  - Control registers
    - program counter - next instruction to execute
    - stack pointer
    - processor status word (from cmp instructions)
  - General purpose registers
  - Floating-point registers
Scheduling

- OS must decide when a process is allowed to run; i.e. it must schedule processes
- Long-term scheduler (or job scheduler) - selects which processes should be brought into the ready queue.
- Short-term scheduler (or CPU scheduler) - selects which process should be executed next and allocates CPU. A.k.a the dispatcher.

Schedulers

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast).
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow).
- The long-term scheduler controls the degree of multiprogramming.
- Processes can be described as either:
  - I/O-bound process - spends more time doing I/O than computations, many short CPU bursts.
  - CPU-bound process - spends more time doing computations; few very long CPU bursts.
Scheduling Policy

- How should the scheduler choose a process to run?
- Can simply pick the first item in the queue
  - called round-robin scheduling
  - is round-robin scheduling fair?
- Various scheduling criteria
  - Process class (I/O bound vs. CPU bound)
  - Priority
  - Resources consumed
  - Etc.

Scheduling Implementation

- Use alarm interrupts to switch between processes
  - when time is up, a process is put back on the end of the ready queue
  - frequency of these interrupts (a.k.a. the quantum) is an important parameter
    - typically 3-10ms on modern systems
    - need to balance overhead of switching vs. responsiveness
Context Switch

- When the OS switches a CPU to another process, it is called a context switch.
- The system must save the state of the old process and load the saved state for the new process.
- Context-switch time is overhead; the system does no useful work while switching.
  - Total time depends on hardware support.
  - Writing context-switch routines almost always requires some assembly language.

OS Process Control Block

- Stores all of the information about a process
- PCB contains
  - Process state: new, ready, etc.
  - (saved) processor registers
  - Memory Management Information
    - page tables, limit registers for segments
  - CPU scheduling information
    - process priority; pointers to process queues
  - Accounting information
    - time used (and limits); files used; program owner; parent process id
  - I/O status information
    - list of open files; pending I/O operations
Sample PCB

<table>
<thead>
<tr>
<th>pointer</th>
<th>process state</th>
</tr>
</thead>
<tbody>
<tr>
<td>process number</td>
<td></td>
</tr>
<tr>
<td>program counter</td>
<td></td>
</tr>
<tr>
<td>registers</td>
<td></td>
</tr>
<tr>
<td>memory limits</td>
<td></td>
</tr>
<tr>
<td>list of open files</td>
<td></td>
</tr>
</tbody>
</table>

Context Switch using PCBs

Diagram showing the process of context switch using PCBs. The diagram illustrates the process of saving and reloading state from PCBs between two processes, $P_0$ and $P_1$. The states include executing, idle, and interrupt or system call.
GeekOS PCB (part I)

- **struct Kernel_Thread**, contains
  - Process Identifier (PID)
  - Scheduling criteria (priority)
  - Accounting info (CPU clock ticks)
  - Kernel stack pointer
    - Context-switch information stored here; i.e. general-purpose registers, program counter, etc.
- User processes in GeekOS are a special kind of kernel thread

GeekOS PCB (part II)

- **struct User_Context**, contains
  - Pointer to process (physical) memory
    - Includes all the segments you set up in Project 1; i.e. code segment, data segment, etc.
  - Pointers to initial program entry point, initial argument, and initial stack.
  - Information for managing address space protection
    - Ix86 descriptor tables and selectors
- You’ll set this up in Project 2
Storing PCBs

- Track which processes are in which states
  - Collection of PCBs is called a process table
- How to store the process table?
- First Option:

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ready</td>
<td>Waiting</td>
<td>New</td>
<td>Term</td>
<td>Waiting</td>
<td>Ready</td>
</tr>
</tbody>
</table>

- Simple, but slow to find processes
- Also need additional datastructures for fairness

Queues of Processes

- Store processes in state-based queues

[Diagram showing queues of processes with processes P1, P2, P3, P4, P5, P6 in different states (Ready, Waiting, New, Term, Waiting, Ready) and labeled queues (Queue Ready, Disk Queue, Network Queue)]
Forking a New Process

- New process is called the child; the process that created it is the parent
- Create a PCB for the new process
  - copy most entries from the parent
  - clear accounting fields
  - buffer pending I/O
  - allocate a PID for the new process
Forking a New Process

- Allocate memory for it
  - Might copy all of the parents’ segments
  - Text segment could be shared
    - rarely changes
  - Use memory mapping hardware to help
    - will talk more about this in the memory management part of the class
- Add it to the ready queue

Process Termination

- Process can terminate itself
  - via the `exit` system call
- One process can terminate another process
  - use the `kill` system call
  - can any process kill any other process?
    - No, that would be bad.
    - Normally an ancestor can terminate a descendant
- OS kernel can terminate a process
  - exceeds resource limits
  - tries to perform an illegal operation
Orphan Processes

- What if a parent terminates before the child?
  - the child called an orphan process
    - in UNIX - becomes child of the root process
    - in VMS - terminated

UNIX example

- Terminated process
  - signals parent of its death (SIGCHILD)
  - is called a zombie in UNIX
  - remains around waiting to be reclaimed

- Parent process
  - wait system call retrieves info about the dead process
    - exit status
    - accounting information
  - signal handler is generally called the reaper
    - since its job is to collect the dead processes
Kernel Mode and User Mode

Kernel Threads:
- Each has own stack (separate from user mode)
- Share heap with other kernel threads
- Run same program (kernel) as other kernel threads

Threads

- Processes can be a heavy (expensive) object
- Threads are like processes but generally a collection of threads will share
  - memory (except stack)
  - open files (and buffered data)
  - signals
- Can be user or system level
  - user level: kernel sees one process
    - easy to implement by users
    - I/O management is difficult
    - in an multi-processor can’t get parallelism
  - system level: kernel schedules threads
### Single and Multithreaded Processes

- **Kernel Threads** – Threads that run with kernel privileges
- **User Threads** – Threads running in user space
  - Kernel may not be aware of them
- **Processes** – An execution context with an address space
  - Visible to and scheduled by the kernel
- **Light-Weight Processes** – An execution context sharing an address space
  - Visible to and scheduled by the kernel

---

### Execution Abstractions
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One Model
One-to-one Model

Many-to-Many Model
Why multiple threads?

- Performance:
  - Parallelism on multiprocessors
  - Concurrency of computation and I/O
- Can easily express some programming paradigms
  - Event processing
  - Simulations
- Keep computations separate

Why not multiple threads?

- Complexity:
  - Dealing with safety, liveness, composition
- Overhead
  - Higher resource usage
- Check out CMSC 433 for lots of information on threads and their alternatives!