Lecture Topics

- Today: Processes
  (Stallings, chapter 3.1-3.6)

- Next: Threads
  (Stallings, chapter 4.1-4.3, 4.6)

Announcements

- Make tutorial
- Self-Study Exercise #4
- Project #2 (due 9/20)
- Project #3 (due 9/27)
Processes

- All multiprogramming operating systems are built around the concept of the process: an active program and related resources

- Main issues:
  - different process states, transitions among them
  - process description (data structures maintained)
  - process control (switching between processes)

Processes

- A process consists of (at least):
  - code and data for executing program
  - program counter (PC) and status register (PSW)
  - general-purpose registers
  - set of OS resources (open files, signals received, network connections, …)

- In other words, everything the OS needs to run the program -- or to re-start it, if it’s interrupted at some point
Five-State Process Model

Process Model: Queues
Process Control Block

All info about process
- Identifier
- State
- Priority
- Program counter
- Memory pointers
- Context data
- I/O status info
- Accounting info

PCB assigned to process when created
OS maintains info in PCB while process is in existence
OS moves PCBs between queues to manage system
Refinement: Seven-State Process Model

Suspended Processes

- Temporarily copy some processes from RAM to disk
- Two new states:
  - Blocked/Suspend
  - Ready/Suspend
- OS control structures (in kernel memory) remain in RAM -- never copied to disk
Suspended Processes (2)

- Primary motivation: free up part of RAM so that it can be used by a different process
  - If too many processes are in RAM at once, system performance can be degraded
  - Take some RAM away from one process and give it to another
  - Process which gives up RAM is no longer able to execute – it is suspended

Reasons for Process Suspension

- Swapping: OS needs to free up memory
- Interactive user request (ex: debugging)
- Parent process request (ex: suspend child)
- Timing (ex: periodic system process)
- Other OS reasons (ex: misbehaving process)
Operating System Control Structures

- Information about the current status of each process and resource
- Table for each resource the OS manages
- Tables in kernel memory

Process Table

- List of processes currently in the system
- Table entry contains pointer to *process image*
  - user text segment (machine language)
  - user data segment (static data objects)
  - user stack segment (dynamic data objects)
Typical PCB Contents

- **Process Identifiers**
  - ID of this process
  - ID of the parent process
  - User ID

- **Processor State Information**
  - User-visible registers
  - Control and status registers

- **Process Control Information**
  - Supplemental info maintained by OS

Processor State Information

- All information needed to resume execution of the process

- Contents of user-visible registers

- Contents of control and status registers
  - program counter
  - program state word (PSW)
  - condition codes (if present)
PSW: x86 EFLAGS Register

ID = Identification flag
VIP = Virtual interrupt pending
VIF = Virtual interrupt flag
AC = Alignment check
VM = Virtual 8086 mode
RF = Resume flag
NT = Nested task flag
IOPL = I/O privilege level
OF = Overflow flag
DF = Direction flag
IF = Interrupt enable flag
TF = Trap flag
SF = Sign flag
ZF = Zero flag
AF = Auxiliary carry flag
PF = Parity flag
CF = Carry flag

PSW: SPARC Processor Status Register

PIL: processor interrupt level
S: supervisor mode? (vs. user mode)
PS: previous S bit
ET: traps enabled?
Process Control Information

- Scheduling and state information
  - Process state: running, ready, blocked, etc.
  - Priority: one or more fields to describe the scheduling priority of the process.
  - Scheduling-related information: depends on the specific scheduling algorithm used (ex: the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running).
  - Event: event the process is awaiting

- Inter-process Communication: flags, signals, and messages may be associated with communication between two independent processes.

- Process Privileges: access restrictions for process

- Memory Management: pointers to page tables that describe the virtual memory assigned to process.

- Resource Ownership and Utilization: resources controlled by process
Memory Tables

- Allocation of main memory to processes
- Allocation of secondary memory to processes
- Protection attributes for access to shared memory regions
- Information needed to manage virtual memory

I/O Tables

- I/O device is available or assigned
- Status of I/O operation
- Location in main memory being used as the source or destination of the I/O transfer
File Tables

- Existence of files
- Location on secondary memory
- Current status
- Attributes
- Sometimes this information is maintained by a file management system

Modes of Execution

- User mode
  - Less-privileged mode
  - User programs execute in this mode
- Kernel mode (control mode, system mode)
  - More-privileged mode
  - OS kernel operates in this mode
- Bit in status register specifies current mode
# Mode Switch

- **User mode to kernel mode**
  - **interrupt:** during execution of current instruction; reaction to asynchronous external event
  - **trap:** caused by execution of current instruction; reaction to exceptional condition
  - **system call:** explicit request by current instruction; request service from OS

- **Kernel mode to user mode**

---

# Process Switch

- **Process currently in Running state**
  - save context (PC, other registers)
  - move PCB to appropriate queue
  - update PCB (state, accounting info)

- **Process selected by OS**
  - update PCB (state, accounting info)
  - move PCB out of Ready queue
  - restore context (PC, registers)
Mode and Process Switch

Approximate timing on "arctic.cse.msu.edu":

- Mode switch: 51-58 ns
- Process switch: 2200-2800 ns

Approximate timing on "cse410.cse.msu.edu":

- Mode switch: 235-240 ns
- Process switch: 5500-6000 ns

Execution of the Operating System

- Non-process Kernel
  - Kernel executes as a separate entity in privileged mode (not a process)
  - Typical of older batch-oriented operating systems
Execution of the Operating System (2)

- Execution Within User Processes
  - Kernel executes within context of a user process
  - Process executes in privileged mode when executing operating system code
  - Process switching code is separate

Execution of the Operating System (3)

- Process-Based Operating System
  - Implement OS as a collection of system processes
  - Useful in multi-processor or multi-computer environment
  - Process switching code is separate
UNIX Process Management

- Two categories of processes
  - System processes
    - run in kernel mode for housekeeping functions (memory allocation, process swapping, etc)
  - User processes
    - run in user mode for user programs
    - run in kernel mode for system calls, traps, interrupts
UNIX Process States

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Running</td>
<td>Executing in user mode.</td>
</tr>
<tr>
<td>Kernel Running</td>
<td>Executing in kernel mode.</td>
</tr>
<tr>
<td>Ready to Run, in Memory</td>
<td>Ready to run as soon as the kernel schedules it.</td>
</tr>
<tr>
<td>Asleep in Memory</td>
<td>Unable to execute until an event occurs; process is in main memory (a blocked state).</td>
</tr>
<tr>
<td>Ready to Run, Swapped</td>
<td>Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute.</td>
</tr>
<tr>
<td>Sleeping, Swapped</td>
<td>The process is awaiting an event and has been swapped to secondary storage (a blocked state).</td>
</tr>
<tr>
<td>Preempted</td>
<td>Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.</td>
</tr>
<tr>
<td>Created</td>
<td>Process is newly created and not yet ready to run.</td>
</tr>
<tr>
<td>Zombie</td>
<td>Process no longer exists, but it leaves a record for its parent process to collect.</td>
</tr>
</tbody>
</table>

UNIX Process States (2)

- Similar to seven state model
- Two running states (User and Kernel): transitions to other states (blocked, ready) must come from Kernel Running
- Sleeping states (in memory/ swapped) correspond to blocked states
- Preempted state is distinguished from Ready state (but they form one queue)
- Preemption can occur only when a process is about to move from kernel to user mode
UNIX Process Image

<table>
<thead>
<tr>
<th>User-Level Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Text</td>
</tr>
<tr>
<td>Process Data</td>
</tr>
<tr>
<td>User Stack</td>
</tr>
<tr>
<td>Shared Memory</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Counter</td>
</tr>
<tr>
<td>Processor Status Register</td>
</tr>
<tr>
<td>Stack Pointer</td>
</tr>
<tr>
<td>General-Purpose Registers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System-Level Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Table Entry</td>
</tr>
<tr>
<td>U (user) Area</td>
</tr>
<tr>
<td>Per Process Region Table</td>
</tr>
<tr>
<td>Kernel Stack</td>
</tr>
</tbody>
</table>

UNIX Process Generation

- Every process (except process 0) is created by the fork() system call
  - fork() allocates entry in process table and assigns a unique PID to the child process
  - child gets a copy of the process image of the parent; both child and parent are executing the same code after fork()
  - fork() returns the PID of the child process to the parent process and returns 0 to the child process
UNIX System Processes

- Process 0 is created at boot time and becomes the "swapper" process after forking process 1 (the "systemd" process) and process 2 (the "kthreadd" process).

- The "kthreadd" process is the ancestor of all kernel processes.

- The "systemd" process is the ancestor of all user processes.

UNIX Process Creation

```c
int pid = fork();

if (pid < 0)
{
    // Error -- fork failed
}
else if (pid > 0)
{
    // Parent process (pid contains child's PID)
}
else
{
    // Child process
}
UNIX Process Creation (2)

- When the child process terminates, the parent process is notified
- Typically, the parent process uses the wait() system call to voluntarily block until the child process terminates
- If the parent process terminates before the child process, the child becomes an orphan process and is adopted by process 1

UNIX Process Specialization

- The fork() system call creates a duplicate copy of the process image of the parent; both child and parent are executing the same code after fork()
- The exec() family of system calls is used to replace the process image of one process (usually the child) with a different process image (a different program)
UNIX Process Specialization (2)

```c
int pid = fork();

if (pid < 0)
{
    // Error -- fork failed
}
else if (pid > 0)
{
    // Parent process (pid contains child's PID)
}
else
{
    flag = execvp( "ls", args );
}
```

Examples

- Use of "ps" command
- Use of "fork" and related functions