Operating System Overview and Processes
Outline

• Overview of what operating systems do
• What a process is
• How a process is managed
What an Operating Systems Does

• I/O Functions
• Process functions
• Memory functions
• Secondary storage functions
• User interface functions
• Other desirable features
I/O Functions

• Read, write data
  • Polling
  • Interrupts, traps

• Byte-oriented devices

• Direct memory access (DMA)
Process Functions

• Create, delete process

• Process status: get information about . . .
  • Resources used
  • Time used
  • UID/owner of process
  • GID/group of process

• Process control
  • Limit resources, increase resources
  • Control access to files
  • Handle interrupts and traps
Memory Functions

- Allocate, deallocate memory
- Share memory among processes
- Translate virtual addresses into physical addresses
- Manage pages, segments, and variants
- Map files into memory
- Protext parts of memory from being read, written, or executed
Secondary Storage Functions

• Manage space on devices
• Map file addresses into secondary storage addresses and *vice versa*
• Scheduling reads, writes to secondary storage
• Manage writing directly into, from main memory locations
User Interface Functions

• Enable users to run processes easily
• Enable users to manage files easily
• Allow easy configuration and control of system by administrators
Other Desirable Features

• Efficiency
• Reliability
• Maintainability
• No larger than necessary
Process

• Obvious definition: A running program

• But more complicated when on a multiprogrammed system
  • As process execution is interleaved, the process does not execute continuously
  • CPU is virtualized, so each process thinks it is running continuously
  • Processes are scheduled by the job/process scheduler

• In this sense, process is an abstraction
Process State

• Running: process is executing
  • Part of it lies in memory
  • The memory that the process can address is called the process’ *address space*
  • Special registers:
    • PC, program counter, gives address of next instruction to be executed
    • SP, stack pointer, points to the word beyond the stack top
    • Frame pointer used to manage the stack for function arguments and (local) variables

• Blocked: process is not running but is waiting on some event

• Ready: process ready to run but the operating system has chosen not to run it for some reason
Process State Diagram

- **Running**
  - scheduler picks another process
  - blocks for input

- **Blocked**
  - input becomes available

- **Ready**
  - scheduler picks this process
### Examples

<table>
<thead>
<tr>
<th>time</th>
<th>process A</th>
<th>process B</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Running</td>
<td>Ready</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Running</td>
<td>Ready</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Running</td>
<td>Ready</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Running</td>
<td>Ready</td>
<td>A done</td>
</tr>
<tr>
<td>5</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Running</td>
<td></td>
<td>B done</td>
</tr>
</tbody>
</table>

### Table

<table>
<thead>
<tr>
<th>time</th>
<th>process A</th>
<th>process B</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Running</td>
<td>Ready</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Running</td>
<td>Ready</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Running</td>
<td>Ready</td>
<td>A initiates I/O</td>
</tr>
<tr>
<td>4</td>
<td>Blocked</td>
<td>Running</td>
<td>A blocks, B runs</td>
</tr>
<tr>
<td>5</td>
<td>Blocked</td>
<td>Running</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Blocked</td>
<td>Running</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Ready</td>
<td>Running</td>
<td>A’s I/O done</td>
</tr>
<tr>
<td>8</td>
<td>Ready</td>
<td>Running</td>
<td>B now done</td>
</tr>
<tr>
<td>9</td>
<td>Running</td>
<td>Running</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Process APIs

• Create: initiate a new process
  • When command entered at shell, shell creates (spawns) new process of that program executing

• Destroy: terminate a process
  • Use this to terminate (kill) an existing process

• Wait: block for some event
  • Useful when the process has to stop for some event

• Status: get status information about a process

• Other controls: varied ways to interact with the process
  • Most systems allow a user to suspend, resume a process

Details of Process Creation

- Operating system loads code, any initialized data into memory
  - This will become the process, so it is assigned an address space
  - Only parts of the program may be loaded at this time; the rest will be moved into memory when needed, and moved out when no longer needed
    - Paging, segmentation, swapping

- Operating system allocates memory for stack
  - On Linux, program arguments and environmental variables are put onto the stack
  - Stack can grow or shrink

- Operating system allocates space for heap
  - Uninitialized memory, accessed by the process allocating itself memory
  - More heap space can be allocated if needed
Details of Process Creation

• Operating system does otherinitializations
  • Opens input, output files and assigns them to the process
    • On Linux, this is done by assigning file descriptors

• Operating system then marks the process as ready to run, putting it in the READYstate
  • It may execute it immediately, changing the state to RUNNING
Data Structures for Process Management

• Process information kept in *process table*
  • May be fixed size or be able to grow

• Process table entry contains information about a process
  • Often split into 2 parts, one remaining always in memory, the other part (*not* part of the process table) able to be moved out of memory
Process Table Entry Example

• Example is from UNIX V6
  • See the handout “Process Information in UNIX V6”

• Modern systems may have more complex entries, but the idea is the same
Process Execution

• Direct execution
  • Kernel runs process on CPU, without allowing any other process to run

• Limited direct execution
  • Kernel runs process, but on trap or interrupt, kernel takes control and allows another process to run
  • Raises issue of switching between process
## Direct Execution

<table>
<thead>
<tr>
<th>operating system</th>
<th>program/process</th>
</tr>
</thead>
<tbody>
<tr>
<td>create entry for process in process table</td>
<td></td>
</tr>
<tr>
<td>allocate memory for program</td>
<td></td>
</tr>
<tr>
<td>load program into memory</td>
<td></td>
</tr>
<tr>
<td>setup stack (including arguments, environment, etc.)</td>
<td></td>
</tr>
<tr>
<td>clear registers</td>
<td></td>
</tr>
<tr>
<td>call start of program (e.g., call <code>main()</code>)</td>
<td>run <code>main()</code></td>
</tr>
<tr>
<td></td>
<td>execute <code>return</code> from <code>main()</code></td>
</tr>
<tr>
<td>free memory of process</td>
<td></td>
</tr>
<tr>
<td>delete process table entry</td>
<td></td>
</tr>
</tbody>
</table>
Problems

• How does operating system prevent program from doing something we don’t want, while impacting performance minimally (if at all)?

• How does the operating system stop a program and switch to another process?
  • In other words, how do we do time-sharing and multiprogramming?
Limited Direct Execution

• Make some instructions privileged
  • These are instructions enabling interference with, or directly interfering with, other processes or system management functions
  • This way, the process carries out its need but does not have control of the complete system

• To do this, introduce modes or levels of privilege
  • Kernel mode: privileged instructions can only be executed in this mode
  • User mode: normal, processes run in this mode but cannot execute privileged instructions
System Calls

• Cause a trap

• When a trap or interrupt (of any kind) occurs:
  • Do a context switch to kernel
  • Service the trap or the interrupt
  • Select the next process to run
  • Do a context switch to that process

• So at boot time, the trap table/interrupt vector must be initialized
Context Switch

• PC, processor status word, registers, pushed onto a small kernel stack allocated for the process

• Jump to routine in kernel indicated by trap table/interrupt vector

• Kernel services the trap/interrupt

• Kernel selects the next process to run
  • It may be a different one

• Kernel pops the information from the kernel stack and restores them to the registers, processor status word, and PC

• PC popped last, as when it is restored, process restarts
Context Switch Example

• Example is from XINU on an LSI-11 system
  • See the handout “Context Switch Routine for XINU System on LSI-11”
  • LSI-11 has same instruction set as PDP-11

• Modern systems may have more complex entries, but the idea is the same
Limited Direct Execution

<table>
<thead>
<tr>
<th>operating system at boot (kernel mode)</th>
<th>hardware</th>
<th>program/process (user mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize interrupt/trap vectors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>create entry for process in process table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>allocate memory for program</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load program into memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>setup stack (including arguments, environment, etc.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fill kernel stack with registers, PC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return-from-trap</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>remember address of syscall handler</td>
<td></td>
</tr>
<tr>
<td></td>
<td>restore registers from kernel stack</td>
<td></td>
</tr>
</tbody>
</table>
## Limited Direct Execution

<table>
<thead>
<tr>
<th>operating system at boot (kernel mode)</th>
<th>hardware</th>
<th>program/process (user mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>change to user mode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>jump to <code>main()</code></td>
<td>run <code>main()</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>system call causes trap into OS</td>
</tr>
<tr>
<td></td>
<td>save registers to kernel stack</td>
<td></td>
</tr>
<tr>
<td></td>
<td>change to kernel mode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>jump to trap handler</td>
<td></td>
</tr>
<tr>
<td>handle trap, ie system call</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return-from-trap</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>restore registers from kernel stack</td>
<td></td>
</tr>
</tbody>
</table>
## Limited Direct Execution

<table>
<thead>
<tr>
<th>operating system at boot (kernel mode)</th>
<th>hardware</th>
<th>program/process (user mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>change to user mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jump to PC after trap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return from <code>main()</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>causes trap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>free memory of process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete process table entry</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>