The Process Abstraction

CMPU 334 – Operating Systems
Jason Waterman
How to Provide the Illusion of Many CPUs?

• Goal: run N processes at once even though there are M CPUs
  • N >> M

• CPU virtualizing
  • The OS can promote the **illusion** that many virtual CPUs exist
  • One **isolated machine** for each program

• Timesharing
  • Running one program, then stopping it and running another
  • The potential cost is **performance**

• What are the benefits?
  • Ease of use for the programmer
  • Protection – program runs on a restricted machine
A Process

• A process is OS’s abstraction of a **running program**

• What constitutes a process?
  • Memory (address space)
    • Instructions
    • Data
  • Registers (state of the processor)
    • General purpose registers
    • Program counter (PC)
    • Stack pointer (SP)
  • I/O Information
    • List of files a process currently has open
Process API

• These APIs are available on any modern OS
  • Create
    • Create a new process to run a program
  • Destroy
    • Halt a runaway process
  • Wait
    • Wait for a process to stop running
  • Miscellaneous Control
    • Suspend
    • Resume
  • Status
    • Get some status information about a process
    • How long it has been running
    • What state is it in
Process Creation

1. Load a program code into memory, the address space of the process
   • Programs reside on a disk in an executable format (e.g., ELF)

2. The program’s run-time stack is allocated
   • Stack is used for local variables, function parameters, return address
   • Initialize the stack with arguments
     • argc and argv array of main() function
Process Creation (Cont.)

3. The program’s **heap** is created
   - Used for explicitly requested dynamically allocated data
   - `malloc(); free()`

4. The OS does some other **initialization**
   - I/O setup (stdin, stdout, stderr)

5. **Start** the program running at the entry point `main()`
   - The OS transfers control of the CPU to the newly-created process
Process States (simplified)

- A process can be in one of three states
  - **Running**
    - A process is running on the CPU
  - **Ready**
    - A process is ready to run but for some reason the OS has chosen not to run it at this given moment
  - **Blocked**
    - A process has performed some kind of operation that it is waiting on
      - E.g., an disk request
Process Data Structures

• The OS has some key data structures that track various pieces of information
  • Process list
    • Ready processes
    • Blocked processes
    • Current running process
  • Register context
    • A copy of all the registers for a process

• The Process Control Block (PCB)
  • A structure that contains information about each process
Process Creation

• We talked about process creation in general terms
• Now let’s discuss process creation in UNIX systems
  • fork() – Makes a copy of the currently running process
  • exec() – Replaces a process with a different program
  • wait() – Wait for a child process to finish

• Questions to think about
  • What interfaces should the OS present for process creation and control?
  • How should these interfaces be designed to enable ease of use as well as utility?
The fork() example

• Create a new process
  • The newly-created process has its own copy of the address space, registers, and PC
The \texttt{wait()} example

- Wait for a child to finish
  - Returns the pid of the child that ended
  - If you care about the exit status of the process, pass in an \texttt{int}\,*
  - Otherwise pass in \texttt{NULL} (0)

```c
#include "kernel/types.h"
#include "user/user.h"

int main(int argc, char *argv[]){
    int pid;
    int child_status;
    printf("parent (pid:%d)\n", getpid());
    pid = fork();
    if (pid > 0) {
        // parent
        pid = wait(&child_status);
        printf("child %d is done with exit code %d\n", pid, child_status);
    }
    else if (pid == 0) {
        // child
        printf("child (pid:%d) exiting\n", getpid());
        exit(-1);
    }
    else {
        printf("fork error\n");
    }
    exit(0);
}
```

\textbf{fork\_ex.c}
The **exec()** System Call

- Run a program that is different from the calling program
  - First argument is the program to run
  - Second argument is the command line for the program
- **exec** never returns (unless error)

```c
#include "kernel/types.h"
#include "user/user.h"

int main(int argc, char *argv[])
{
    char *new_argv[3];
    new_argv[0] = "echo";
    new_argv[1] = "hello";
    new_argv[2] = 0;
    exec("echo", new_argv);
    printf("exec error
");
    exit(0);
}
```
Motivating the API

• Why the odd interface for the simple act of creating a new process?
• Why are `fork()` and `exec()` separate functions?
• Necessary for building a UNIX shell
  • It lets the shell run code after the call to `fork()` but before the call to `exec()`
  • Can alter the environment of the about to be run program
  • Can easily support things like redirection and pipes
File redirection

• Simplified version of the code a shell runs for the command
  • `cat < input.txt`

```c
char *argv[2];
argv[0] = "cat";
argv[1] = 0;
if (fork() == 0) {
    close(0);
    open("input.txt", O_RDONLY);
    exec("cat", argv);
}
```
How to Efficiently Virtualize the CPU with Control?

• The OS needs to share the physical CPU by **time sharing**

• Issues
  • **Performance**: How can we implement virtualization without adding excessive overhead to the system?
  • **Control**: How can we run processes efficiently while retaining control over the CPU?
Direct Execution

Just run the program directly on the CPU

<table>
<thead>
<tr>
<th>OS</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Create entry for process list</td>
<td>7. Run main()</td>
</tr>
<tr>
<td>2. Allocate memory for program</td>
<td>8. Execute return from main()</td>
</tr>
<tr>
<td>3. Load program into memory</td>
<td></td>
</tr>
<tr>
<td>4. Set up stack with argc / argv</td>
<td></td>
</tr>
<tr>
<td>5. Clear registers</td>
<td></td>
</tr>
<tr>
<td>6. Execute call main()</td>
<td></td>
</tr>
<tr>
<td>9. Free memory of process</td>
<td></td>
</tr>
<tr>
<td>10. Remove from process list</td>
<td></td>
</tr>
</tbody>
</table>

Without *limits* on running programs, the OS wouldn’t be in control of anything and thus, would be “just a library”
Problem 1: Restricted Operation

What if a process wishes to perform some kind of restricted operation such as ...

- Issuing an I/O request to a disk
- Gaining access to more system resources such as CPU or memory

Solution: Using protected control transfer

- **User mode**: Applications do not have full access to hardware resources
- **Kernel mode**: The OS has access to the full resources of the machine
System Call

- Allow the kernel to **carefully expose** certain **key pieces of functionality** to user program, such as ...
  - Accessing the file system
  - Creating and destroying processes
  - Communicating with other processes
  - Allocating more memory

- **Trap** instruction
  - Jump into the kernel
  - Raise the privilege level to kernel mode

- **Return-from-trap** instruction
  - Return into the calling user program
  - Reduce the privilege level back to user mode
Limited Direction Execution Protocol @Boot

OS @ boot (kernel mode)

| initialize trap table |

Hardware

| remember address of ... syscall handler |
## Limited Direction Execution Protocol @Run

<table>
<thead>
<tr>
<th>OS @ run (kernel mode)</th>
<th>Hardware</th>
<th>Program (user mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create entry for process list</td>
<td></td>
<td>Run main()</td>
</tr>
<tr>
<td>Allocate memory for program</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Load program into memory</td>
<td></td>
<td>Call system call</td>
</tr>
<tr>
<td>Setup user stack with argv</td>
<td></td>
<td>trap into OS</td>
</tr>
<tr>
<td>Fill kernel stack with reg/PC</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>return-from-trap</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Handle trap

- Do work of syscall

**return-from-trap**

Free memory of process

Remove from process list

- save regs to kernel stack
- move to kernel mode
- jump to trap handler

- restore regs from kernel stack
- move to user mode
- jump to main

- ... from main
- trap (via exit())
Problem 2: Switching Between Processes

• How can the OS regain control of the CPU so that it can switch between processes?
  • A cooperative Approach: Wait for system calls
  • A Non-Cooperative Approach: The OS takes control
A cooperative Approach: Wait for system calls

• Processes **periodically give up the CPU** by making **system calls** such as **yield**
  • The OS decides to run some other task
  • Application also transfer control to the OS when they do something illegal
    • Divide by zero
    • Try to access memory that it shouldn’t be able to access

• Examples: early versions of the Macintosh OS, the old Xerox Alto system

A process gets stuck in an infinite loop
→ **Reboot the machine**
A Non-Cooperative Approach: OS Takes Control

• **A timer interrupt**
  • During the boot sequence, the OS start the timer
  • The timer raise an interrupt every so many milliseconds
  • When the interrupt is raised:
    • The currently running process is halted
    • Save enough of the state of the program
    • A pre-configured interrupt handler in the OS runs

A timer interrupt gives OS the ability to run again on a CPU
Saving and Restoring Context

- **Scheduler** makes a decision:
  - Whether to continue running the **current process**, or switch to a **different one**
  - If the decision is made to switch, the OS executes a **context switch**
Context Switch

• A low-level piece of assembly code
  • **Save a few register values** for the current process onto its kernel stack
    • General purpose registers
    • PC
    • Kernel stack pointer
  • **Restore a few register values** for the soon-to-be-executing process from its kernel stack
  • **Switch to the kernel stack** for the soon-to-be-executing process
### Limited Direction Execution Protocol (Timer interrupt) @Boot

<table>
<thead>
<tr>
<th>OS @ boot (kernel mode)</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize trap table</td>
<td>remember address of ...</td>
</tr>
<tr>
<td></td>
<td>syscall handler</td>
</tr>
<tr>
<td></td>
<td>timer handler</td>
</tr>
<tr>
<td>start interrupt timer</td>
<td>start timer</td>
</tr>
<tr>
<td></td>
<td>interrupt CPU in X ms</td>
</tr>
</tbody>
</table>
Limited Direction Execution Protocol (Timer interrupt) @Run

<table>
<thead>
<tr>
<th>OS @ run (kernel mode)</th>
<th>Hardware</th>
<th>Program (user mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Process A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>timer interrupt</td>
<td></td>
<td>Handle the trap</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call switch() routine</td>
</tr>
<tr>
<td></td>
<td></td>
<td>save regs(A) to k-stack(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>move to kernel mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>jump to trap handler for timer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>return-from-trap (into B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>restore regs(B) from k-stack(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>move to user mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>jump to B’s PC</td>
</tr>
</tbody>
</table>

Process B

...
Worried About Concurrency?

• What happens if, during interrupt or trap handling, another interrupt occurs?

• OS handles these situations:
  • Disable *interrupts* during interrupt processing
  • Use a number of sophisticated *locking* schemes to protect concurrent access to internal data structures
Separating Policy and Mechanism

• Design paradigm
  • Separate high-level policies from their low-level mechanisms

• Mechanism
  • Answers the “how” question about a system
  • How does the OS perform a context switch?

• Policy
  • Answers the “which” question about a system
  • Which process should the OS run right now?

• Allows for policies to change without having to rethink the underlying mechanism
  • Gives the system good modularity