SE350: Operating Systems

Lecture 3: Process Management
Outline

• Safe control transfer
  • How do we switch from one mode to the other?
  • What should hardware provide?

• Native control of process
  • Can processes create other processes?
  • `fork()`, `exec()`, `wait()`, `signal()`
Recall: Three Types of Mode Transfer

- **Syscall**
  - Process requests system service, e.g., `exit`
    - Like function call, but outside process
  - Process does not have address of system function to call
    - Like a Remote Procedure Call (RPC) – for later
  - OS marshalls syscall id and args in registers and exec syscall

- **Interrupt**
  - External asynchronous event triggers context switch, e.g., Timer, I/O device
    - Independent of user process

- **Trap or exception**
  - Internal synchronous event in process triggers context switch, e.g., protection violation (segmentation fault), divide by zero, …
Implementing Safe Mode Transfers

• It should be impossible for buggy or malicious user program to cause kernel to corrupt itself
  • Controlled transfer into kernel (e.g., interrupt vector table)
  • Separate kernel stack

• Carefully constructed kernel code should pack up user process state and set it aside
  • Details depend on the machine architecture
Need for Separate Kernel Stacks

• Kernel cannot put anything on user stack (Why?)
  • Reliability: what if user program’s SP is not valid?
  • Security: what if other threads in user process change kernel’s return address?

• Two-stack model
  • Kernel keeps separate stack for each thread in kernel memory (in addition to user stack in user memory)
Two-Stack Model

User stack

Running
- main
- Proc1
- Proc2
- syscall

Handling syscall
- main
- Proc1
- Proc2
- syscall

Ready to run
- main
- Proc1
- Proc2

Kernel stack

Running
- user CPU state
- syscall handler
- I/O driver

Handling syscall
- user CPU state

Ready to run
- user CPU state
Interrupt Masking

- Interrupt handler runs with interrupts off
  - Re-enabled when interrupt completes
- OS kernel can also turn interrupts off
  - E.g., when determining next process/thread to run
  - On x86
    - `cli`: disable interrupts
    - `sti`: enable interrupts
    - Only applies to current CPU (on a multicore)
- We will need this to implement synchronization (more on this later)
Atomic Transfer of Control

```
foo() {
    while (...) {
        x = x + 1;
        y = y - 2;
    }
}
```

```
handler() {
pushad;
...
}
```
Atomic Transfer of Control (cont.)

- Single instruction to
  - Save some registers (e.g., SP, PC)
  - Change PC and SP
  - Switch Kernel/user mode

```
foo() {
  while (...) {
    x = x + 1;
    y = y - 2;
  }
}
```

```
handler() {
  pushad;
  ...
}
```

Kernel stack

Processor Registers
- PC
- SP
- EFLAGS
- Other Regs
  - EAX, EXB,
  - ...

Kernel

User-level process

User stack
Atomic Transfer of Control (cont.)

- Single instruction to save all registers
- Why is stack pointer saved twice?
  - Hint: are they the same?

```c
foo() {
  while (...) {
    x = x + 1;
    y = y - 2;
  }
}
```

User-level process

```
handler() {
  pushad;
  ...
}
```

Kernel

```
handler() {
  pushad;
  ...
}
```

Kernel stack

User stack
Kernel System Call Handler

• Vector through well-defined syscall entry points!
  • Table mapping system call number to handler

• Locate arguments
  • In registers or on user (!) stack

• Copy arguments (copy before check)
  • From user memory into kernel memory
  • Protect kernel from malicious code evading checks

• Validate arguments
  • Protect kernel from errors in user code

• Copy results back
  • Into user memory
At the End of Handler

- Handler restores saved registers
- **Atomically** return to (interrupted) process/thread
  - Restore PC, SP
  - Restore processor status
  - Switch to user mode
How Does Kernel Provide Services?

• You said that applications request services from the operating system via syscall, but …
  • I’ve been writing all sort of useful applications and I never ever saw a “syscall” !!!

• That’s right

• It was buried in the programming language runtime library (e.g., libc.a)
  • … Layering
OS Run-Time Library

Proc 1  Proc 2  ...  Proc n

OS

Proc 1  Proc 2  ...  Proc n

OS Library  OS Library  OS Library

OS

Portable OS Library

System Call Interface

Portable Operating System Kernel

Web Servers  Compilers  Source Code Control  Web Browsers  Email  Databases  Word Processing

x86  ARM  PowerPC

10Mbps/100Mbps/1Gbps Ethernet  802.11 a/b/g/n  SCSI  IDE  Graphics Accelerators  LCD Screens
Putting it together: web server

Client

Web Server

Request

Reply
(retrieved by web server)
Putting it together: web server

1. network socket read
2. copy arriving packet (DMA)
3. kernel copy
4. parse request
5. file read
6. disk request
7. disk data (DMA)
8. kernel copy
9. format reply
10. network socket write
11. kernel copy from user buffer to network buffer
12. format outgoing packet and DMA

Request

Reply
Summary of Hardware Support for OS

• Privilege modes
• Privileged instructions
• Memory translation
• Processor exceptions
• Timer interrupts
• Device interrupts
• Interprocessor interrupts
• Interrupt masking
• System calls
• Return from interrupt
Can a Process Create a Process?

- **fork()** system call creates copy of current process with new unique process ID (PID)
- Return value from **fork()** is integer
  - When > 0
    - Running in (original) **parent** process
    - return value is PID of new **child**
  - When = 0
    - Running in new child process
  - When < 0
    - Error! Must be handled somehow
    - Running in original process
- All state of original process duplicated in both parent and child!
  - Memory, file descriptors (more on this later), etc…
UNIX Process Management

```c
main() {
    pid = fork();
    if (pid == 0) {
        exec(...);
    } else {
        wait(pid);
    }
}
```
UNIX Process Management (cont.)

• fork()
  • Syscall to create copy of current process and start it

• exec()
  • Syscall to change program being run by current process

• wait()
  • Syscall to wait for process to finish

• signal()
  • Syscall to send notification to another process
    (e.g., SIGKILL, SIGINT)
```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    pid_t cpid, mypid;
    pid_t pid = getpid(); /* get current processes PID */
    printf("Parent pid: %d\n", pid);
    cpid = fork();
    if (cpid > 0) { /* Parent Process */
        mypid = getpid();
        printf("[%d] parent of [%d]\n", mypid, cpid);
    } else if (cpid == 0) { /* Child Process */
        mypid = getpid();
        printf("[%d] child\n", mypid);
    } else {
        perror("Fork failed");
        exit(1);
    }
    exit(0);
}
```
Implementing a Shell

1. char *prog, **args;
2. int child_pid;

3. // Read and parse the input a line at a time
4. while (readAndParseCmdLine(&prog, &args)) {
5.     child_pid = fork(); // create a child process

6.     if (child_pid == 0) {
7.         exec(prog, args); // I'm the child process. Run program
8.     } else {
9.     } else {
10.        wait(child_pid); // I'm the parent, wait for child
11.        return 0;
12.    }
13.  }

Summary

• Safe control transfer
  • How do we switch from one mode to the other?
  • What should hardware provide?

• Native control of process
  • Can processes create other processes?
  • `fork()`, `exec()`, `wait()`, `signal()`
Questions?
Acknowledgment

• Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, and Canny