CSC2/452 Computer Organization
Privileges, System Calls, and Processes

Sreepathi Pai

URCS

November 11, 2019

Outline

Outline

Administrivia

- Homework #5 was due today
- Homework #6 is out today
  - Some printouts with me
- Assignment #4 will be out later today
  - Due date: Tuesday, Nov 26, 7PM
- Assignment #3 grading should be done by then

Outline

Virtual Memory Protections

- Different processes have distinct address spaces
  - Virtual addresses from 0 onwards ...
- MMU translates virtual addresses to physical addresses
  - Physical addresses may be shared
  - Different virtual addresses may map to same physical addresss
  - Translation determined by structures called page tables
- Translation performed at granularity of a page (usually 4096 bytes)

Setup and Usage

- Page tables are setup by the OS kernel
  - They live in memory
- Your process can change page table entries
  - by calling `mmap`
  - by calling `mprotect`
- These functions are implemented ultimately by the kernel
  - Why?

The Big Question

Why is the OS kernel necessary?

The Big Question: Rephrased

Why aren't processes allowed to directly modify page table entries?

The Real Question

How do we prevent processes from performing potentially unsafe actions (like changing page table entries) directly?

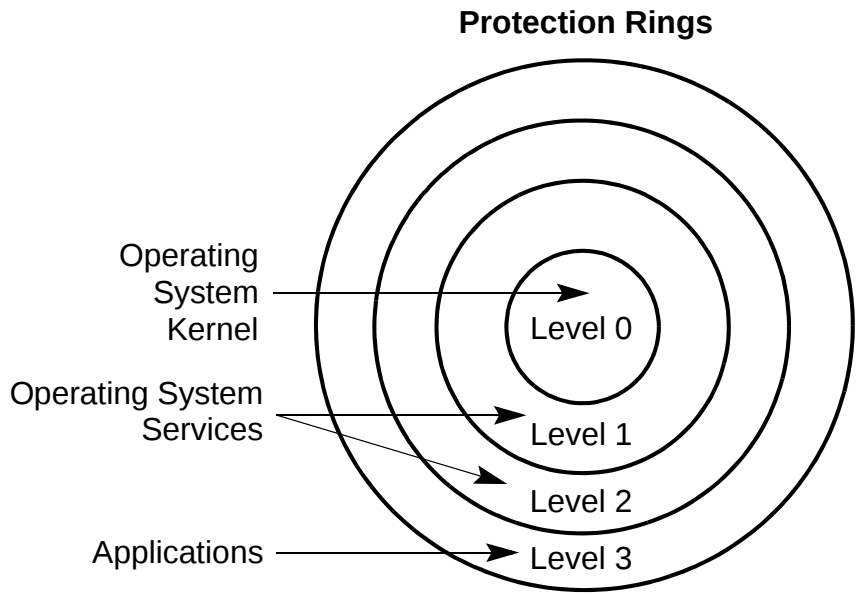
Outline

Privilege Level

- Privilege levels are a mechanism to restrict activities performed by processes
- Many processors support multiple privilege levels
  - x86 supports at least 4 visible to programmers
- All code runs at some privilege level
- Different privilege levels support different abilities
- For example, instructions can be distinguished into "unprivileged" and "privileged"

x86 Privilege Model

- x86 has 4 privilege levels organized as “rings”
  - Innermost ring (Ring 0) is most privileged, runs OS kernel
  - Outermost ring (Ring 3) is least privileged, runs applications
- Most operating systems do not use the other rings



RISC-V Privilege Model

- The RISC-V processor is the "fifth" RISC design out of Berkeley
  - Completely open design (`https://www.risc-v.org/`)
- Supports three privilege levels
  - User/Application (U)
  - Supervisor (S)
  - Machine (M)
- Privilege ordering
  - M has highest privilege, and can do anything
  - S has medium privilege, and can do most OS functions
  - U has least privilege, and runs user applications

General Operation of Privilege Levels

- OS sets up privilege levels on boot
- OS transitions to a lower privilege level and begins executing user-space programs
  - On Unix systems, this is the `init` process (PID 1)
  - The `init` process ultimately runs the shell (i.e. the command-line on Linux)
- Processes invoke *system calls* to get the OS to perform actions on their behalf
  - A system call is like a function call, except it transitions between privilege levels
- Attempts by processes to directly perform actions that require higher privileges are detected by CPU
  - Usually delivers a general protection fault on x86

What requires privileges (on x86)?

- Executing "privileged" instructions
  - INVLPG
  - MOV (to and from control registers, CR0–8)
  - MOV (to and from debug registers, DR0–7)
  - RDTSC (if it is restricted by OS kernel, otherwise it is unprivileged)
  - IN/OUT instructions
  - 9 more (see Section 5.9, Privileged Instructions in the Intel System Developers Manual)
- Reading/Writing pages that have the U/S (user/supervisor bit) set to 0
  - These are called "supervisor-mode" pages/addresses
  - Page tables have these set to 0
- Memory-mapped I/O
  - Usually though page table permissions

CPU privileges != OS privileges

- CPU privileges are not the same as OS-level privileges/users
  - `root`'s programs also run in user mode
- CPU has no idea of OS-level users

Beyond Privileges

- Capabilities
  - Hardware capabilities allow programs that possess them to perform actions associated with those capabilities
- Hardware Protection Domains
  - Carve out isolation domains for programs
  - A program inside the isolation domain is private, even from the OS kernel
  - Examples: Intel SGX, ARM TrustZone, etc.

Outline

System Call

- Like a function call, except it changes privilege levels
  - From user mode to supervisor mode
- Older x86 mechanisms
  - JMP, CALL, INT
  - Look up descriptor tables for the destination address
  - Table entry could describe a "gate" that would allow change of privilege levels
  - ... (lots of details elided)
  - Lots of book keeping involved – this was a very general mechanism
- Newer x86-64 mechanism
  - SYSCALL and SYSRET instructions

SYSCALL

- SYSCALL changes to privilege level 0 and runs code at an address specified in the `IA32_LSTAR` machine-specific register
  - This code is inside the OS kernel
- Saves return address (instruction after SYSCALL) in `%rcx`
- Saves `%rflags` in `%r11`, and masks out certain flags
- Changes code segment and stack segment
  - Segments are an x86 oddity, they allow partitioning memory into segment + offset pairs
  - Most OS uses flat addressing, ignoring segments
  - In flat addressing, the OS may use the same segment
- OS kernel code is now running at privilege level 0

SYSRET

- Returns from system call, switching back to privileged mode
- Switches back to privilege level 3
- Restores `%rflags`, and returns to address stored in `%rcx`
- User program now resumes in privilege level 3

Programmer's View of System Calls

- Invoking system calls is very system specific
  - x86-64, SYSCALL
  - x86, INT 0x80 (for Linux)
- Method #1: Just use the libc wrapper
  - Most portable
  - The C Standard Library wraps many system calls as functions
  - This is normal way of using them
- Example: `mmap()` is a libc wrapper around the `mmap` system call

Advanced methods: Use the GNU `syscall` wrapper

- Method #2: Use GNU libc's `syscall` function
  - Not to be confused with the SYSCALL instruction
  - Useful when no wrapper exists (yet)
  - Less portable (may not be supported by non-GNU libc)
- Example: `syscall(SYS_mmap, ...)` will call the `mmap` system call

Advanced methods: Use assembly language

- Method #3: Use inline (or external) assembly
  - Write assembly code
  - Maybe you don't have access to libc?
  - Least portable, most direct
- Example:

```
movq $9, %rax     # 9 is syscall number for mmap on x86-64
 ...              # pass arguments to mmap, using standard ABI
syscall
```

Summary

- Privilege levels prevent programs from doing anything they want
  - Execute privileged instructions
  - Read/write arbitrary memory addresses
- System calls allow the OS kernel to perform actions on behalf of programs
  - Like a function call, but more expensive
  - Involves changing privilege levels

Outline

Process

- A process is a running program
  - has a virtual address space
  - has a stack, heap, code, etc.
- Constrast to a *thread*, which is sometimes called a "light-weight process"
  - A process can consist of multiple threads of execution
  - All threads share the same virtual address space
  - Each thread has its own stack, but shares the heap and code with other threads
  - We will revisit threads later in this course

Creating Processes in Unix

- Two primary methods:
  - The `fork` system call
  - The `exec*` family of system calls

The `fork` system call

- The `fork` system call duplicates a process
- The duplicate is called a child process, whereas the original is called a parent process
- Execution in both processes continues after the `fork` call
  - Usually noted as "`fork` returns twice", once in parent and once in child
- Return value in child is 0
- Return value in parent is child's *process ID*

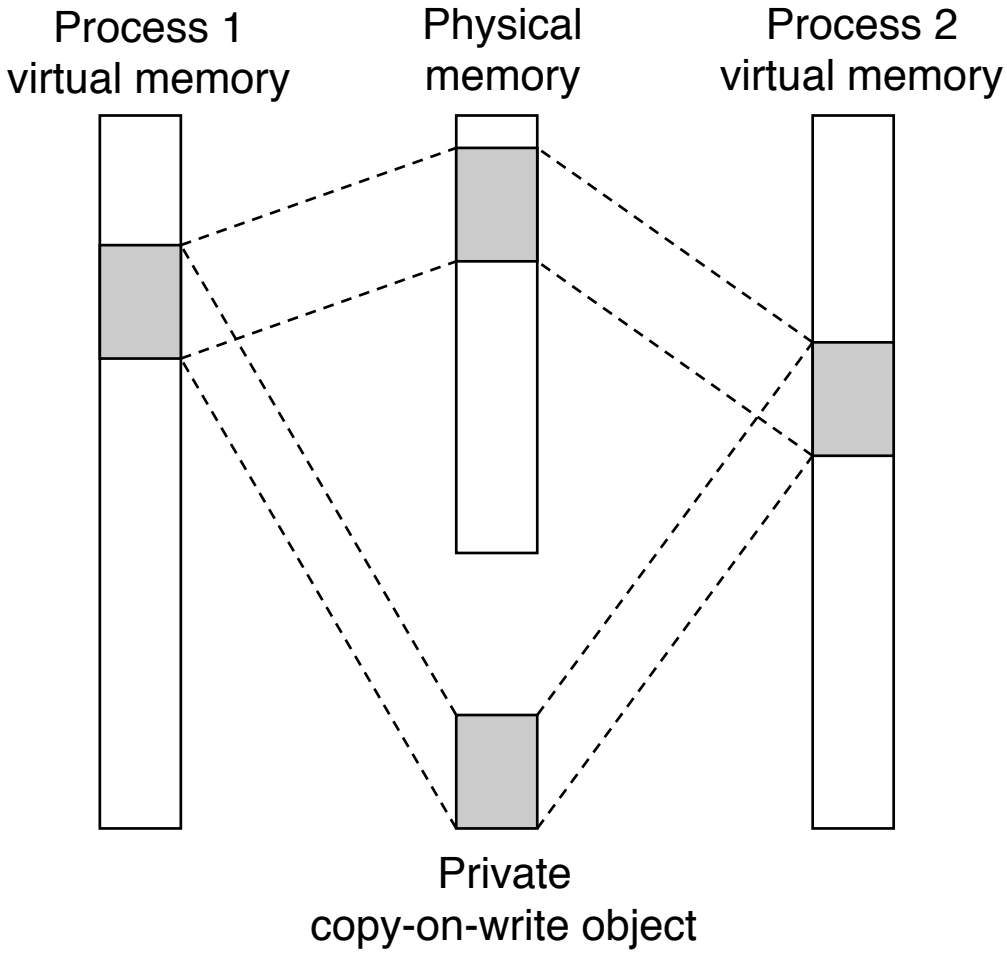
After the fork

- The child inherits an exact copy of the parent's address space
- However, all changes it makes from that point onwards are not reflected back to the parent
  - Except when pages are shared
- Similarly, changes made by parent after the fork are not reflected in the child

Forking is cheap

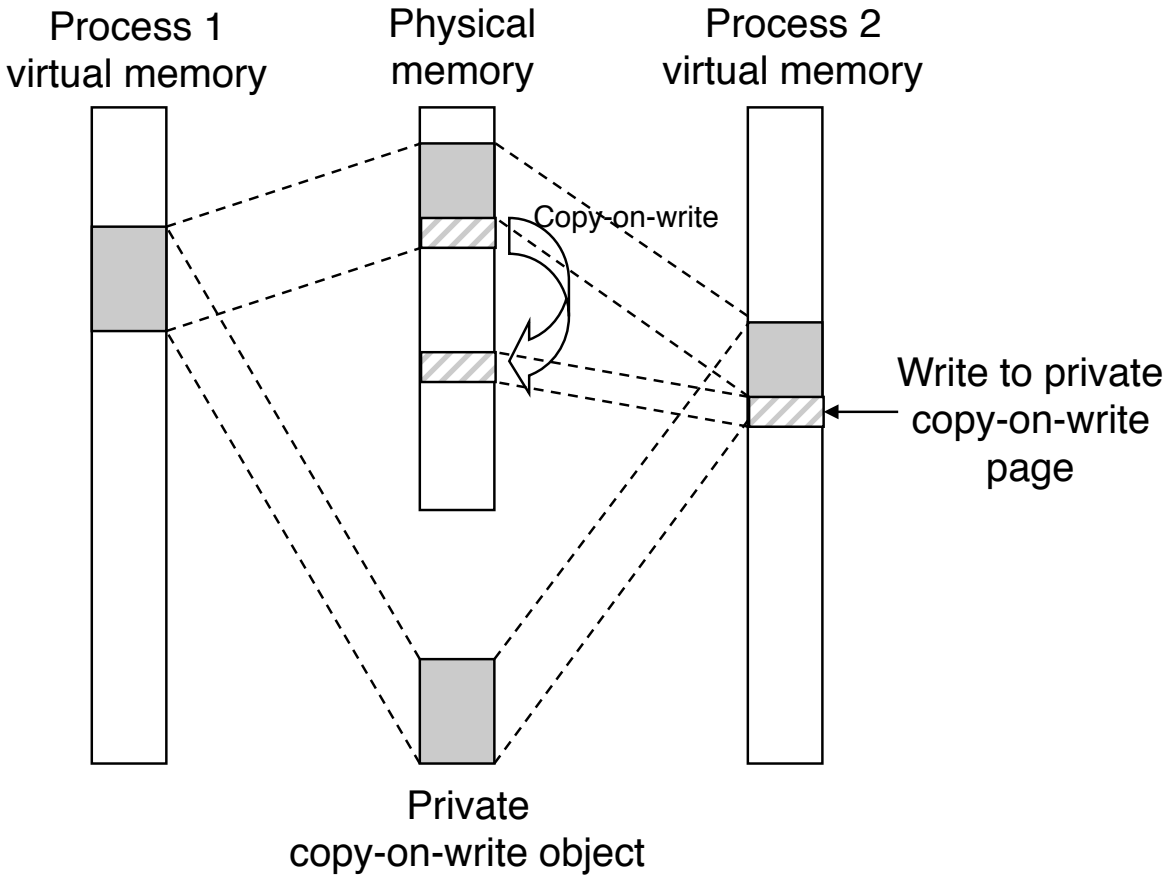
- On Unix systems, the primary method of concurrency is to fork processes
- Need to handle 100 web requests?
  - Fork 10 processes to handle 10 requests each
- Other operating systems use *threads* for concurrency
  - Process creation can be very expensive on those OSes
- How is process creation so cheap on Unix-like systems?

Virtual Memory to the Rescue: Copy-on-write #1



- After forking, the physical pages of the two processes are placed into "copy-on-write" mode
- But no copies are made – only permissions changed to read-only!

Virtual Memory to the Rescue: Copy-on-write #2



- When a write actually happens, only the page written to is copied/duplicated
- There are now two physical pages, one for each process, and they will have different content

Virtual memory and Copy-on-write

- On Linux, `fork` is implemented using *copy-on-write* mechanisms
- When a process calls `fork`, Linux only has to:
  - set all page table entries to read only
  - duplicate all page tables
  - create any OS-level data structures for the child (e.g. process ID)
  - Note: no copies of physical pages made!
- When either the parent or child writes to a page
  - Linux makes a physical copy of the page, and changes the page to read-write if possible.
  - Write can continue on new page

Other uses of fork

- `fork` can be used to make copies of a process
- But, how to run other programs?
  - I.e. load a different program from disk as a new process

The exec* family of functions

execve(filename, argv, envp);

- Typically, `execve` is used a representative of this family
- The function takes 3 arguments
  - `filename` contains the filename of the program
  - `argv` is an array of strings containing arguments to the program
  - `envp` is an array of strings containing environment variables for the program
- Both `argv` and `envp` contain NULL as the last element
- `argv[0]`, contains by convention, the name of the program
- If `execve` is successful, it does not return
  - Current process is replaced by newly loaded program

Running another program

- To run another program:
  - `fork` and create a child process
  - The child process then executes `execve` to replace itself with the new program
- Or:
  - Just call `execve`

execve example

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  char *newargv[] = { NULL, "/", NULL };
  char *newenviron[] = { NULL };

  newargv[0] = "/bin/ls";

  execve(newargv[0], newargv, newenviron);
  perror("execve");   /* execve() returns only on error */
  exit(EXIT_FAILURE);
}
```

- This code calls `/bin/ls /`
- Example adapted from the execve manual page

execve example with fork, in child

```
if((pid = fork()) == 0) {
  /* in child */
  execve(newargv[0], newargv, newenviron);
  perror("execve");   /* execve() returns only on error */
  exit(EXIT_FAILURE);
}
```

- Execute `fork`, and test if we're in parent or child
- If we're in child, call `execve` to run the new program

execve example with fork, in parent

```
else {
  /* in parent */
  int wstatus;

  waitpid(pid, &wstatus, 0);

  if(WIFEXITED(wstatus))
    printf("normal termination. return value: %d\n",
           WEXITSTATUS(wstatus));
}
```

- Meanwhile in parent, we've received a process ID for the child
- Wait for the child to terminate, using the `wait` system call
  - Detect if the child exited normally, and retrieve its return value

`fork` is asynchronous

- A `fork` executes asynchronously
- The child process is started by a fork, but:
  - may not start running until much later, OR
  - may have already finished by the time control returns to parent, OR
  - may actually start after parent finishes, etc.
- I.e., processes run concurrently, and no ordering can be assumed
  - Unless you explicitly synchronize with a child process
  - Take CSC2/458 or CSC2/456

Defunct ("Zombie") Processes

- The kernel assumes *somebody* is interested in a process's information after it terminates
  - e.g. return value
- Therefore, a process that exits or is killed is not fully cleaned up
  - Memory, files, etc. are freed and/or closed
  - But PID is not freed
- The process is in a *defunct* state, usually referred to as a *zombie* state, waiting for a *reaper*
  - If the parent process has not called `waitpid`, the zombie will hang around until it does
  - If the parent process terminates without calling `waitpid`, the child is "reparented" to `init`, which will reap it
- What happens if `init` dies?

Summary

- Processes can be created by `fork` and `execve`
- `fork` is especially cheap on Unix-like systems
  - Thanks to copy-on-write

References and Acknowledgements

- Volume 3 of the Intel Architectures Software Developers Manual contains details on protection
  - Figure of rings from this manual
- The RISC-V Instruction Set Manual: Volume II: Privileged Architecture
- Section 2 of the Linux Programmers Manual covers system calls
- Portions of Chapter 8 and 9 of the textbook cover creating and managing processes, and also the role of VM
  - Figure of copy-on-write from the textbook