WASHINGTON STATE
UNIVERSITY

CPTS 360 Systems Programming C/C++

Spring 2024

Lecture 22
Exam Review

Monowar Hasan
School of Electrical Engineering & Computer Science
Washington State University

Announcements

- Last lecture today!
  - Additional office hours before the exam:
    - Wed (April 24) 2:30-3:30 PM
    - Thu (April 25) noon-1 PM
    - Thu (April 25) 2:00-3:30 PM
    - Fri (April 26) 1:00-2:00 PM

Announcements

- Final Exam
  - **04/30/24 10:30 AM - 12:30 PM @ Classroom**
- Format
  - MCQ, True/False, Essay
- Syllabus
  - **ALL lectures AND Labs**
    - 80% questions from new topics (after mid)
- One-page handwritten sheet allowed (both sides okay)
  - **Must write by hand on the paper**
  - **Print from digital handwriting is NOT allowed**
- **Bring your ID!**

Focus on conceptual understanding

Scheduling

The Linux Completely Fair Scheduling (CFS)

- The current CPU scheduler in Linux
  - Non-fixed timeslice
  - CFS assigns process's timeslice a proportion of the processor
  - Priority
    - Enables control over priority by using "nice" value
- CFS control parameters
  - Virtual runtime (`vruntime`)
    - Denote how long the process has been executing
  - Per-process variable
    - Increase in proportion with physical (real) time when it runs

- CFS will pick the process with the lowest `vruntime` to run next

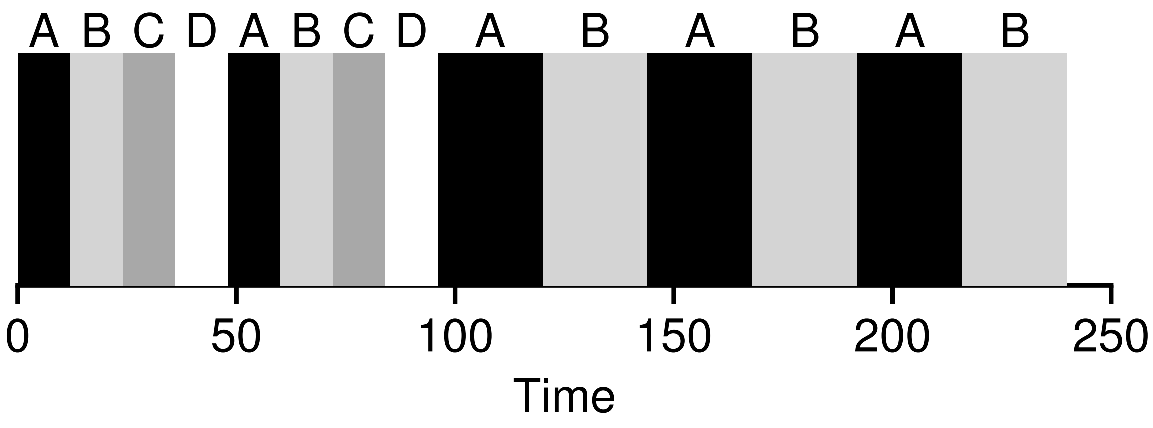
CFS Basics

- **How does the scheduler know when to stop the currently running process and run the next one?**
  - CFS switches too often → fairness is increased
    - CFS will ensure that each process receives its share of CPU, but at the cost of performance (too much context switching)
  - CFS switches less often → performance is increased
    - Reduced context switching, but at the cost of near-term fairness

- CFS manages this tension through various control parameters

CFS Control Parameters

- To determine how long one process should run before considering a switch → `sched_latency`
  - A typical value is 48 (milliseconds)
  - Process's timeslice = `sched_latency` / (the number of processes)
- Example:
  - N=4 processes (A, B, C, D) and then 2 processes (C, D) complete
  - `sched_latency`/N = 12 ms



- Four jobs (A, B, C, D) each run for two timeslices with 12 ms timeslice
- Two of them (C, D) then complete → leaving (A, B) remaining
- (A, B) then each run for 24 ms in round-robin fashion

CFS Control Parameters

- **What if there are "too many" processes running?**
  - Does that lead to too small of a timeslice (too many context switches)?

- `min_granularity`
  - The minimum timeslice (6 ms)
  - Ensure that not too much time is spent in scheduling overhead, when there are too many processes running
  - Example:
    - Ten processes running
      - Without `min_granularity`: time slice (`sched_latency`/10: 4.8 ms)
      - With `min_granularity`:. time slice 6 ms

Kernel Programming

Writing First Kernel Module

```c
#include <linux/init.h> /* Needed for the macros */
#include <linux/module.h> /* Needed by all modules */
#include <linux/printk.h> /* Needed for pr_info() */

static int hello3_data __initdata = 3;

static int __init hello_3_init(void)
{
    pr_info("Hello, world %d\n", hello3_data);
    return 0;
}

static void __exit hello_3_exit(void)
{
    pr_info("Goodbye, world 3\n");
}

module_init(hello_3_init);
module_exit(hello_3_exit);
```

hello-3.c

Read and Write a `/proc` File

- Example:
  - Proc file: `/proc/procbuf`
  - Read and write to a kernel buffer from user space

```
$> echo "Cougs" > /proc/procbuf
Cougs
$> cat /proc/procbuf
Cougs
```

Read and Write a `/proc` File

- To move data between kernel/user space:
  - `copy_to_user()`
    - Copies a block of data from the kernel into user space
    - Accepts a pointer to a user space buffer, a pointer to a kernel buffer, and a length defined in bytes
    - Returns zero on success or non-zero to indicate the number of bytes that weren't transferred
  - `copy_from_user()`
    - Copies a block of data from user space into a kernel buffer
    - Accepts a destination buffer (in kernel space), a source buffer (from user space), and a length defined in bytes
    - As with `copy_to_user`, the function returns zero on success and non-zero to indicate a failure to copy some number of bytes

Read and Write a `/proc` File

```c
static const struct proc_ops proc_file_fops = {
    .proc_read = procfile_read,
    .proc_write = procfile_write,
};
```

Read and Write a /proc File

```c
/* This function is called then the /proc file is read */
static ssize_t procfile_read(struct file *file_pointer, char __user *buffer,
                             size_t buffer_length, loff_t *offset)
{
    int len = sizeof(procfs_buffer);
    ssize_t ret = len;

    if (*offset >= len) {
        return 0;
    }
    if (copy_to_user(buffer, procfs_buffer, len)) {
        pr_info("copy_to_user failed\n");
        ret = 0;
    } else {
        pr_info("procfile read /proc/%s\n", PROCFS_NAME);
        *offset += len;
    }

    return ret;
}
```

procfs.c

Read and Write a /proc File

```c
/* This function is called with the /proc file is written. */
static ssize_t procfile_write(struct file *file, const char __user *buff,
                              size_t len, loff_t *off)
{
    /* Clear internal buffer */
    memset(&procfs_buffer[0], 0, sizeof(procfs_buffer));

    procfs_buffer_size = len;
    if (procfs_buffer_size > PROCFS_MAX_SIZE)
        procfs_buffer_size = PROCFS_MAX_SIZE;

    if (copy_from_user(procfs_buffer, buff, procfs_buffer_size))
        return -EFAULT;

    procfs_buffer[procfs_buffer_size & (PROCFS_MAX_SIZE - 1)] = '\0';
    *off += procfs_buffer_size;
    pr_info("procfile write %s\n", procfs_buffer);

    return procfs_buffer_size;
}
```

procfs.c

Kernel Timer

- Operate in units called "jiffies", not seconds
  - `msec_to_jiffies()` to convert ms to jiffies
  - `jiffies_to_msec()` to convert jiffies to ms

Kernel Timer

```
$> dmesg | tail -2
[138520.717823] Initializing a module with timer.
[138525.936054] This line is printed after 5000 ms.
```

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/jiffies.h>
#include <linux/timer.h>

MODULE_LICENSE("GPL");

static struct timer_list my_timer;
int delay = 5000;

void my_timer_callback(struct timer_list *timer) {
  pr_info("This line is printed after %d ms.\n", delay);
}

static int init_module_with_timer(void) {
  pr_info("Initializing a module with timer.\n");

  /* Setup the timer for initial use. */
  timer_setup(&my_timer, my_timer_callback, 0);
  mod_timer(&my_timer, jiffies + msecs_to_jiffies(delay));

  return 0;
}

static void exit_module_with_timer(void) {
  pr_info("Goodbye, cruel world!\n");
  del_timer(&my_timer);
}

module_init(init_module_with_timer);
module_exit(exit_module_with_timer);
```

timer.c

Periodic Timer

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/jiffies.h>
#include <linux/timer.h>

MODULE_LICENSE("GPL");

static struct timer_list my_timer;
int time_interval = 1000;

void my_timer_callback(struct timer_list *timer) {
  pr_info("This line is printed every %d ms.\n", time_interval);

  /* this will make a periodic timer */
  mod_timer(&my_timer, jiffies + msecs_to_jiffies(time_interval));
}

static int init_module_with_timer(void) {
  pr_info("Initializing a module with a periodic timer.\n");

  /* Setup the timer for initial use. */
  timer_setup(&my_timer, my_timer_callback, 0);
  mod_timer(&my_timer, jiffies + msecs_to_jiffies(time_interval));

  return 0;
}

static void exit_module_with_timer(void) {
  pr_info("Goodbye, cruel world!\n");
  del_timer(&my_timer);
}

module_init(init_module_with_timer);
module_exit(exit_module_with_timer);
```

timer-periodic.c

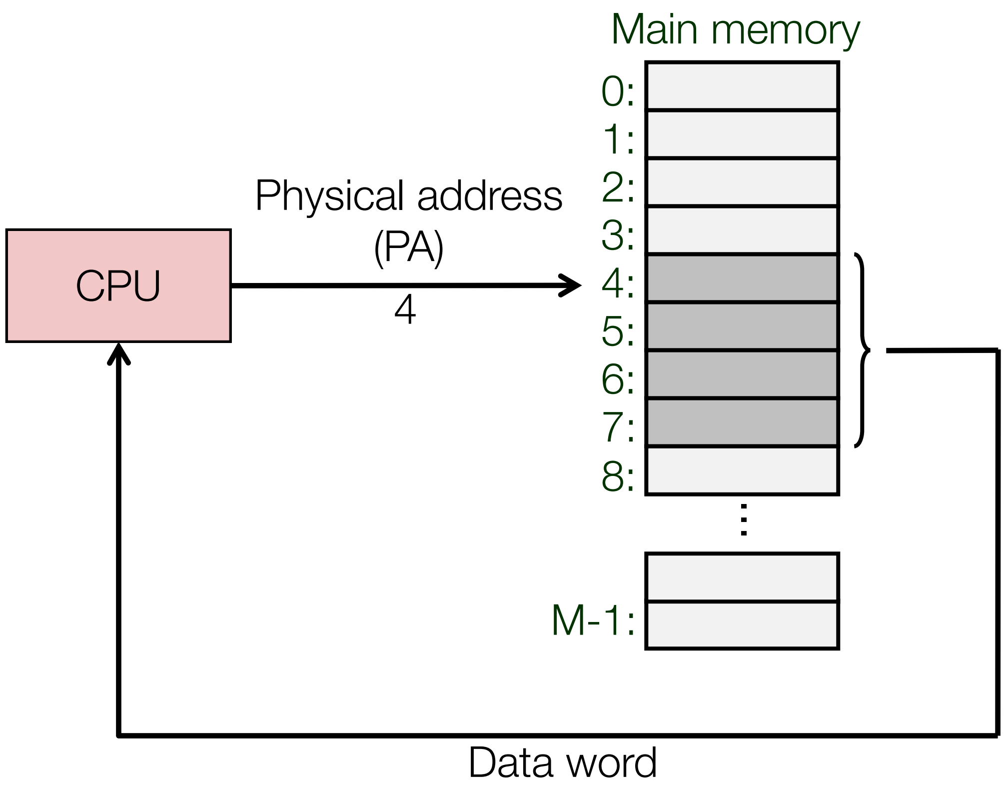
```
$> dmesg | tail -4
[139059.344067] Initializing a module with a periodic timer.
[139060.364666] This line is printed every 1000 ms.
[139061.389089] This line is printed every 1000 ms.
[139062.412243] This line is printed every 1000 ms.
```

Virtual Memory

A System with Physical Memory Only

- Examples:
  - Early PCs
  - Nearly all embedded systems

- CPU's load or store addresses used directly to access memory

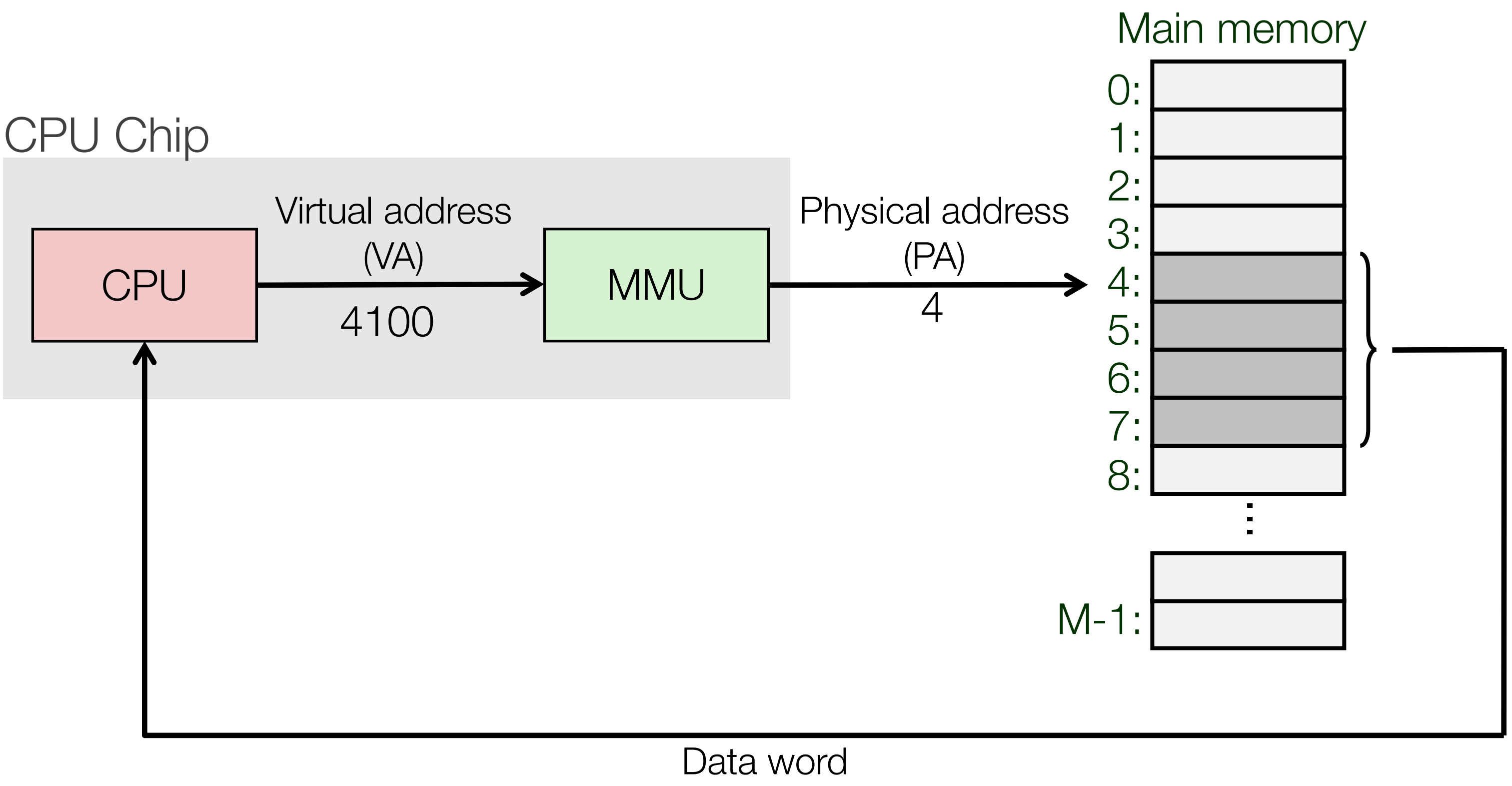


Challenges

- Physical memory is of limited size (cost)
  - What if you need more?
  - Should the programmer be concerned about the size of code/data blocks fitting physical memory?
  - Should the programmer manage data movement from disk to physical memory?
  - Should the programmer ensure two processes do not use the same physical memory?

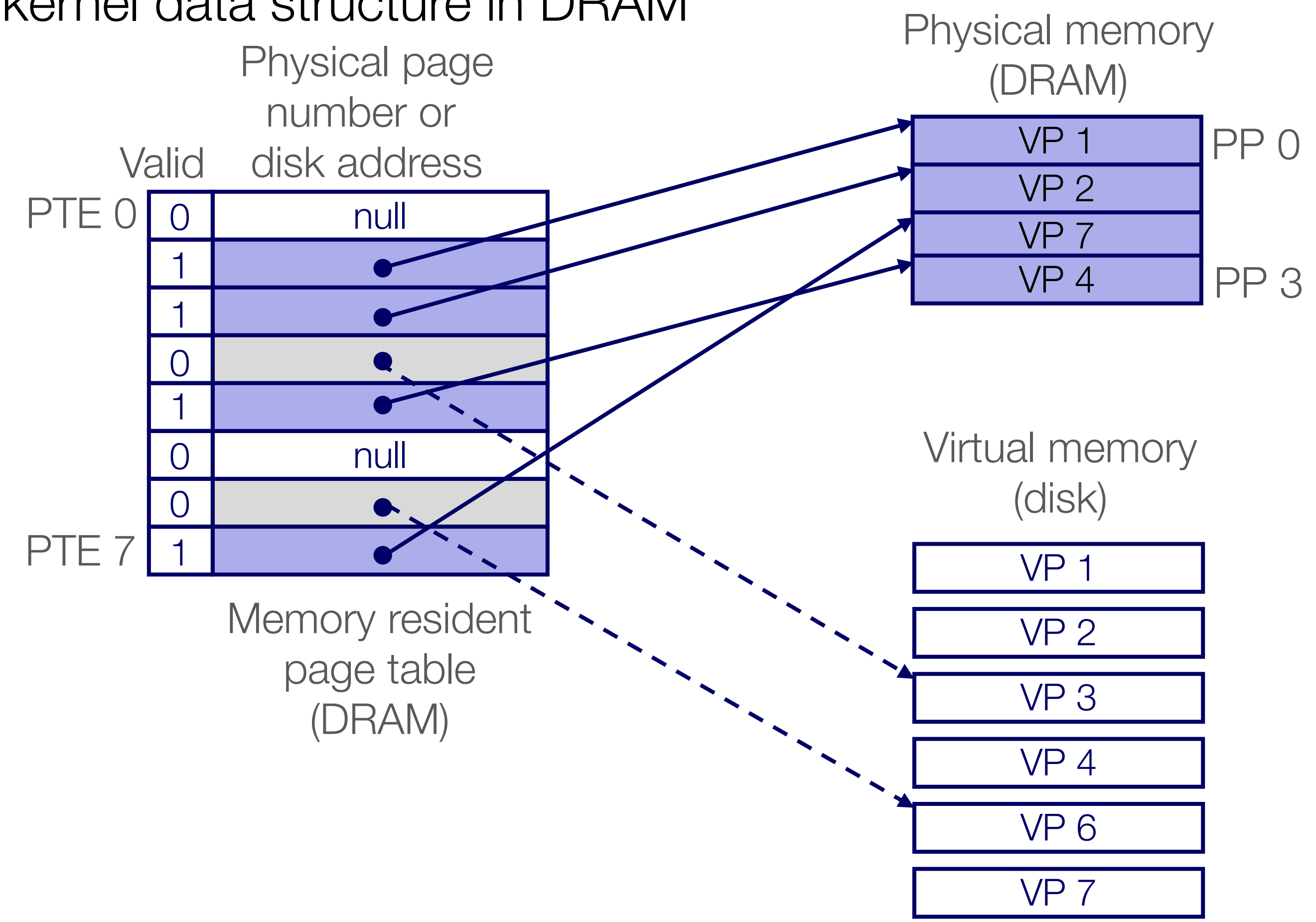
A System with Virtual Memory

- Used in all modern servers, laptops, and smart phones



Enabling Data Structure: Page Table

- A **page table** is an array of page table entries (PTEs) that maps virtual pages to physical pages
  - Per-process kernel data structure in DRAM



VM & Locality

- Virtual memory works well because of locality
- At any point in time, programs tend to access a set of active virtual pages called the **working set**
  - Programs with better temporal locality will have smaller working sets

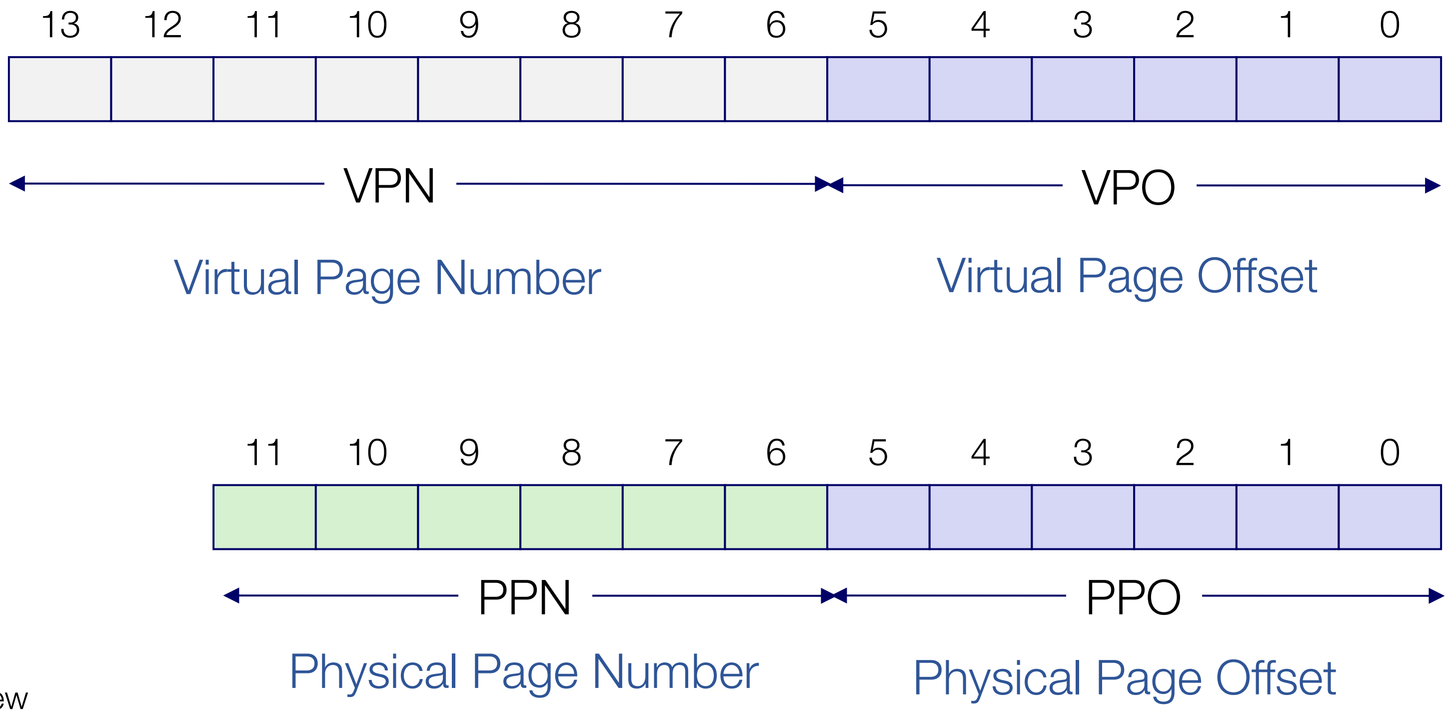
- **If (working set size < main memory size)**
  - Good performance for one process after compulsory misses
- **If (SUM(working set sizes) > main memory size)**
  - Thrashing: Performance meltdown where pages are swapped (copied) in and out continuously

Summary of Address Translation Symbols

- Basic Parameters
  - $N = 2^n$: Number of addresses in virtual address space
  - $M = 2^m$: Number of addresses in physical address space
  - $P = 2^p$: Page size (bytes)
- Components of the virtual address (VA)
  - TLBI: TLB index
  - TLBT: TLB tag
  - VPO: Virtual page offset
  - VPN: Virtual page number
- Components of the physical address (PA)
  - PPO: Physical page offset (same as VPO)
  - PPN: Physical page number

Simple Memory System Example

- Addressing
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes



1. Simple Memory System: TLB

- 16 entries
- 4-way associative

TLBT						TLBI							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
VPN								VPO					

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

2. Simple Memory System: Page Table

Only show first 16 entries (out of 256)

VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

3. Simple Memory System: Cache

- 16 lines, 4-byte block size
- Physically addressed
- Direct mapped

CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
PPN						PPO					

Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Address Translation Example

Virtual Address: `0x03D4`

TLBT						TLBI							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0
VPN								VPO					

VPN 0x0F TLBI 0x3 TLBT 0x03 TLB Hit? Y Page Fault? N PPN: 0x0D

Physical Address

CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	0	0
PPN						PPO					

CO 0 CI 0x5 CT 0x0D Hit? Y Byte: 0x36

Memory-Related Perils and Pitfalls

1. Dereferencing bad pointers
2. Reading uninitialized memory
3. Overwriting memory
4. Referencing nonexistent variables
5. Freeing blocks multiple times
6. Referencing freed blocks
7. Failing to free blocks

Process

fork Example

```c
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}                                    fork.c
```

```
$> ./fork
parent: x=0
child : x=2
```

- Call once, return twice
- Concurrent execution
  - Can't predict execution order of parent and child
- Duplicate but separate address space
  - `x` has a value of 1 when `fork` returns in parent and child
  - Subsequent changes to `x` are independent
- Shared open files
  - `stdout` is the same in both parent and child

`fork` Example

```c
int main()
{
    int a = 9;

    if (Fork() == 0)
        printf("p1: a=%d\n", a--);

    printf("p2: a=%d\n", a++);
    exit(0);
}                                forkprob0.c
```

- What is the output of the child process?

  `p1: a=9`

  `p2: a=8`

- What is the output of the parent process?

  `p2: a=9`

Signals

- A **signal** is a small message that notifies a process that an event of some type has occurred in the system
  - Akin to exceptions and interrupts
  - Sent from the kernel (sometimes at the request of another process) to a process
  - Signal type is identified by small integer ID's (1-30)
  - Only information in a signal is its ID and the fact that it arrived

ID	Name	Default Action	Corresponding Event
2	`SIGINT`	Terminate	User typed `ctrl-c`
9	`SIGKILL`	Terminate	Kill program (cannot override or ignore)
11	`SIGSEGV`	Terminate	Segmentation violation
14	`SIGALRM`	Terminate	Timer signal
17	`SIGCHLD`	Ignore	Child stopped or terminated
10	`SIGUSR1`	Terminate	User-defined signal 1
12	`SIGUSR2`	Terminate	User-defined signal 2

Signal Handling Example

```c
void sigint_handler(int sig) /* SIGINT handler */
{
    printf("So you think you can stop the bomb with ctrl-c, do you?\n");
    sleep(2);
    printf("Well...");
    fflush(stdout);
    sleep(1);
    printf("OK. :-)\n");
    exit(0);
}

int main()
{
    /* Install the SIGINT handler */
    if (signal(SIGINT, sigint_handler) == SIG_ERR)
        unix_error("signal error");

    /* Wait for the receipt of a signal */
    pause();

    return 0;
}
```

sigint.c

Signal Problem

- What is the output of the following program?

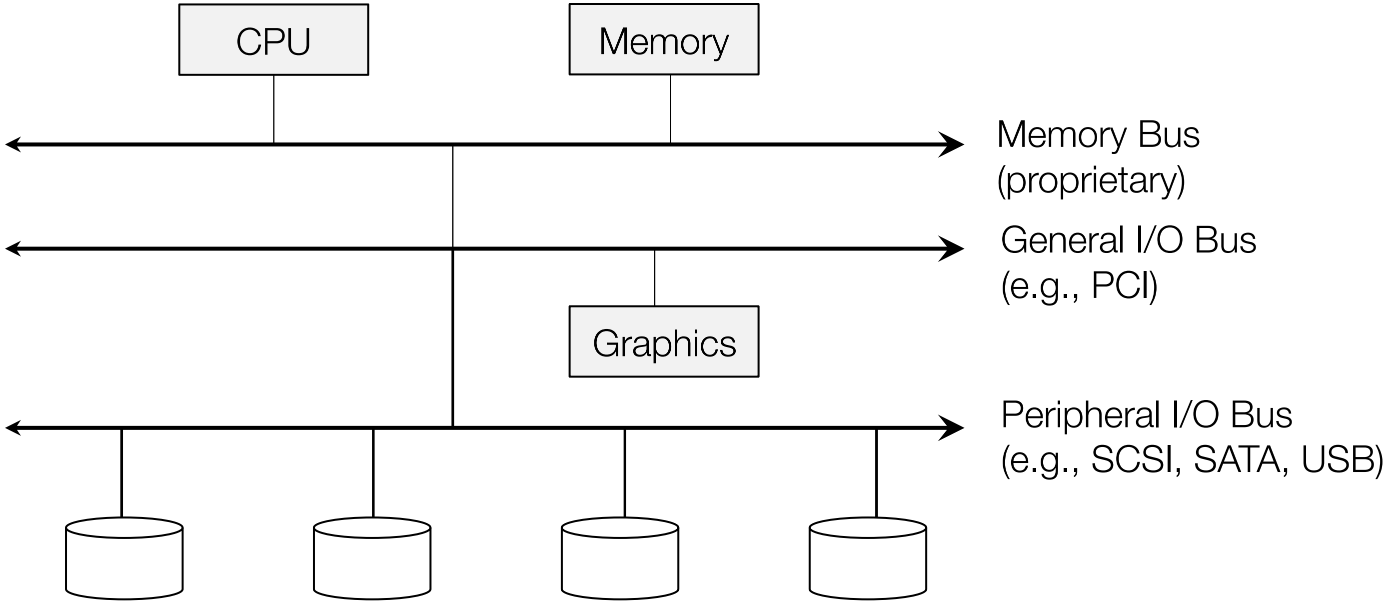
```
counter = 1
```

```c
#include "csapp.h"
int counter = 0;
void handler(int sig)
{
  counter++;
  sleep(1); /* Do some work in the handler */
  return;
}

int main()
{
  int i;

  Signal(SIGUSR2, handler);

  if (Fork() == 0) {  /* Child */
        for (i = 0; i < 5; i++) {
                Kill(getppid(), SIGUSR2);
        }
        exit(0);
  }

  Wait(NULL);
  printf("counter=%d\n", counter);
  exit(0);
}
```

I/O

Input/Output (I/O)

- I/O allows a computer system to interact with other systems



Prototypical System Architecture

- CPU is attached to the main memory of the system via some kind of **memory bus**
- Some devices are connected to the system via a general **I/O bus**

I/O Problem 1

- What is the output of the following program?

```
#include "csapp.h"
int main() {
  int fd1, fd2;
  fd1 = Open("foo.txt", O_RDONLY, 0);
  Close(fd1);
  fd2 = Open("bar.txt", O_RDONLY, 0);
  printf("fd2 = %d\n", fd2);
  exit(0);
}
```

- Unix processes begin life with open descriptors assigned to:
  - `stdin` (descriptor 0), `stdout` (descriptor 1), `stderr` (descriptor 2)
- The `open` function always returns the lowest unopened descriptor
- The output of the program is `fd2 = 3`

I/O Problem 2

- What is the output of the following program?

```
#include "csapp.h"
int main() {
  int fd1, fd2;
  fd1 = Open("foo.txt", O_RDONLY, 0);
  fd2 = Open("bar. txt", O_RDONLY, 0);
  Close(fd2) ;
  fd2 = Open("baz.txt", O_RDONLY, 0);
  printf ("fd2 = %d\n", fd2);
  exit(0);
}
```

- The output of the program is `fd2 = 4`

Concurrency

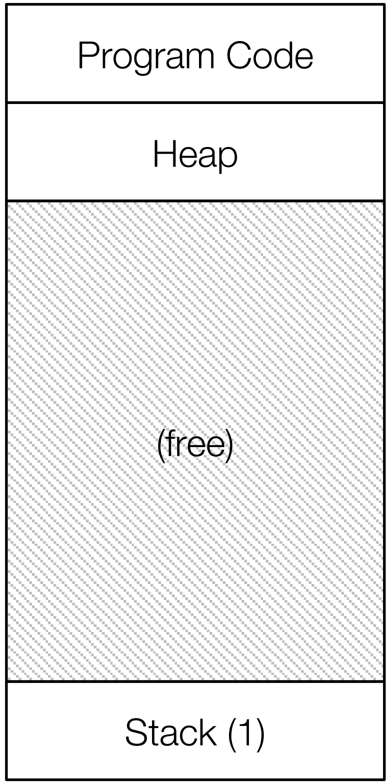
Thread

- A new abstraction for a single-running process

- Multi-threaded program
  - A multi-threaded program has more than one point of execution
  - Multiple PCs (Program Counter)
  - They share the same address space

The Stack

- There will be one stack per thread

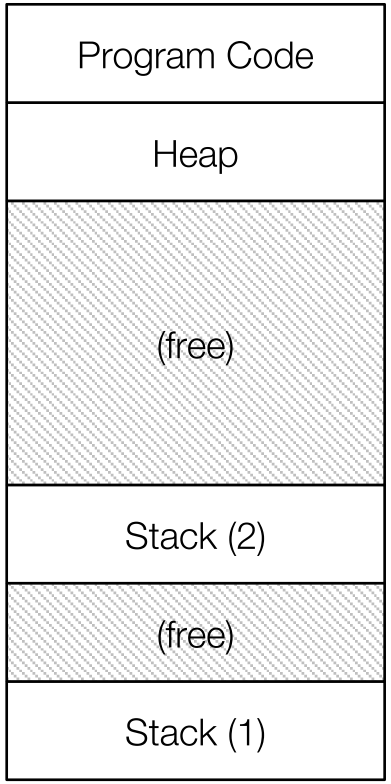


The code segment: where instructions live

The heap segment: contains `malloc`'d data dynamic data structures (it grows downward)

(it grows upward)
The stack segment: contains local variables arguments to routines, return values, etc.

A Single-Threaded Address Space



Two threaded Address Space

Problems with Shared Data: Race Condition

- Increasing the value of a variable
  - `counter = counter + 1`
- Assume:
  - The variable `counter` is in address `0x8049a1c`
  - Variable-length instructions (x86);
    - `mov` instruction takes 5 bytes of memory
    - `add` instruction takes 3 bytes of memory

```
105   mov 0x8049a1c, %eax
108   add $0x1, %eax
113   mov %eax, 0x8049a1c
```

Race Condition

- Example with two threads
  - `counter = counter + 1` (default is 50)
- We expect the result to be **52**. However:

OS	Thread1	Thread2	PC	`%eax`	`counter`
			(after instruction)		
	before critical section		100	0	50
	`mov 0x8049a1c, %eax`		105	**50**	50
	`add $0x1, %eax`		108	**51**	50
`interrupt`					
`save T1's state`					
`restore T2's state`			100	0	50
		`mov 0x8049a1c, %eax`	105	**50**	50
		`add $0x1, %eax`	108	**51**	50
		`mov %eax, 0x8049a1c`	113	51	**51**
`interrupt`					
`save T2's state`					
`restore T1's state`			108	51	51
	`mov %eax, 0x8049a1c`		113	$\boxed{51}$	**51**

Semaphore: A definition

- An object with an integer value
  - We can manipulate with two routines: `sem_wait()` and `sem_post()`

- Initialization

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1); // initialize s to the value 1
```

  - Declare a semaphore `s` and initialize it to the value 1
  - The second argument, 0, indicates that the semaphore is shared between threads in the same process

Interact with Semaphore

- `sem_wait()`

```
int sem_wait(sem_t *s) {
    decrement the value of semaphore s by one
    wait if value of semaphore s is negative
}
```

- If the value of the semaphore was one or higher
  - Return right away
- Otherwise,
  - It will cause the caller to suspend execution and wait for a subsequent `post`
  - When negative, the value of the semaphore is equal to the number of waiting threads

Interact with Semaphore

- `sem_post()`

```
int sem_post(sem_t *s) {
     increment the value of semaphore s by one
     if there are one or more threads waiting, wake one
}
```

- Increments the value of the semaphore
- If there is a thread waiting to be woken, wakes one of them up

Semaphore as a Lock (Binary Semaphore)

- What should `X` be?
  - The initial value should be **1**

```
sem_t m;
sem_init(&m, 0, X); // initialize semaphore to X; what should X be?
sem_wait(&m);
//critical section here
sem_post(&m);
```

Value of Semaphore	Thread 0	Thread 1
1		
1	`call sema_wait()`	
0	`sem_wait() returns`	
0	`(crit sect)`	
0	`call sem_post()`	
1	`sem_post() returns`	

Semaphores As Condition Variables

```
sem_t s;

void *
child(void *arg) {
      printf("child\n");
      sem_post(&s); // signal here: child is done
      return NULL;
}

 int
 main(int argc, char *argv[]) {
      sem_init(&s, 0, X); // what should X be?
      printf("parent: begin\n");
      pthread_t c;
      pthread_create(c, NULL, child, NULL);
      sem_wait(&s); // wait here for child
      printf("parent: end\n");
      return 0;
 }
```

A Parent Waiting For Its Child

```
parent: begin
child
parent: end
```

The execution result

- What should X be?
  - The value of semaphore should be set to is **0**

Atomicity-Violation Bugs

- Two different threads access the field `proc_info` in the `struct thd`
- What is the problem with this code?

```
    Thread1:
    if(thd->proc_info){
      ...
      fputs(thd->proc_info , ...);
      ...
    }

    Thread2:
    thd->proc_info = NULL;
```

- The desired **serializability** among multiple memory accesses is violated
  - If the first thread performs the check but then is interrupted before the call to `fputs`, the second thread could run in-between, thus setting the pointer to `NULL`
  - When the first thread resumes, it will crash
    - As a `NULL` pointer will be dereferenced by `fputs`

Atomicity-Violation Bugs

- Solution?
  - Add locks around the shared-variable references

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Thread1:
pthread_mutex_lock(&lock);
if(thd->proc_info){
    …
    fputs(thd->proc_info , …);
    …
}
pthread_mutex_unlock(&lock);

Thread2:
pthread_mutex_lock(&lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&lock);
```

Privileged Programs

Need for Privileged Programs

- Password dilemma
  - Permissions of `/etc/shadow` file:

```
$> ls -l /etc/shadow

-rw-r----- 1 root shadow 1595 Dec 24 15:47 /etc/shadow
```

  - Only writable to the owner!

  - **How would normal users change their password?**

Privileged Programs

- `rwx` (read, write, and execute permissions) has an octal value of 7 (4 + 2 + 1 = 7)
- `rw-` (read and write permissions, no execute permission) has an octal value of 6 (4 + 2 + 0 = 6)

- Implementing fine-grained access control in operating systems makes OS over complicated
  - OS relies on extensions to enforce fine-grained access control
  - Privileged programs are such extensions

- Types of Privileged Programs
  - Daemons/Services
    - Computer program that runs in the background
    - Needs to run as root or other privileged users
  - Set-UID Programs
    - Widely used in UNIX systems
    - Program marked with a special bit

Conversion table for four-character octal numbers

4000 set UID on execution
0400 read by owner
0200 write by owner
0100 execute by owner
0040 read by group
0020 write by group
0010 execute by group
0004 read by other
0002 write by other
0001 execute by other

Set-UID Concept

- Allow user
  - to run a program with the program owner's privilege
  - to run programs with temporary elevated privileges
- Every process has **two** User IDs
  - Real UID (RUID): Identifies real owner of process
  - Effective UID (EUID): Identifies privilege of a process
    - Access control is based on EUID
- When a normal program is executed, RUID = EUID
  - They both equal to the ID of the user who runs the program
- When a Set-UID is executed, RUID ≠ EUID
  - RUID still equal to the user's ID, but EUID equals to the program **owner**'s ID
  - If the program is owned by `root`, the program runs with the `root` privilege

Example of Set-UID

```
$> cp /bin/cat mycat
$> sudo chown root mycat
$> ls -l mycat
-rwxr-xr-x 1 root cougs 35080 Jan  5 16:52 mycat
$> mycat /etc/shadow
mycat: /etc/shadow: Permission denied
```

Not a privileged program

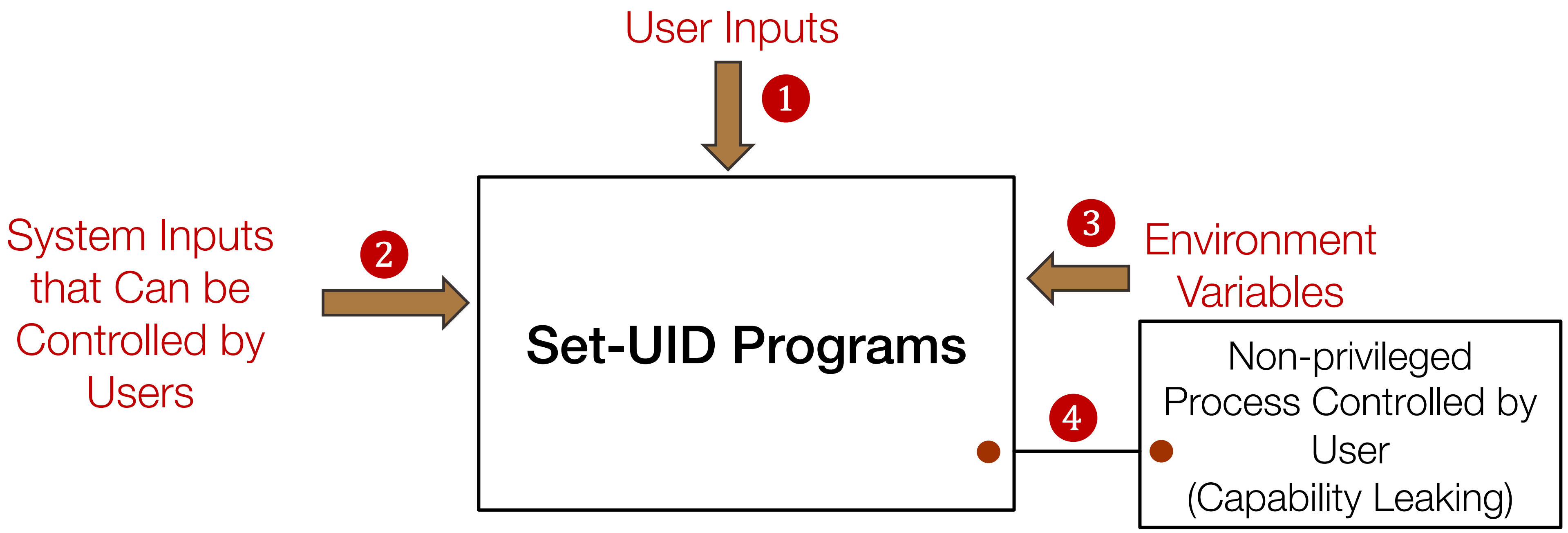
```
$> sudo chmod 4755 mycat
$> mycat /etc/shadow
root:*:18863:0:99999:7:::
...
...
```

Become a privileged program

```
$> sudo chown cougs mycat
$> chmod 4755 mycat
$> mycat /etc/shadow
mycat: /etc/shadow: Permission denied
```

Still a privileged program, but not the `root` privilege

Vulnerabilities of Set-UID Programs



Attacks via Environment Variables

- Consider the following code

```c
/* The vulnerable program (vul.c) */
#include <stdlib.h>
int main() {
    system("cal") ;
}
```

vul.c

- We will force the above program to execute the following program

```c
/* Malicious calendar program (cal.c) */
#include <stdlib.h>
int main() {
    system("/bin/dash") ;
}
```

cal.c

Attacks via Environment Variables

```
$> gcc -o vul vul.c
$> sudo chown root vul
$> sudo chmod 4755 vul
$> vul
December 2015
Su Mo Tu We Th Fr Sa
1  2  3  45
6  7  8  9  10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
$> gcc -o cal cal.c
$> export PATH=.: $PATH
$> echo $PATH
.:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:...
$> vul
#
# id
uid=1000 (cougs) gid=1000 (cougs) euid=0 (root)
```

1 First run the first program without doing the attack

2 Now change the `PATH` environment variable

Got a `root` shell!

Capability Leaking

- In some cases, Privileged programs downgrade themselves during execution
- Example: The `su` program
  - This is a privileged Set-UID program
  - Allows one user to switch to another user (say user1 to user2)
  - Program starts with EUID as root and RUID as user1
  - After password verification, both EUID and RUID become user2's (via privilege downgrading)
- Such programs may lead to capability leaking
  - Programs may not clean up privileged capabilities before downgrading

Attacks via Capability Leaking

The `/etc/zzz` file is only writable by `root`

File descriptor is created (the program is a root-owned Set-UID program)

The privilege is downgraded

Invoke a shell program, so the behavior restriction on the program is lifted

```c
fd = open("/etc/zzz", O_RDWR | O_APPEND);
if (fd == -1) {
  printf("Cannot open /etc/zzz\n");
  exit(0);
}

// print out the file descriptor value
printf("fd is %d\n", fd);

// permanently disable privilege by
// making the effective uid the same as the real uid
setuid(getuid());

// execute /bin/sh
v[0] = "/bin/sh"; v[1] = 0; v[2] = 0;
execve(v[0], v, 0);
```

Attacks via Capability Leaking

The program forgets to close the file → file descriptor is still valid

↓

Capability Leak!

```
$> gcc -o cap_leak cap_leak.c
$> sudo chown root cap_leak
$> sudo chmod 4755 cap_leak
$ ls -l cap_leak
-rwsr-xr-x 1 root cougs 7386 Jan 5 09:24 cap_leak
$> cat /etc/zzz
bbbbbbbbbbbbbbb
$> echo aaaaaaaaaa > /etc/zzz
bash: /etc/zzz: Permission denied  ← Cannot write to the file
$> cap_leak
fd is 3
$> echo cccccccccccc >& 3        ← Using the leaked capability
$> exit
$> cat /etc/zzz
bbbbbbbbbbbbbbb
cccccccccccc                      ← File modified
```

Invoking Programs : Unsafe Approach

- Scenario: use `cat` to view files (view only, NOT writable)

- The easiest way:
  - Invoke an external command is the `system()` function
  - This program uses the `/bin/cat` program
  - Should be a root-owned Set-UID program (to view restricted files)
  - The program can view all files, but it can't write to any file

Invoking Programs: Unsafe Approach

- Scenario: use `cat` to view files (view only, NOT writable)

```c
// use cat to view files (NOT writable)
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
  char *cat = "/bin/cat";
  if (argc < 2) {
    printf("Enter a filename.\n");return 1;
  }
  char *command = malloc(strlen(cat) + strlen(argv[1]) + 2);
  sprintf(command, "%s %s", cat, argv[1]);
  system(command);
  return 0;
}
```

catall.c

Invoking Programs: Unsafe Approach

```
$> gcc -o catall catall.c
$> sudo chown root catall
$> sudo chmod 4755 catall
$> ls -l catall
-rwsr-xr-x 1 root cougs 7275 Dec 29 09:41 catall
$> catall /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWb...
daemon:*:15749:0:99999:7:::
bin:*:15749:0:99999:7:::
...
```

Invoking Programs: Unsafe Approach

- Scenario: use `cat` to view files (view only, NOT writable)

Can we use this program to run other command, with the `root` privilege?

YES!

```
$> catall "aa;/bin/sh"
/bin/cat: aa: No such file or directory
#                                        🡐 Got the root shell!
# id
uid=1000(cougs) gid=1000(cougs) euid=0(root) groups=0(root), ...
```

`catall.c`

Invoking Programs Safely: Using `execve()`

`execve(v[0], v, 0)`

Command name is provided by the **program**

Input data are provided by the **user**

```c
// safecatall.c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
  char *v[3];
  if (argc < 2) {
  printf("Enter a filename.\n"); return 1;
}
  v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = 0;
  execve(v[0], v, 0);
  return 0;
}
```

Why is it safe?

- Code (command name) and data are clearly separated
- There is no way for the user data to become code

Set-UID & Dynamic Linking

```c
// mytest.c
#include <stdio.h>

int main()
{
  sleep(1);
  return 0;
}
```

```
$> gcc -o mytest mytest.c
$> ./mytest
$>
```

```c
// sleep.c
#include <stdio.h>
void sleep(int s)
{
  printf("I am not sleeping!\n");
}
```

```
$> gcc -c sleep.c
$> gcc -shared -o libmylib.so.1.0.1 sleep.o
$> export LD_PRELOAD=./libmylib.so.1.0.1
```

Now we implement our own `sleep()` function

```
$> sudo chown root mytest
$> sudo chmod 4755 mytest
$> ls -l mytest
-rwsr-xr-x 1 root cougs 7161 Dec 27 08:35 mytest
$> export LD_PRELOAD=./libmylib.so.1.0.1
$> ./mytest
$>
```

Set-UID & Dynamic Linking

- Our `sleep()` function was not invoked!
  - This is due to a countermeasure implemented by the dynamic linker
  - It ignores the `LD_PRELOAD` and `LD_LIBRARY_PATH` environment variables when the EUID and RUID differ

Privileged Program: Remarks

- **Principle of isolation:** *Don't mix code and data*
  - `system()` code execution
  - Buffer overflow attacks
  - Cross site scripting
  - SQL injection

  CPTS 327!

- **Principle of least privilege:** *A privileged program should be given the power which is required to perform its tasks*
  - Disable the privileges (temporarily or permanently) when a privileged program doesn't need those
  - In Linux, `seteuid()` and `setuid()` can be used to disable/discard privileges

That's all for today

Good Luck with the Finals!