Section 1: OS Concepts, Processes, Threads

CS 162

January 31, 2020

Contents

1 Vocabulary

With credit to the Anderson & Dahlin textbook (A&D):

- **process** - A process is an instance of a computer program that is being executed, typically with restricted rights. It consists of an address space and one or more threads of control. It is the main abstraction for protection provided by the operating system kernel.

- **thread** - A thread is a single execution sequence that can be managed independently by the operating system. (See A&D, 4.2)

- **isolation** - Isolating (separating) applications from one another so that a potentially misbehaving application cannot corrupt other applications or the operating system.

- **dual-mode operation** - Dual-mode operation refers to hardware support for multiple privilege levels: a privileged level (called *supervisor-mode* or *kernel-mode*) that provides unrestricted access to the hardware, and a restricted level (called *user-mode*) that executes code with restricted rights.

- **privileged instruction** - Instruction available in kernel mode but not in user mode. Two examples of privileged instructions are the instructions to enable and disable interrupts on the processor. If user-level code could disable interrupts, it would guarantee that the user-level process could run on a hardware thread for as long as it wanted.

- **unprivileged instruction** - Instruction available in both user mode and kernel mode. An example of an unprivileged instruction is the `add` instruction or the instructions that read or write to memory. User-level processes are allowed to perform these standard operations that all computer programs need in order to run.

- **fork** - A C function that calls the fork syscall that creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process (except for a few details, read more in the man page). Both the newly created process and the parent process return from the call to fork. On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created.

- **wait** - A class of C functions that call syscalls that are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal.

- **exit code** - The exit status or return code of a process is a 1 byte number passed from a child process (or callee) to a parent process (or caller) when it has finished executing a specific procedure or delegated task

- **exec** - The exec() family of functions replaces the current process image with a new process image. The initial argument for these functions is the name of a file that is to be executed.

- **pthreads** - A POSIX-compliant (standard specified by IEEE) implementation of threads. A `pthread_t` is usually just an alias for "`unsigned long int`".

- **pthread_create** - Creates and immediately starts a child thread running in the same address space of the thread that spawned it. The child executes starting from the function specified. Internally,

this is implemented by calling the clone syscall.

```
/* On success, pthread_create() returns 0; on error, it returns an error
 * number, and the contents of *thread are undefined. */
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

- **pthread_join** - Waits for a specific thread to terminate, similar to `waitpid(3)`.

```
/* On success, pthread_join() returns 0; on error, it returns an error number. */
int pthread_join(pthread_t thread, void **retval);
```

- **critical section** - A section of code that accesses a shared resource and must not be concurrently run by more than a single thread.

- **race condition** - A situation whose outcome is dependent on the sequence of execution of multiple threads running simultaneously.

- **lock** - Synchronization primitives that provide mutual exclusion. Threads may acquire or release a lock. Only one thread may hold a lock at a time. If a thread attempts to acquire a lock that is held by some other thread, it will block at that line of code until the lock is released and it successfully acquires it. Implementations can vary.

2 Fundamental Operating System Concepts

1. What are the 3 roles the OS plays?

2. How is a process different from a thread?

3. What is the process address space and address translation? Why are they important?

4. What is dual mode operation and what are the three forms of control transfer from user to kernel mode?

5. Why does a thread in kernel mode have a separate kernel stack? What can happen if the kernel stack was in the user address space?

6. How does the syscall handler protect the kernel from corrupt or malicious user code?

3 Processes

3.1 Forks

How many new processes are created in the below program assuming calls to fork succeeds?

```
int main(void)
{
  for (int i = 0; i < 3; i++)
      pid_t pid = fork();
}
```

3.2 Process Stack Allocation

What can C print?

```
int main(void)
{
    int stuff = 5;
    pid_t pid = fork();
    printf("The last digit of pi is %d\n", stuff);
    if (pid == 0)
        stuff = 6;
}
```

3.3 Process Heap Allocation

What can C print?

```
int main(void)
{
    int* stuff = malloc(sizeof(int)*1);
    *stuff = 5;
    pid_t pid = fork();
    printf("The last digit of pi is %d\n", *stuff);
    if (pid == 0)
        *stuff = 6
}
```

3.4 Simple Wait

What can C print? Assume the child PID is 90210.

```
int main(void)
{
    pid_t pid = fork();
    int exit;
    if (pid != 0) {
        wait(&exit);
    }
    printf("Hello World\n: %d\n", pid);
}
```

3.5 Exec

What will C print?

```
int main(void)
{
    char** argv = (char**) malloc(3*sizeof(char*));
    argv[0] = "/bin/ls";
    argv[1] = ".";
    argv[2] = NULL;
    for (int i = 0; i < 10; i++) {
        printf("%d\n", i);
        if (i == 3)
            execv("/bin/ls", argv);
    }
}
```

3.6 Exec + Fork

How would I modify the above program using fork so it both prints the output of `ls` and all the numbers from 0 to 9 (order does not matter)? You may not remove lines from the original program; only add statements (and use fork!).

4 Threads

4.1 Join

What does C print in the following code?
(Hint: There may be zero, one, or multiple answers.)

```
void *helper(void *arg) {
    printf("HELPER\n");
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    printf("MAIN\n");
    return 0;
}
```

How can we modify the code above to always print out `"HELPER"` followed by `"MAIN"`?

4.2 Thread Stack Allocation

What does C print in the following code?

```
void *helper(void *arg) {
    int *num = (int*) arg;
    *num = 2;
    return NULL;
}

int main() {
    int i = 0;
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, &i);
    pthread_join(thread, NULL);
    printf("i is %d\n", i);
    return 0;
}
```

4.3 Thread Heap Allocation

What does C print in the following code?

```
void *helper(void *arg) {
    char *message = (char *) arg;
    strcpy(message, "I am the child");
    return NULL;
}

int main() {
    char *message = malloc(100);
    strcpy(message, "I am the parent");
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, message);
    pthread_join(thread, NULL);
    printf("%s\n", message);
    return 0;
}
```

4.4 The Central Galactic Floopy Corporation

It's the year 3162. Floopies are the widely recognized galactic currency. Floopies are represented in digital form only, at the Central Galactic Floopy Corporation (CGFC).

You receive some inside intel from the CGFC that they have a Galaxynet server running on some old OS called x86 Ubuntu 14.04 LTS. Anyone can send requests to it. Upon receiving a request, the server forks a POSIX thread to handle the request. In particular, you are told that sending a transfer request will create a thread that will run the following function immediately, for speedy service.

```
void transfer(account_t *donor, account_t *recipient, float amount) {
  assert (donor != recipient); // Thanks CS161

  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

Assume that there is some struct with a member `balance` that is `typedef`-ed as `account_t`. Describe how a malicious user might exploit some unintended behavior.

Since you're a good person who wouldn't steal floopies from a galactic corporation, what changes would you suggest to the CGFC to defend against this exploit?

5 Interrupt Handlers

Refer to the "Pintos Interrupt Handler" section at the end of this discussion worksheet to answer these questions:

What do the instructions `pushal` and `popal` do?

The interrupt service routine (ISR) must run with the kernel's stack. Why is this the case? And which instruction is responsible for switching the stack pointer to the kernel stack?

The `pushal` instruction pushes 8 values onto the stack (32 bytes). With this information, please draw the stack at the moment when "`call intr_handler`" is about to be executed.

What is the purpose of the "`pushl %esp`" instruction that is right before "`call intr_handler`"?

Inside the `intr_exit` function, what would happen if we reversed the order of the 5 `pop` instructions?

5.1 Pintos Interrupt Handler

```
/**
* An example of an entry point that would reside in the interrupt
* vector. This entry point is for interrupt number 0x30.
*/
.func intr30_stub
intr30_stub:
      pushl %ebp     /* Frame pointer */
      pushl $0       /* Error code */
      pushl $0x30    /* Interrupt vector number */
      jmp intr_entry
.endfunc
/* Main interrupt entry point.

   An internal or external interrupt starts in one of the
   intrNN_stub routines, which push the 'struct intr_frame'
   frame_pointer, error_code, and vec_no members on the stack,
   then jump here.

   We save the rest of the 'struct intr_frame' members to the
   stack, set up some registers as needed by the kernel, and then
   call intr_handler(), which actually handles the interrupt.

   We "fall through" to intr_exit to return from the interrupt.
*/
.func intr_entry
intr_entry:
        /* Save caller's registers. */
        pushl %ds
        pushl %es
        pushl %fs
        pushl %gs
        pushal

        /* Set up kernel environment. */
        cld                     /* String instructions go upward. */
        mov $SEL_KDSEG, %eax    /* Initialize segment registers. */
        mov %eax, %ds
        mov %eax, %es
        leal 56(%esp), %ebp     /* Set up frame pointer. */

        /* Call interrupt handler. */
        pushl %esp
.globl intr_handler
        call intr_handler
        addl $4, %esp
.endfunc
```

```
48 /* Interrupt exit.
49
50    Restores the caller's registers, discards extra data on the
51    stack, and returns to the caller.
52
53    This is a separate function because it is called directly when
54    we launch a new user process (see start_process() in
55    userprog/process.c). */
56 .globl intr_exit
57 .func intr_exit
58 intr_exit:
59         /* Restore caller's registers. */
60         popal
61         popl %gs
62         popl %fs
63         popl %es
64         popl %ds
65
66         /* Discard `struct intr_frame' vec_no, error_code,
67            frame_pointer members. */
68         addl $12, %esp
69
70         /* Return to caller. */
71         iret
72 .endfunc
```