

# Section 1: OS Concepts, Processes, Threads

CS 162

September 4, 2020

## Contents

<b>1</b>	<b>Vocabulary</b>	<b>2</b>
<b>2</b>	<b>Warmup</b>	<b>4</b>
2.1	Pointer and C Programming Practice . . . . .	4
<b>3</b>	<b>Fundamental Operating System Concepts</b>	<b>5</b>
<b>4</b>	<b>Processes</b>	<b>7</b>
4.1	Forks . . . . .	7
4.2	Process Stack Allocation . . . . .	7
4.3	Process Heap Allocation . . . . .	8
4.4	Simple Wait . . . . .	8
4.5	Exec . . . . .	9
4.6	Exec + Fork . . . . .	9
<b>5</b>	<b>Threads</b>	<b>10</b>
5.1	Join . . . . .	10
5.2	Thread Stack Allocation . . . . .	11
5.3	Thread Heap Allocation . . . . .	11
<b>6</b>	<b>Pintos Lists</b>	<b>12</b>
<b>7</b>	<b>Interrupt Handlers</b>	<b>14</b>
7.1	Pintos Interrupt Handler . . . . .	15

# 1 Vocabulary

- **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);
```

- **pthread\_yield** - Equivalent to `thread_yield()` in Pintos. Causes the calling thread to vacate the CPU and go back into the ready queue without blocking. The calling thread is able to be scheduled again immediately. This is not the same as an interrupt and will succeed in Pintos even if interrupts are disabled.

```
/* On success, pthread_yield() returns 0; on error, it returns an error number. */  
int pthread_yield(void);
```

## 2 Warmup

### 2.1 Pointer and C Programming Practice

Write a function that places source inside of dest, starting at the offset position of dest. This is effectively swapping the tail-end of dest with the string contained in source (including the null terminator). Assume both are null-terminated and the programmer will never overflow dest. As an exercise in using pointers, implement it **without using libraries**.

```
void replace(char *dest, char *source, int offset)
{

}
}
```

### 3 Fundamental Operating System Concepts

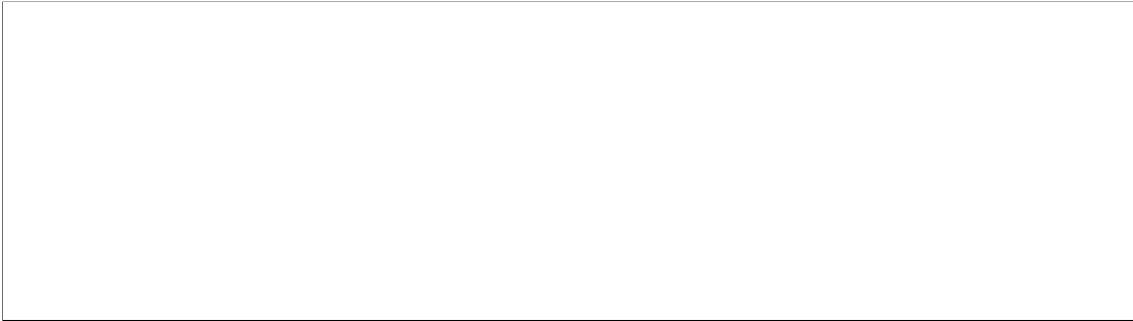
1. What are the 3 roles the OS plays?

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?



## 4 Processes

### 4.1 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();
    return 0;
}
```

### 4.2 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;
    return 0;
}
```

### 4.3 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;
    return 0;
}
```

### 4.4 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: %d\n", pid);
    return 0;
}
```



## 4.5 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);
    }
    return 0;
}
```

## 4.6 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!).

## 5 Threads

### 5.1 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"?

## 5.2 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;
}
```

## 5.3 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;
}
```

## 6 Pintos Lists

This section is intended to help you get more familiar with the pintos list abstraction, which will be used heavily in all three projects, as well as in homework 1. Consider the following code, which finds the sum of a traditional linked-list:

```
struct ll_node
{
    int value;
    struct ll_node *next;
};

/* Returns the sum of a linked list. */
int ll_sum(ll_node *start) {
    ll_node *iter;

    int total = 0;
    for (iter = start; iter != NULL; iter = iter->next)
        total += iter->value;

    return total;
}
```

Take a second to make sure you understand the structure of the for-loop, as this kind of iteration is key when dealing with linked lists.

Write code below that emulates the above code, but for pintos-style lists. That is, write a function that finds the sum of a pintos-style list. Some useful methods are listed below.

```
struct pl_node
{
    int value;
    struct list_elem elem;
};

/* Returns the sum of a pintos-style list of pl_nodes. */
int pl_sum(struct list *lst) {
    struct list_elem *iter;
    struct pl_node *temp;

    int total = 0;
```

```

    return total;
}
```

Here are some useful helper functions for pintos lists:

```
/* Given a struct list, returns a reference to the
 * first list_elem in the list. */
struct list_elem *list_begin(struct list *lst);

/* Given a struct list, returns a reference to the
 * last list_elem in the list. */
struct list_elem *list_end(struct list *lst);

/* Given a list_elem, finds the next list_elem in the list. */
struct list_elem *list_next(struct list_elem *elem);

/* Converts pointer to list element LIST_ELEM into a pointer to the
 * structure that LIST_ELEM is embedded inside. You must also
 * provide the name of the outer structure STRUCT and the member
 * name MEMBER of the list element. */
STRUCT *list_entry(LIST_ELEM, STRUCT, MEMBER);
```

Note that because `list_entry()` is actually defined as a preprocessor macro, it doesn't follow the normal rules of C functions, and introduces some interesting polymorphism.

If you need more help with the pintos list abstraction, check out the documentation in the pintos source code at `lib/kernel/list.h`. The documentation is very comprehensive, and you should refer to it as you do more with pintos lists.

## 7 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?

## 7.1 Pintos Interrupt Handler

```

1 /**
2  * An example of an entry point that would reside in the interrupt
3  * vector. This entry point is for interrupt number 0x30.
4  */
5  .func intr30_stub
6  intr30_stub:
7      pushl %ebp      /* Frame pointer */
8      pushl $0        /* Error code */
9      pushl $0x30     /* Interrupt vector number */
10     jmp intr_entry
11 .endfunc
12 /* Main interrupt entry point.
13
14  An internal or external interrupt starts in one of the
15  intrNN_stub routines, which push the 'struct intr_frame'
16  frame_pointer, error_code, and vec_no members on the stack,
17  then jump here.
18
19  We save the rest of the 'struct intr_frame' members to the
20  stack, set up some registers as needed by the kernel, and then
21  call intr_handler(), which actually handles the interrupt.
22
23  We "fall through" to intr_exit to return from the interrupt.
24  */
25  .func intr_entry
26  intr_entry:
27      /* Save caller's registers. */
28      pushl %ds
29      pushl %es
30      pushl %fs
31      pushl %gs
32      pushal
33
34      /* Set up kernel environment. */
35      cld                /* String instructions go upward. */
36      mov $SEL_KDSEG, %eax /* Initialize segment registers. */
37      mov %eax, %ds
38      mov %eax, %es
39      leal 56(%esp), %ebp /* Set up frame pointer. */
40
41      /* Call interrupt handler. */
42      pushl %esp
43  .globl intr_handler
44      call intr_handler
45      addl $4, %esp
46  .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
```