

Discussion 1: Operating Systems

January 28, 2022

Contents

1	Fundamentals of Operating Systems	2
1.1	Concept Check	3
2	Processes	5
2.1	Fork and Friends	6
2.2	Signal Handling	8
3	Pintos Lists	10

1 Fundamentals of Operating Systems

While not a well-defined concept, **operating systems (OS)** provide hardware abstractions (e.g. file systems, processes) to software applications and manage hardware resources (e.g. memory, CPU). It can be seen as a special layer of software that provides application software access to hardware resources. An OS serves its purpose by taking on three roles.

Role	Purpose
Referee	Manage protection, isolation, and sharing of resources
Illusionist	Provide clean, easy-to-use abstractions of physical resources
Glue	Provide common services

Thread

A **thread** is a single unique execution context consisting of the program counter, registers, execution flags, and stack. These hold information regarding the context of the thread. For instance, the stack pointer register holds the address of the top of the stack.

A thread is **executing** on a processor when it is **resident** in the processor's registers. For instance, the program counter register of the processor would point to the next instruction for the executing thread. When a thread is **suspended** on a processor, a thread's context is stored elsewhere since another thread may execute on that processor. The information is saved in memory as part of a data structure called the **thread control block (TCB)**.

Threads provide an illusion of multiple processors. From a programmer's perspective, threads can be seen as **virtual cores** which makes it seem as though there are multiple cores on a single processor.

Address Space

An **address space** is the set of accessible addresses and the state associated with them. For a 32-bit processor, there are $2^{32} \approx 4$ billion addresses.

The entire address space doesn't necessarily represent real locations but rather potential spaces. A user program reading or writing to an address may result in a regular memory access, exception or fault if not allowed due to protection, I/O operation from a memory-mapped I/O device, or more. However, the kernel can access the entire address space without limitations.

In most modern operating systems, programs operate with **virtual memory**. Instead of accessing a physical memory directly, programs will request to access memory add a virtual memory address which is translated into a physical memory address.

Process

A **process** is an execution environment with restricted rights. Each process has its own address space along with one or more threads which share the process's resources such as code, global data, and files.

Processes are protected from each other due to differing address spaces, resulting in a fundamental tradeoff between protection and efficiency. Communication between threads within the same process is quick, but the threads are not protected from each other. On the other hand, communication between threads of different processes is slow, but the threads are protected from each other.

This shows that threads embody the idea of concurrency, while the address spaces and processes provide protection. If a bug was to corrupt a process, it would generally avoid compromising the entire system due to the protections from the address space.

Dual Mode Operation

The underlying hardware that the OS interfaces with provides at least two modes: **kernel** and **user**. Kernel mode, also known as supervisor or privileged mode, has the most privileges, meaning the kernel itself and other parts of the OS operate in this mode. On the other hand, user mode prohibits certain operations. This is where programs are normally executed. The restricted access is important in user mode to make sure processes running in user mode cannot maliciously corrupt another process.

There are three main ways the system switches from a user mode to kernel mode (**mode transfer**). When processes request a system service, this is known as a **system call (syscall)**. While similar to a function call, syscalls occur "outside" the process, meaning it is executed in the kernel mode. Therefore, syscalls encompass functionality that requires the privileges or abstractions of being in the kernel mode. **Interrupts**, sometimes referred to as hardware interrupts, are external asynchronous events that trigger a context switch (e.g. timer, I/O). **Traps**, also known as **exceptions** or software interrupts, are internal synchronous events that trigger a context switch (e.g. segmentation fault, divide by zero).

All three types of mode transfers are **unprogrammed control transfers**. Instead of the process specifying the specific address like in a regular function call, the process specifies the an index into the **interrupt vector table (IVT)**, which is a table that contains the address and properties of each interrupt handler. The interrupt in IVT is used as a general term that's just not limited to the interrupts mentioned in the aforementioned paragraph. The location of the IVT is stored in a designated processor register.

1.1 Concept Check

1. Between threads and processes, which encapsulates the idea of isolation? What idea does the other one encapsulate?

Processes encapsulate the idea of isolation, while threads encapsulate the idea of concurrency.

2. What is the importance of address translation?

Address translation is necessary for the idea of virtual memory which provides an isolation abstraction. This gives the illusion that each process is the sole user of the address space. It also provides protection between different processes since virtual addresses will not translate to the same physical address, preventing processes from accessing and potentially corrupting each other's memory.

3. Similar to what's done in the prologue at calling convention, what needs to happen before a mode transfer occurs?

The processor's state (e.g. registers) need to be saved in the process control block (PCB) because the kernel may overwrite the registers when it executes its own code.

4. Why does a thread in kernel mode have a separate kernel stack?

Each OS thread has a kernel stack (located in kernel memory) plus an user stack (located in user memory). This kernel stack is memory that is only accessible by the thread in privileged kernel mode and thus provides protection for the kernel from the user.

If the user process had write permission to the kernel stack, it could corrupt the kernel in the event that the user purposely or accidentally overwrite the kernel stack. Some OSs keep the PCB in the kernel stack, which if the user had access to can allow for permissions changes information leakage, and breaking of process isolation.

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

When executing a syscall, the user program specifies an index instead of the direct address of the handler, meaning the user program cannot directly execute in kernel mode. The arguments will be validated by the handler to make sure the user is not intending a malicious attack. Moreover, the handler will copy over the arguments instead of using them from the user stack directly. This is necessary because the user program could change the arguments after the handler performs initial checks for malicious purposes (i.e. [TOCTTOU^a](#)). After the syscall finishes, the results are copied back in to user memory. The user process is not allowed to access the results stored in kernel memory for security reasons.

^ahttps://en.wikipedia.org/wiki/Time-of-check_to_time-of-use

2 Processes

Processes are an instance of a running program with their own protected address spaces and one or more threads.

Process Control Block

An OS needs to run many programs, meaning there will be many processes. As a result, basic mechanisms such as switching between user processes and the kernel, switching among user processes through the kernel, and protecting the OS from user processes and among themselves need to be present in an operating system.

To accomplish this, the kernel represents each process with a **process control block (PCB)**, a data structure that keeps track of the various properties of a process including (but not limited to) status, register state, process id (pid). The kernel scheduler allocates the CPU to different processes by maintaining a data structure containing the aforementioned PCBs. Other resources, such as memory and I/O, need to be managed and allocated as well, though not necessarily by the kernel scheduler.

Syscalls

Generally, an OS provides a library or API that implements process management syscalls. For Unix-based operating systems, this usually resides as part of the C standard library (libc). As a result, full documentation is available through the `man` pages.

`void exit(int status)` terminates the calling process with the exit code specified by `status`. Generally speaking, exit code 0 means the code executed without any error, while non-zero exit codes indicate otherwise. Rarely, you will see a C program where `main` directly calls `exit` since the OS library will call `exit` for you once `main` returns.

`pid_t fork(void)` creates a new process by copying the current process. Typically, the process *created* from `fork` is known as the **child process**, while the process *calling* `fork` is known as the **parent process**. The parent and child processes are identical in many ways (e.g. address space) except for the PID. The child process also only has one thread even if the parent had multiple. More differences are listed on the `man` pages. The return type of `fork` is `pid_t`, which is a signed integer. If the return value is greater than 0, this means the current process is the parent process. On the other hand, if the return value is 0, this means the current process is the child process. When the return value is -1, an error has occurred; the current process remains (i.e. no new child process has been created).

A typical workflow you might see is

```
1 int main() {
2     pid_t fork_ret = fork();
3     if (fork_ret > 0) {
4         /* parent process logic */
5     } else if (fork_ret == 0) {
6         /* child process logic */
7     } else {
8         /* error handling */
9     }
10 }
```

fork_simple.c

`exec` change the program being run by the current process. A key distinction is that unlike `fork`, `exec` does not create a new process. In Unix-based operating systems, `exec` is a family of functions with different parameters, which is why a full method header is not given. For a full list, visit the `man` page.

`pid_t wait(int *wstatus)` waits for a child process to finish. On success, it returns the PID of the terminated child process, while returning -1 on an error. `wstatus` is used to store status information if not NULL.

`int kill(pid_t pid, int sig)` sends a signal, an interrupt-like notification, to another process. `SIGINT` (ctrl-C), `SIGTERM` (kill on command line), and `SIGSTOP` (ctrl-Z) are all types of signals you may be familiar with. When a process receives a signal, it has a defined behavior known as the signal handler. A custom signal handler can be defined for most signals except for `SIGKILL` and `SIGSTOP` using `sigaction`.

2.1 Fork and Friends

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

```
1 int main(void) {
2     for (int i = 0; i < 3; i++)
3         pid_t fork_ret = fork();
4     return 0;
5 }
```

fork_loop.c

7. Newly forked processes will continue to execute the loop from wherever the parent process left off.

2. What are the possible outputs when the following program is run?

```
1 int main(void) {
2     int stuff = 5;
3     pid_t fork_ret = fork();
4     printf("The last digit of pi is %d\n", stuff);
5     if (fork_ret == 0)
6         stuff = 6;
7     return 0;
8 }
```

fork_stack.c

`fork` creates a new process, meaning the address spaces are identical but not shared. As a result, changing `stuff` in the child process will have no effect on the value of `stuff` in the parent process, regardless of the execution order of the two processes. This means the following will be printed.

```
The last digit of pi is 5.
The last digit of pi is 5.
```

However, it is possible the `fork` doesn't succeed as no such assumption was made. As a result, the child process would never be created, only resulting in one statement being printed.

3. What are the possible outputs when the following program is run?

```
1 int main(void) {
2     int* stuff = malloc(sizeof(int));
3     *stuff = 5;
4     pid_t fork_ret = fork();
5     printf("The last digit of pi is %d\n", *stuff);
6     if (fork_ret == 0)
7         *stuff = 6;
8     return 0;
9 }
```

fork_heap.c

The heap is part of the address space like the stack, meaning the heap will not be shared between the parent and child process after a `fork`. As a result, the possible outputs are the exact same as the previous question: two lines of `The last digit of pi is 5` if `fork` succeeds, one line otherwise.

4. What are the possible outputs when the following program is run? Assume the child process has PID 162162.

```

1 int main(void) {
2     pid_t fork_ret = fork();
3     int exit;
4     if (fork_ret != 0)
5         wait(&exit)
6     printf("Hello World: %d\n", fork_ret);
7     return 0;
8 }
```

fork_wait.c

The parent process will wait until the child process completes, meaning it will not print until after the child process. As a result, the program will output

```
Hello World: 0
Hello World: 162162
```

Note that the order matters here.

If `fork` fails, then the program will print

```
Hello World: -1
```

since `fork` returns -1.

5. Does the following program print all numbers from 0 to 9 as well as the output of running `ls`? If not, what is the minimal code change to accomplish this? Assume all syscalls succeed.

```

1 int main(void) {
2     char** argv = (char**) malloc(3 * sizeof(char*));
3     argv[0] = "/bin/ls";
4     argv[1] = ".";
5     argv[2] = NULL;
6     for (int i = 0; i < 10; i++) {
7         printf("%d\n", i);
8         if (i == 3) {
9             execv("/bin/ls", argv);
10        }
11    }
12    return 0;
13 }
```

fork_exec.c

Currently, the program stops after printing 3, giving an output of

```
0
1
2
```

3
<output of ls>

When `execv` is run, the entire process image is overwritten, meaning the rest of the loop will not execute. To achieve the desired output, we can fork and `exec` only in the child process to make sure the parent process continues the loop.

```

1 int main(void) {
2     char** argv = (char**) malloc(3 * sizeof(char*));
3     argv[0] = "/bin/ls";
4     argv[1] = ".";
5     argv[2] = NULL;
6     for (int i = 0; i < 10; i++) {
7         printf("%d\n", i);
8         if (i == 3) {
9             pid_t fork_ret = fork();
10            if (fork_ret == 0)
11                execv("/bin/ls", argv);
12        }
13    }
14    return 0;
15 }

```

fork_exec.fixed.c

2.2 Signal Handling

The following are a list of standard POSIX signals.

Signal	Value	Action	Comment
SIGHUP	1	Terminate	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Terminate	Interrupt from keyboard (Ctrl - c)
SIGQUIT	3	Core Dump	Quit from keyboard (Ctrl - \)
SIGILL	4	Core Dump	Illegal Instruction
SIGABRT	6	Core Dump	Abort signal from abort(3)
SIGFPE	8	Core Dump	Floating point exception
SIGKILL	9	Terminate	Kill signal
SIGSEGV	11	Core Dump	Invalid memory reference
SIGPIPE	13	Terminate	Broken pipe: write to pipe with no readers
SIGALRM	14	Terminate	Timer signal from alarm(2)
SIGTERM	15	Terminate	Termination signal
SIGUSR1	30,10,16	Terminate	User-defined signal 1
SIGUSR2	31,12,17	Terminate	User-defined signal 2
SIGCHLD	20,17,18	Ignore	Child stopped or terminated
SIGCONT	19,18,25	Continue	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

1. Overriding SIGSTOP and SIGKILL is disabled. Why?

If a process were to override their signal handlers to ignore SIGSTOP and SIGKILL, it can run a malicious process forever, posing substantial threats to the operating system and rest of the processes.

2. What are the different ways you can use the keyboard to cause the program to exit? Assume the program is run in a bash shell.

```
1 void sigquit_handler(int sig) {
2     if (sig == SIGINT || sig == SIGQUIT)
3         exit(1);
4 }
5 void sigint_handler(int sig) {
6     if (sig == SIGINT)
7         signal(SIGINT, sigquit_handler);
8 }
9 int main() {
10    signal(SIGQUIT, sigquit_handler);
11    signal(SIGINT, sigint_handler);
12    while (1) {
13        printf("Sleeping for a second ...\n");
14        sleep(1);
15    }
16 }
```

exit_signals.c

In main, we initialize SIGINT and SIGQUIT to custom handlers. SIGQUIT's new handler exits after checking sig, so pressing Ctrl - \ will cause the program to exit. When the program receives SIGINT the first time, it will redefine the signal handler to be sigquit_handler. As a result, Ctrl - C needs to be pressed twice or Ctrl-C and Ctrl - \.

3 Pintos Lists

Consider the following code, which sums up the items of a traditional linked list.

```

1 struct ll_node {
2     int value;
3     struct ll_node *next;
4 };
5
6 /* Returns the sum of a linked list. */
7 int ll_sum(ll_node *start) {
8     ll_node *iter;
9
10    int total = 0;
11    for (iter = start; iter != NULL; iter = iter->next)
12        total += iter->value;
13
14    return total;
15 }
```

ll.c

Fill in the blanks below that emulates the above code but for Pintos lists (i.e. a list summer).

```

1 /* Given a struct list, returns a reference to the first list_elem in the list. */
2 struct list_elem *list_begin(struct list *lst);
3
4 /* Given a struct list, returns a reference to the last list_elem in the list. */
5 struct list_elem *list_end(struct list *lst);
6
7 /* Given a list_elem, finds the next list_elem in the list. */
8 struct list_elem *list_next(struct list_elem *elem);
9
10 /* Converts pointer to list element LIST_ELEM into a pointer to the structure
11    that LIST_ELEM is embedded inside. You must also provide the name of the
12    outer structure STRUCT and the member name MEMBER of the list element. */
13 STRUCT *list_entry(LIST_ELEM, STRUCT, MEMBER);
14
15 struct pl_node {
16     int value;
17     struct list_elem elem;
18 };
19
20 /* Returns the sum of a pintos-style list of pl_nodes. */
21 int pl_sum(struct list *lst) {
22     struct list_elem *iter;
23     struct pl_node *temp;
24
25     int total = 0;
26
27     for (_____ ; _____ ; _____) {
28         temp = list_entry(_____);
29         _____;
30     }
31
32     return total;
33 }
```

pl.c

```
1 for (iter = list_begin(lst); iter != list_end(lst); iter = list_next(iter)) {  
2     temp = list_entry(iter, struct pl_node, elem);  
3     total += temp->value;  
4 }
```

pl_sum.c

If you need more help with Pintos Lists abstraction, check out `lib/kernel/list.h` in your Pintos source code. For a deeper dive into the design, check out this [article](https://medium.com/@414apache/kernel-data-structures-linkedlist-b13e4f8de4bf)¹ on Linux's doubly linked lists, which are what Pintos lists are based off of.

¹<https://medium.com/@414apache/kernel-data-structures-linkedlist-b13e4f8de4bf>