Discussion 1: Operating Systems

January 27, 2023

Contents

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

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 at 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 and resources such as code, global data, and files.

Processes are protected from each other due to differing address spaces. If a bug were 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 (e.g. timer, I/O) that trigger a mode switch (from user mode to kernel mode). **Traps**, also known as **exceptions** or software interrupts, are internal synchronous events (e.g. segmentation fault, divide by zero) that trigger a mode switch.

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 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 not just limited to the interrupts mentioned in the previous paragraph. The location of the IVT is stored in a designated processor register.

1.1   Concept Check

1. What is the importance of address translation?

   address translation translate virtual address used by process to physical address to provide isolation to process

   also provides protection between multiple processes from accessing other's memory

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

   processor must save its state and switch

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

   1. limited entry to the kernel
   2. atomic changes to processor state
   3 Transprant. restartable execution

2   Processes

Processes are instances of a running program with their own protected address spaces.

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 will you 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. 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 }
```

<center>fork_simple.c</center>

`exec` changes 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.

```c
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
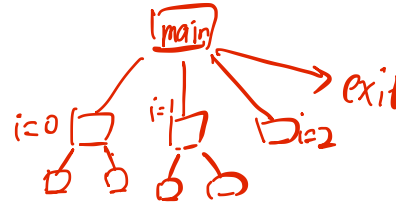
3   7 newly forked processes will continue execute

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

```c
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

the loop wherever the parent process left off

The last digit of pi is 5
The last digit of pi is 5
The address spaces of parent and child are not

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

Shared, change stuff in child has no effect on parent

```c
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 last digit of pi is 5
> The last digit of pi is 5   The heap will not
>   be shared as well

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  }
```

<div align="center">fork_wait.c</div>

> Hello World: 0.
> Hello World: 162162

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 }
```

<div align="center">fork_exec.c</div>

> No. it will only print 0,1,2,3
>
> Create a child process to call exe
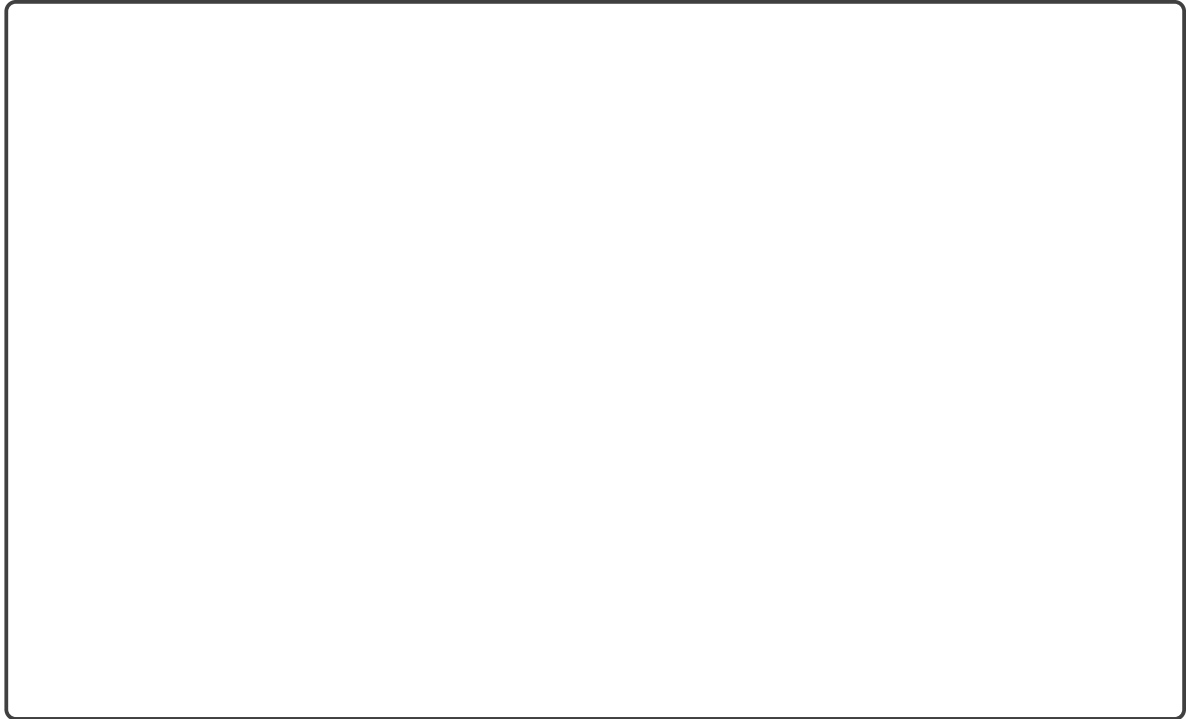
2.2 Signal Handling

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?

> The program can run forever if they are override

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.

```c
void sigquit_handler(int sig) {
  if (sig == SIGINT || sig == SIGQUIT)
    exit(1);
}
void sigint_handler(int sig) {
  if (sig == SIGINT)
    signal(SIGINT, sigquit_handler);
}
int main() {
  signal(SIGQUIT, sigquit_handler);
  signal(SIGINT, sigint_handler);
  while (1) {
    printf("Sleeping for a second ...\n");
    sleep(1);
  }
}
```

exit_signals.c

> Crtl + C   or   Ctrl + \

3   Pintos Lists

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

```c
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;
}
```

<div align="center">ll.c</div>

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

```c
/* 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);

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

  return total;
}
```

<div align="center">pl.c</div>

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[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