RV College of Engineering

Go, change the world

Course Code:22MCE22TL

Advances in Operating System

Unit1

Introduction to Operating Systems

Introduction

- Von Neumann model of computing
  Running a program executes many millions/billions of instructions/second, the processor fetches an instruction from memory, decodes it, and executes it (Ex: add two numbers, access memory, check a condition, jump to a function etc..) and then the processor moves on to the next instruction, and so on, and so on, until the program finally completes.
- While a program runs, a lot of other things are going on with the primary goal of making the system easy to use. The body of software that is responsible for making it easy to run programs (even allowing users to seemingly run many at the same time), allowing programs to share memory, enabling programs to interact with devices etc. is the operating system (OS). It is in charge of making sure the system operates correctly and efficiently in an easy-to-use manner.
- The primary way the OS does this is through a general technique that is called virtualization. That is, the OS takes a physical resource (such as the processor, or memory, or a disk) and transforms it into a more general, powerful, and easy-to-use virtual form of itself.
- In order to allow users to tell the OS what to do and thus make use of the features of the virtual machine (such as running a program, or allocating memory, or accessing a file), the OS also provides some interfaces (APIs).

Introduction

- A typical OS exports a few hundred system calls that are available to applications. Because the OS provides these calls to run programs, access memory and devices, and other related actions, it is referred that the OS provides a standard library to applications.
- Finally, because virtualization allows many programs to run (thus sharing the CPU), and many programs to concurrently access their own instructions and data (thus sharing memory), and many programs to access devices (thus sharing disks and so forth), the OS is sometimes known as a **resource manager**.
- Each of the CPU, memory, and disk is a resource of the system; it is thus the operating system's role to manage those resources, doing so efficiently or fairly or indeed with many other possible goals.

Virtualizing The CPU To understand the role of the OS, consider the example in Figure 2.1 depicting the program that calls Spin(), a function that repeatedly checks the time and returns once it has run for a second. Then, it prints out the string that the user passed in on the command line, and repeats.

If cpu.c is compiled and run on a system with a single processor...

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
^C
prompt>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include "common.h"

int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];
    while (1) {
        Spin(1);
        printf("%s\n", str);
    }
    return 0;
}
```

Figure 2.1: Simple Example: Code That Loops And Prints (cpu.c)

Virtualizing The CPU : Figure 2.2 shows the results of this slightly more complicated example.

Even though there is only one processor, somehow all four of these programs seem to be running at the same time!

OS with some help from the hardware, is in charge of this illusion, i.e., the illusion that the system has a very large number of virtual CPUs. Turning a single CPU (or a small set of them) into a seemingly infinite number of CPUs and thus allowing many programs to seemingly run at once is what is called virtualizing the CPU.

To run programs, stop and to tell the OS which programs to run, there need to be some interfaces (APIs) that can be used to communicate users desires to the OS.

These APIs are the major way in which most users interact with OS.

If 2 programs want to run at a particular time, which should run will be decided by a policy of the OS; policies are used in many different places within an OS. Hence the role of the OS as a resource manager.

```
prompt> ./cpu A & ./cpu B & ./cpu C & ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
...
```

Figure 2.2: Running Many Programs At Once

Note: 4 processes at the same time were ran by using the & symbol which runs a job in the background in the shell, i.e user is able to immediately issue their next command, which in this case is another program to run.

Virtualization
Virtualizing The Memory

Virtualizing Memory: Memory an array of bytes; to read/write to and from memory, an address is specified to be able to access the data.

A program keeps all of its data structures in memory, and accesses them through various instructions, like loads and stores or other explicit instructions that access memory.

Instruction memory and Data Memory are used while running programs

Example program (in Figure 2.3) allocates some memory using malloc(). The o/p of this program is:

```
prompt> ./mem
(2134) address pointed to by p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int
main(int argc, char *argv[])
{
    int *p = malloc(sizeof(int));                    // a1
    assert(p != NULL);
    printf("(%d) address pointed to by p: %p\n",
           getpid(), p);                             // a2
    *p = 0;                                          // a3
    while (1) {
        Spin(1);
        *p = *p + 1;
        printf("(%d) p: %d\n", getpid(), *p);        // a4
    }
    return 0;
}
```

Figure 2.3: A Program That Accesses Memory (mem.c)

Multiple Instances of the memory allocation program:

Each running program has allocated memory at the same address (0x200000), and yet each seems to be updating the value at 0x200000 independently!

It is as if each running program has its own private memory, instead of sharing the same physical memory with other running programs.

That is exactly what is happening as the OS is virtualizing memory. Each process accesses its own private virtual address space (its address space), which the OS maps onto the physical memory of the machine.

A memory reference within one running program does not affect the address space of other processes (or the OS itself); as far as the running program is concerned, it has physical memory all to itself.

The reality, however, is that physical memory is a shared resource, managed by the OS.

Note: For this example to work, make sure address-space randomization is disabled; randomization, can be a good defense against certain kinds of security flaws. Read more about it on your own, especially if you want to learn how to break into computer systems via stack-smashing attacks

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

Figure 2.4: **Running The Memory Program Multiple Times**

Concurrency

Concurrently running many programs leads to issues that has to be addressed, problems of concurrency arose first within the OS itself

Ex: As seen virtualization, OS runs many processes at once, first running one process, then another, and so forth.
Indeed, modern multi-threaded programs exhibit the same problems and is demonstrated with an example of a multi-threaded program (Figure 2.5).

prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value : 2000

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

Figure 2.5: A Multi-threaded Program (threads.c)

Concurrency

When the 2 threads are finished, the final value of the counter is 2000, as each thread incremented the counter 1000 times.

When the input value of loops is set to N, user would expect the final output to be 2N.

But with higher values for loops, Ex:

prompt> ./thread 100000

Initial value : 0

Final value : 143012

prompt> ./thread 100000

Initial value : 0

Final value : 137298

The program gives not only the wrong value, but also a different value than the last time. In fact, if the program is run over and over with high values of loops, user may find that sometimes even get the right answer! So….As it turns out, the reason for these odd and unusual outcomes relate to how instructions are executed, which is one at a time. Unfortunately, a key part of the program above, where the shared counter is incremented, takes three instructions: one to load the value of the counter from memory into a register, one to increment it, and one to store it back into memory. Because these three instructions do not execute atomically (all at once), strange things can happen.

Persistence

Persistence: In system memory, data can be easily lost, as devices such as DRAM store values in a volatile manner; when power goes away or the system crashes, any data in memory is lost. Thus, hardware and software are needed to be able to store data **persistently**; such storage is thus critical to any system as users care a great deal about their data.

The hardware comes in the form of some kind of **input/output** or **I/O** device; a **hard drive** is a common repository for long lived information, and **solid-state drives (SSDs)** are making headway in this arena as well.

The software component of OS managing the disk is called the **file system;** responsible for storing any **files** the user creates in a reliable and efficient manner on the disks of the system.

Unlike the abstractions provided by the OS for the CPU and memory, the OS does not create a private, virtualized disk for each application. Rather, it is assumed that often times, users will want to **share** information that is in files.

For example, when writing a C program user uses first an editor. Once done, the compiler to turn the source code into an executable and the user then runs the new executable (e.g., ./main). Thus, user can see how files are shared across different processes. First, editor creates a file that serves as input to the compiler; the compiler uses that input file to create a new executable file, finally, the new executable is then run. And thus a new program is written!

Persistence

- Figure 2.6 presents code to create a file (/tmp/file) that contains the string “hello world”.
- To accomplish this task, the program makes three calls into the OS. First, a call to open(), opens the file and creates it; the second, write(), writes some data to the file; the third, close(), simply closes the file thus indicating the program won’t be writing any more data to it. These system calls are routed to the part of the operating system called the file system, which then handles the requests and returns some kind of error code to the user.
- Steps an OS takes in order to actually write to disk, first figuring out where on disk this new data will reside, and then keeping track of it in various structures the file system maintains. Doing so requires issuing I/O requests to the underlying storage device, to either read existing structures or update (write) them. Fortunately, the OS provides a standard and simple way to access devices through its system calls. Thus, the OS is sometimes seen as a standard library. Of course, there are many more details in how devices are accessed, and how file systems manage data persistently atop said devices.
- For performance reasons, most file systems first delay such writes for a while, hoping to batch them into larger groups. To handle the problems of system crashes during writes, most file systems incorporate some kind of intricate write protocol, such as journaling or copy-on-write, carefully ordering writes to disk to ensure that if a failure occurs during the write sequence, the system can recover to reasonable state afterwards. To make different common operations efficient, file systems employ many different data structures and access methods, from simple lists to complex btrees.

```c
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    int fd = open("/tmp/file", O_WRONLY|O_CREAT|O_TRUNC,
                  S_IRWXU);
    assert(fd > -1);
    int rc = write(fd, "hello world\n", 13);
    assert(rc == 13);
    close(fd);
    return 0;
}
```

Figure 2.6: A Program That Does I/O (io.c)

Design Goals

- OS takes physical resources, such as a CPU, memory, or disk, and virtualizes them. It handles tough and tricky issues related to concurrency. And it stores files persistently, thus making them safe over the long-term.
- Given that the user want to build such a system, some goals must be kept in mind to help focus design and implementation and make trade-offs as necessary; finding the right set of trade-offs is a key to building systems.
- One of the most basic goals is to build up some abstractions in order to make the system convenient and easy to use. Abstractions are fundamental to everything we do in computer science. Abstraction makes it possible to write a large program by dividing it into small and understandable pieces, to write such a program in a high-level language like C without thinking about assembly, to write code in assembly without thinking about logic gates, and to build a processor out of gates without thinking too much about transistors. Abstraction is so fundamental that sometimes we forget its importance.
- Some of the major abstractions that have developed over time, giving a way to think about pieces of the OS are One goal in designing and implementing an operating system is to provide high performance; another way to say this is our goal is to minimize the overheads of the OS. Virtualization and making the system easy to use are well worth it, but not at any cost; thus, we must strive to provide virtualization and other OS features without excessive overheads. These overheads arise in a number of forms: extra time (more instructions) and extra space (in memory or on disk). Solutions that minimize one or the other or both, if possible are seeked. Perfection, however, is not always attainable

Design Goals

- Another goal will be to provide protection between applications, as well as between the OS and applications. Because we wish to allow many programs to run at the same time, we want to make sure that the malicious or accidental bad behavior of one does not harm others; we certainly don't want an application to be able to harm the OS itself (as that would affect all programs running on the system). Protection is at the heart of one of the main principles underlying an operating system, which is that of isolation; isolating processes from one another is the key to protection and thus underlies much of what an OS must do.

- The operating system must also run non-stop; when it fails, all applications running on the system fail as well. Because of this dependence, operating systems often strive to provide a high degree of reliability. As operating systems grow ever more complex (sometimes containing millions of lines of code), building a reliable operating system is quite a challenge— and indeed, much of the on-going research in the field focuses on this exact problem.

- Other goals make sense: energy-efficiency is important in our increasingly green world; security (an extension of protection, really) against malicious applications is critical, especially in these highly-networked times; mobility is increasingly important as OSes are run on smaller and smaller devices. Depending on how the system is used, the OS will have different goals and thus likely be implemented in at least slightly different ways.

- However, many of the principles that will be presented on how to build an OS are useful on a range of different devices.

Virtualization

The Abstraction: The Process

The Abstraction: The Process: The most fundamental abstractions that the OS provides to users is **the process**.

Process, informally, is quite simple: it is a running program. The program itself just sits there on the disk, a bunch of instructions (and maybe some static data), it is the OS that takes these bytes and gets them running, transforming the program into something useful.

It turns out that one often wants to run more than one program at once; for example, consider your desktop or laptop where user might like to run a web browser, mail program, a game, a music player etc. Typical system may be seemingly running tens or even hundreds of processes at the same time.

HOW TO PROVIDE THE ILLUSION OF MANY CPUS?

Although there are only a few physical CPUs available, how can the OS provide the illusion of a nearly-endless supply of CPUs?

OS creates this illusion by **virtualizing the CPU**. By running one process, then stopping it and running another, and so forth, the OS can promote the illusion that many virtual CPUs exist when in fact there is only one physical CPU (or a few). This basic technique, known as **time sharing of the CPU**, allows users to run as many concurrent processes as they would like; the potential cost is performance, as each will run more slowly if the CPU(s) must be shared.

The Abstraction: The Process

To implement virtualization of the CPU, and to implement it well, the OS will need both some low-level machinery and some high-level intelligence.

Low-level machinery is called mechanisms; as these are low-level methods or protocols that implement a needed piece of functionality.
Ex: A context switch, which gives the OS the ability to stop running one program and start running another on a given CPU; this time-sharing mechanism is employed by all modern OSes.

On top of these mechanisms resides some of the intelligence in the OS, in the form of policies.
Policies are algorithms for making some kind of decision within the OS.
Ex: given a number of possible programs to run on a CPU, which program should the OS run?

A scheduling policy in the OS makes this decision, likely using historical information (e.g., which program has run more over the last minute?), workload knowledge (e.g., what types of programs are run), and performance metrics (e.g., is the system optimizing for interactive performance, or throughput?) to make its decision.

The Abstraction: The Process

The Abstraction: The Process: The abstraction provided by the OS of a running program is called a process. At any instant in time, a process can be summarize by taking an inventory of the different pieces of the system it accesses or affects during the course of its execution.

To understand what constitutes a process, its machine state has to be understood: what a program can read or update when it is running.

At any given time, what parts of the machine are important to the execution of this program?

One obvious component of machine state that comprises a process is its memory. Instructions lie in memory; the data that the running program reads and writes sits in memory as well. Thus the memory that the process can address (called its **address space**), registers, many instructions are part of the process, explicitly read or update registers and thus clearly they are important to the execution of the process.

Some particular **special registers** forms part of this machine state. Ex: **PC or IP** gives instruction of the program that will execute next; a **stack pointer** and associated **frame pointer** are used to manage the stack for function parameters, local variables, and return addresses.

Finally, programs often access persistent storage devices too. Such I/O information might include a list of the files the process currently has open.

Process APIs are available on any modern operating system.

- **Create**: An OS must include some method to create new processes. When you type a command into the shell, or double-click on an application icon, the OS is invoked to create a new process to run the program that is indicated.
- **Destroy**: As there is an interface for process creation, systems also provide an interface to destroy processes forcefully and most of the processes will run and just exit by themselves when complete; when they don't, user may wish to kill them, and thus an interface to halt a runaway process is quite useful.
- **Wait**: Sometimes it is useful to wait for a process to stop running; thus some kind of waiting interface is often provided.
- **Miscellaneous Control**: Other than killing or waiting for a process, there are sometimes other controls that are possible.

Ex: most OS provide some kind of method to suspend a process (stop it from running for a while) and then resume it (continue it running).

- **Status**: There are usually interfaces to get some status information about a process, such as how long it has run for, or what state it is in.

The Abstraction: The Process

Process Creation

Process Creation: How programs are transformed into processes?

How does the OS get a program up and running? How does process creation actually work?

The first thing that the OS must do to run a program is to load its code and any static data (e.g., initialized variables) into memory, into the address space of the process.

Programs initially reside on disk (or, in some modern systems, flash-based SSDs) in some kind of executable format; loading a program and static data into memory requires the OS to read those bytes from disk and place them in memory (Figure 4.1).

In early (or simple) operating systems, the loading process is done eagerly, i.e., all at once before running the program; modern Oses Perform the process lazily, i.e., by loading pieces of code or data only as they are needed during program execution.

Once the code and static data are loaded into memory, there are a few other things the OS needs to do before running the process. Some memory must be allocated for the program's run-time stack (or just stack).

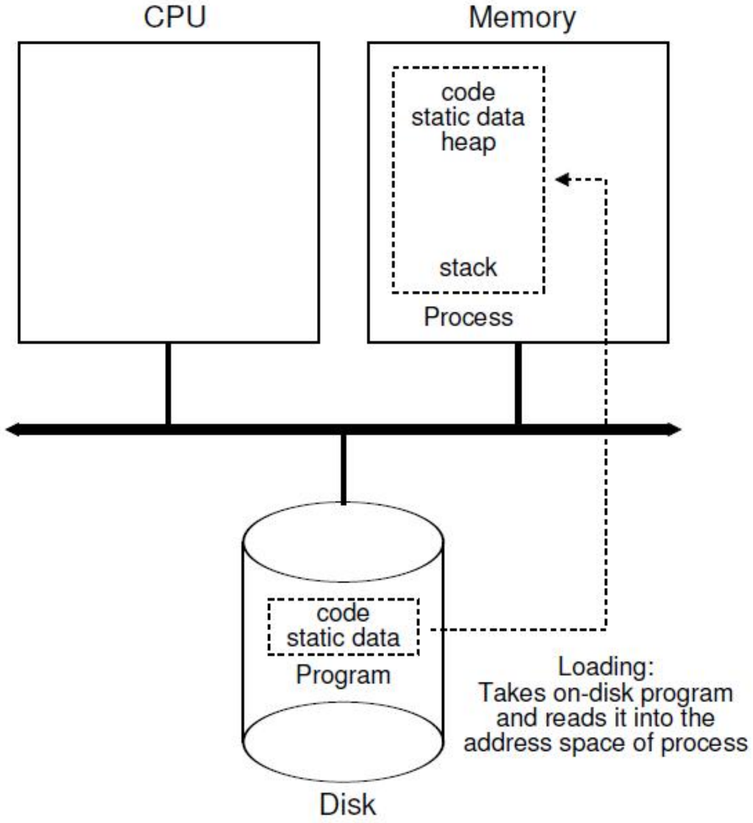

Figure 4.1: Loading: From Program To Process

Process Creation

- As programs use the stack for local variables, function parameters, and return addresses; the OS allocates this memory and gives it to the process, also it initializes the stack with arguments; specifically, it will fill in the parameters to the main() function, i.e., argc and the argv array.
- The OS may also allocate some memory for the program's heap, used for explicitly requested dynamically-allocated data; programs request such space by calling malloc() and free it explicitly by calling free(). The heap is needed for data structures such as linked lists, hash tables, trees etc. The heap will be small at first; as the program runs, and requests more memory via the malloc() OS allocate more memory to the process to help satisfy such calls.
- The OS will also do some initialization tasks related to input/output (I/O). Ex: in UNIX systems, each process by default has three open file descriptors, for standard input, output, and error; these descriptors let programs easily read input from the terminal and print output to the screen.
- By loading the code and static data into memory, by creating and initializing a stack, and by doing other work as related to I/O setup, the OS has now (finally) set the stage for program execution. It thus has one last task: to start the program running at the entry point, namely main(). By jumping to the main() OS transfers control of the CPU to the newly-created process, and thus the program begins its execution.

Process States A process can be in one of three states:

Running: In the running state, a process is running on a processor (executing instructions).

Ready: In the ready state, a process is ready to run but for some reason the OS has chosen not to run it at this given moment.

Blocked: In the blocked state, a process has performed some kind of operation that makes it not ready to run until some other event takes place. Ex: when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.(Figure 4.2)

A process can be moved between the ready and running states at the discretion of the OS. Being moved from ready to running means the process has been scheduled; being moved from running to ready means the process has been descheduled. Once a process has become blocked (e.g., by initiating an I/O operation), the OS will keep it as such until some event occurs (e.g., I/O completion); at that point, the process moves to the ready state again Let's look at an example of how two processes might transition through some of these states. First, imagine two processes running, each of which only use the CPU (they do no

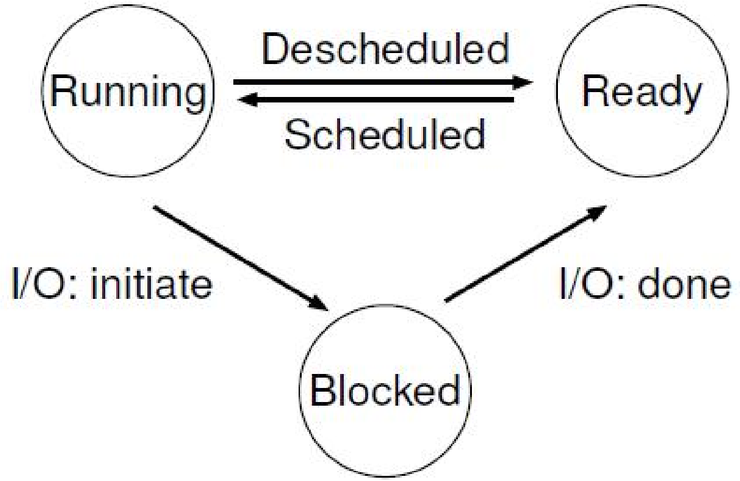

Figure 4.2: **Process: State Transitions**

Process States

Ex1: Two processes transition through some of these states. First, consider 2 processes running, each of which only use the CPU (they do no I/O). In this case, a trace of the state of each process is in Figure 4.3.

Ex2: First process issues an I/O after running for some time. At that point, the process is blocked, giving the other process a chance to run. Figure 4.4 shows a trace of this scenario.

More specifically, Process0 initiates an I/O and becomes blocked waiting for it to complete; processes become blocked, for example, when reading from a disk or waiting for a packet from a network. The OS recognizes Process0 is not using the CPU and starts running Process1. While
Process1 is running, the I/O completes, moving Process0 back to ready. Finally, Process1 finishes, and Process0 runs and then is done.

Many decisions the OS must make, even in this simple example. System had to decide to run P1 while P0 issued an I/O; doing so improves resource utilization by keeping the CPU busy. System decided not to switch back to P0 when its I/O completed; it is not clear if this is a good decision or not. These types of decisions are made by the OS scheduler

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	$Process_0$ now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	$Process_1$ now done

Figure 4.3: **Tracing Process State: CPU Only**

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

Figure 4.4: **Tracing Process State: CPU and I/O**

Data Structures

Data Structures OS is a program, and like any program, it has some key data structures that track various relevant pieces of information. Ex: To track the state of each process, OS keeps process list for all processes that are ready and some additional information to track which process is currently running. The OS must also track, blocked processes; when an I/O event completes, the OS should make sure to wake the correct process and ready it to run again.

Figure 4.5 shows type of information an OS needs to track about each process in the xv6 kernel.

From the figure, important pieces of information the OS tracks about a process. The register context will hold, for a stopped process, the contents of its registers. When a process is stopped, its registers will be saved to this memory location; by restoring these registers (i.e., placing their values back into the actual physical registers), the OS can resume running the process. (Context Switching)

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If !zero, sleeping on chan
  int killed;                 // If !zero, has been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

Figure 4.5: The xv6 Proc Structure

Data Structures

In Proc structure there are some other states a process can be in, beyond running, ready, and blocked. Sometimes a system will have an initial state that the process is in when it is being created. Also, a process could be placed in a final state where it has exited but has not yet been cleaned up (in UNIX-based systems, this is called the zombie state). This final state can be useful as it allows other processes (usually the parent that created the process) to examine the return code of the process and see if the just-finished process executed successfully (usually, programs return zero in UNIX-based systems when they have accomplished a task successfully, and non-zero otherwise).

When finished, the parent will make one final call (e.g., wait()) to wait for the completion of the child, and to also indicate to the OS that it can clean up any relevant data structures that referred to the now-extinct process.

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If !zero, sleeping on chan
  int killed;                 // If !zero, has been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

Figure 4.5: The xv6 Proc Structure

The fork() system call

The fork() system call is used to create a new process

Figure 5.1 shows an example program for invoking fork()

the process that is created is an (almost) exact copy of the calling process. That means that to the OS, it now looks like there are two copies of the program p1 running, and both are about to return from the fork() system call.

The newly-created process (called the child) doesn't start running at main() rather, it just comes into life as if it had called fork() itself

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
  } else {
    // parent goes down this path (main)
    printf("hello, I am parent of %d (pid:%d)\n",
           rc, (int) getpid());
  }
  return 0;
}
```

Figure 5.1: Calling **fork()** (**p1.c**)

Fork() and wait()

The child isn't an exact copy. Specifically, it will have its own copy of the address space (i.e., its own private memory), its own registers, its own PC, and so forth, the value it returns to the caller of fork() is different.

Specifically, while the parent receives the PID of the newly-created child, the child receives a return code of zero.
This differentiation is useful, because it is simple then to write the code that handles the two different cases.

The CPU scheduler, determines which process runs at a given time

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {          // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
  } else {               // parent goes down this path (main)
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
           rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

Figure 5.2: Calling `fork()` And `wait()` (p2.c)

Wait()

Sometimes, it is quite useful for a parent to wait for a child process to finish execution. This task is accomplished with the wait() system call (or waitpid()).

In this example (p2.c), the parent process calls wait() to delay its execution until the child finishes executing. When the child is done, wait() returns to the parent.

Adding a wait() call to the code above makes the output deterministic.

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (rc_wait:29267) (pid:29266)
prompt>
```

With this code, the child will always print first. It might simply run first, as before, and thus print before the parent. However, if the parent does happen to run first, it will immediately call wait(); this system call won't return until the child has run and exited. Thus, even when the parent runs first, it waits for the child to finish running, then wait() returns, and then the parent prints its message.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {          // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
  } else {               // parent goes down this path (main)
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
           rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

Figure 5.2: Calling `fork()` And `wait()` (`p2.c`)

exec() system call

This system call is useful when the user want to run a program that is different from the calling program. Calling fork() is only useful if the user want to keep running copies of the same program. However, often one want to run a different program; exec() does just that.

In p3.c, the child process calls execvp() in order to run the word counting program on the source file p3.c, thus giving how many lines, words, and bytes are found in the file:

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
29 107 1030 p3.c
hello, I am parent of 29384 (rc_wait:29384) (pid:29383)
prompt>
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {          // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: "wc" (word count)
    myargs[1] = strdup("p3.c"); // argument: file to count
    myargs[2] = NULL;           // marks end of array
    execvp(myargs[0], myargs);  // runs word count
    printf("this shouldn't print out");
  } else {               // parent goes down this path (main)
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
           rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

Figure 5.3: Calling **fork()**, **wait()**, And **exec()** (p3.c)

Exec()

Exec() system call given the name of an executable (e.g., wc), & some arguments (e.g., p3.c), it loads code (and static data) from that executable and overwrites its current code segment (and current static data) with it.

Heap and stack and other parts of the memory space of the program are re-initialized.

Then the OS simply runs that program, passing in any arguments as the argv of that process.

Thus, it does not create a new process; rather, it transforms the currently running program (formerly p3) into a different running program (wc).

After the exec() in the child, it is almost as if p3.c never ran; a successful call to exec() never returns.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {          // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: "wc" (word count)
    myargs[1] = strdup("p3.c"); // argument: file to count
    myargs[2] = NULL;           // marks end of array
    execvp(myargs[0], myargs);  // runs word count
    printf("this shouldn't print out");
  } else {               // parent goes down this path (main)
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
           rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

Figure 5.3: Calling **fork()**, **wait()**, And **exec()** (p3.c)