So far, we have discussed threads. All the threads have their private stack and registers and a scheduler schedules these threads after some time interval. The memory is shared among all the threads, and they can see each other data. Usually, the applications we run are untrusted third-party software. For example, it is not desirable that a gaming software installed on your laptop to see data from your banking software.

For this reason, we have another abstraction called process. A process is a container of threads, with at least one thread. Initially, a process has only one thread, but it can create more threads if needed. Every process has a reserved quota in the RAM (also called the address space of a process), so one process can't see other process data. We will discuss in detail how memory isolation is done in future lectures.

Draw figure of address spaces.

In today's class, we will discuss another important aspect of OS design called system calls.

Software's have bugs, and they can be malicious. By malicious we mean that they deliberately try to steal our data, passwords, etc. Even if their intentions are pure, their implementation might have bugs that might leak our data. To prevent this scenario, we divide the applications into two parts. One is the programmer written code (also called user program), and second is the OS code (which you can also think of as a shared library). Like any other library, the operating system also exports some routines to the programmer that the programmer can call to use operating system services. But, there is a difference between a standard library and the OS library. The OS enforce untrusted applications to only enter through these exported routines. The user programs cannot have their own implementation of the library (because an OS manage critical resources like memory, disk, network, etc. that is shared among all the libraries).

To understand this, consider the following example: Let us say the OS exports a routine create_thread for creating a new thread. The definition of create_thread is the following:

```
create_thread () {
   struct thread *t = malloc (sizeof(struct thread));
   t->esp = malloc (4096) + 4096;
   status = interrupt_disable ();
   add_to_ready_list (t);
   set_interrupt_status (status);
}
```

Suppose an untrusted routine somehow manage to call add_to_ready_list directly, the OS will fall into concurrency issue and crash. To prevent this scenario, the OS enforce applications to only enter through create_thread. But, so far, our discussions of x86 architecture, it seems impossible. An application can jump to any address (including add_to_ready_list). To prevent this, the OS loads user programs and OS into different address spaces.

Draw figure and explain.

Because OS and user-programs live in different address spaces the OS can not directly call the OS routines; instead, this is done via system calls.

Before, we go to system call let us discuss the interrupt handling mechanism again. As we already know that interrupt handlers are device specific code (timer, disk, network, etc.). So, the interrupt handlers are the part of the OS. So, let us see what happens when an interrupt occurs during the execution of a user program.

The hardware automatically, switches to kernel address space on an interrupt during the execution of a user program. Because, they are in different address space, the kernel uses a different stack corresponding to every user thread.

Why is a different stack needed (explain)?

On an interrupt in user space:

The hardware:

switch to kernel (OS) stack
push user stack
push user eflags
push user eip

On iret the hardware:
restore user eip
restore user eflags
restore user stack

In addition to hardware interrupts, x86 also facilitates software interrupts. In x86, an int $100 instruction generates an interrupt with vector number 100. Here, you can replace 100 with any other integer between (0-255), to trigger an interrupt of the given type. To implement system calls the OS kernel reserves one of these vector numbers for a system_call handler. For example, Linux uses 128 for the handling of the system call. The kernel puts the address of system_call handler at index 128 in IDT. The kernel assigns an integer id to each system call. These ids are available to user program, before generating the software interrupt for system call the user program can pass the system call vector id to the kernel in some registers or stack. The OS kernel and user programs mutually decide the argument passing convention during a system call.

In this lecture we will discuss some system calls abstraction provided by the Unix OS.

In Unix, shell is the first user process. Shell takes command line arguments from users and spawn new processes.

Pseudo code:

```
while (1) {
    write (1, "$ ", 2);
    readcommand (0, command, args);
    if ((pid = fork ()) == 0) {
       exec (command, args, 0);
    } else if (pid > 0) {
       wait (0);
    } else
      printf ("Failed to fork\n");
}
```

system calls : read, write, fork, exec, wait.
What is the shell doing? fork, exec, wait: process diagram (PID, address space)
Why call "wait"? to wait for the child to terminate and collect its exit status. (if child finishes, child becomes a zombie until parent calls wait.)

Example:

```
$ ls
```

how does ls know which directory to look at? (cwd variable is copied during fork)
how does it know what to do with its output? (fd 1 is initialized by the shell)
I/O: process has file descriptors, numbered starting from 0.
system calls: open, read, write, close
numbering conventions:

- file descriptor 0 for input (e.g, keyboard). read_command
  `read (0, buf, bufsize)`
- file descriptor 1 for output (e.g., terminal)
  `write (1, "hello\n", strlen ("hello\n"))`
- file descriptor 2 for error (e.g., terminal)

On fork, child inherits open file descriptors from parent
On exec, process retains file descriptors, except for those specifically marked as close-on-exec: fnctl (fd, F_SETFD, FD_CLOEXEC)

How does shell implement:

```
ls > tmp1
```

just before exec insert:

```
close (1)
creat ("tmp1", 0666); //fd will be 1
```

The kernel always uses the first free file descriptor, 1 in this case. Could also use dup2() to clone a file descriptor to a new number.

What if you run the shell itself with redirection?
`$ sh < script > tmp1`

If for example the file script contains

echo one
echo two
FD inheritance makes this works well.

What if we want to redirect multiple FDs (stdout, stderr) for programs that print both?
$ ls f1 f2 nonexistent-f3 > tmp1 2>&1

after creat, insert:
close (2)
creat ("tmp1", 0666); // fd will be 2
why is this bad? illustrate what's going on with file descriptors. better:
close (2);
dup (1);

how to run a series of program on some data?
sort < file.txt > tmp1
uniq < tmp1 > tmp2
wc < tmp2
rm tmp1 tmp2

can be more concisely done as:
sort < file.txt | uniq | wc

discuss advantages.

A pipe is a one-way communication channel. Here is a simple example:

```
int fd[2];
char buf[512];
int n;

pipe (fd);
write (fd[1], "hello", 5);
n = read (fd[0], buf, sizeof(buf));
// buf[] now contains 'h', 'e', 'l', 'l', 'o'
```

file descriptors are inherited across fork (), so this also works:

```
int fd[2];
char buf[512];
int n, pid;

pipe (fd);
pid = fork ();
if (pid > 0) {
   write (fd[1], "hello", 5);
} else {
   n = read(fd[0], buf, sizeof (buf));
```

}

How does shell implement pipelines (i.e., cmd 1 | cmd 2 | …)? We want to arrange that the output of cmd 1 is the input of cmd 2.

The shell creates processes for each command in the pipeline, hooks up their stdin and stdout, and waits for last process of the pipeline to exit.

```
int fd[2];
if (pipe (fd) < 0) {
   printf (“error\n”);
   exit (-1);
}
if ((pid == fork()) == 0) {
   close (1);
   tmp = dup (fd[1]);
   close (fd[0]);
   close (fd[1]);
   exec (command1, args1, 0);
} else if (pid > 0) {
   close (0);
   tmp = dup (fd[0]);
   close (fd[0]);
   close (fd[1]);
   exec (command2, args2, 0);
} else {
   printf (“Unable to fork\n”);
   exit (-1);
}
```