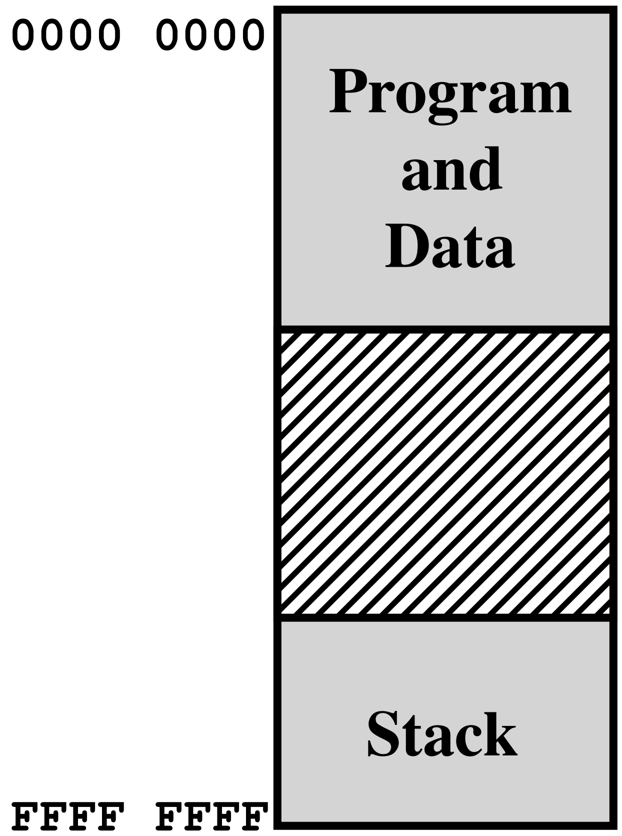
Overview of the Kernel and Shell

Outline:

- User Processes
- Kernel
- Virtual Address Space
- User Mode / System Mode
- Syscall Interface
- A Simple Shell

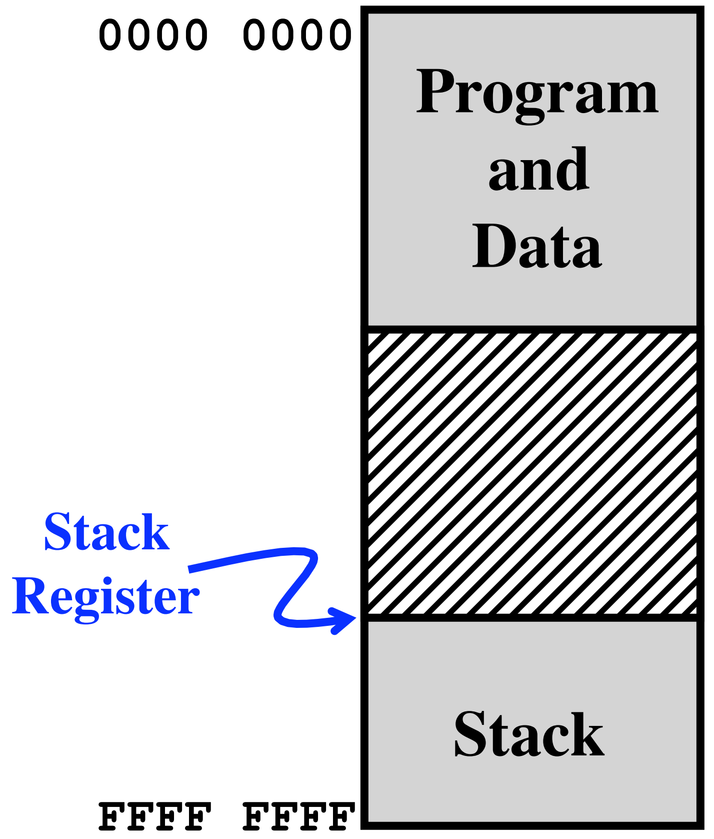
User Processes

“Virtual Address Space”
“Logical Address Space”
“Core Image”



User Processes

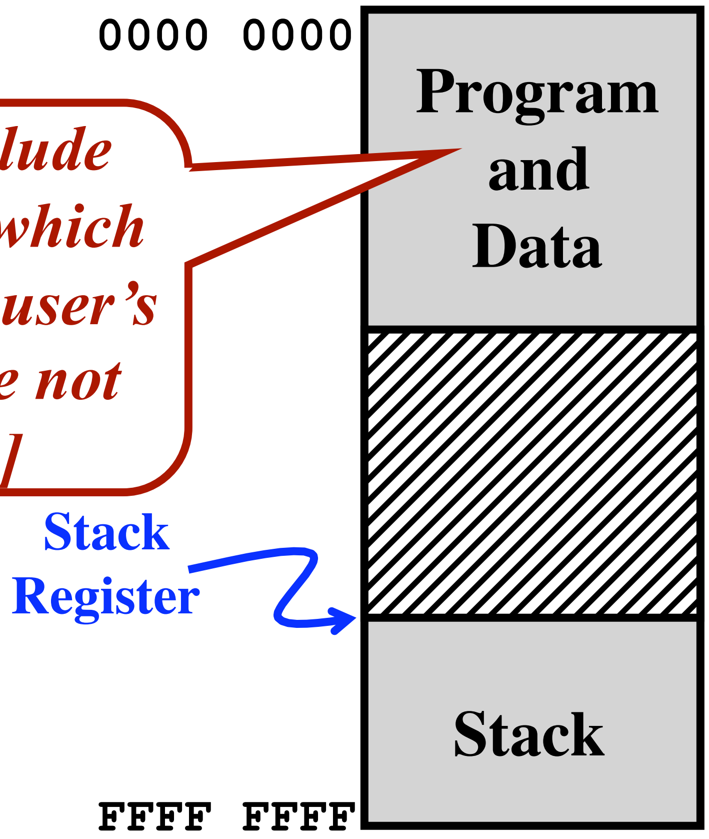
"Virtual Address Space"
"Logical Address Space"
"Core Image"



User Processes

"Virtual Address Space"
"Logical Address Space"
"Core Image"

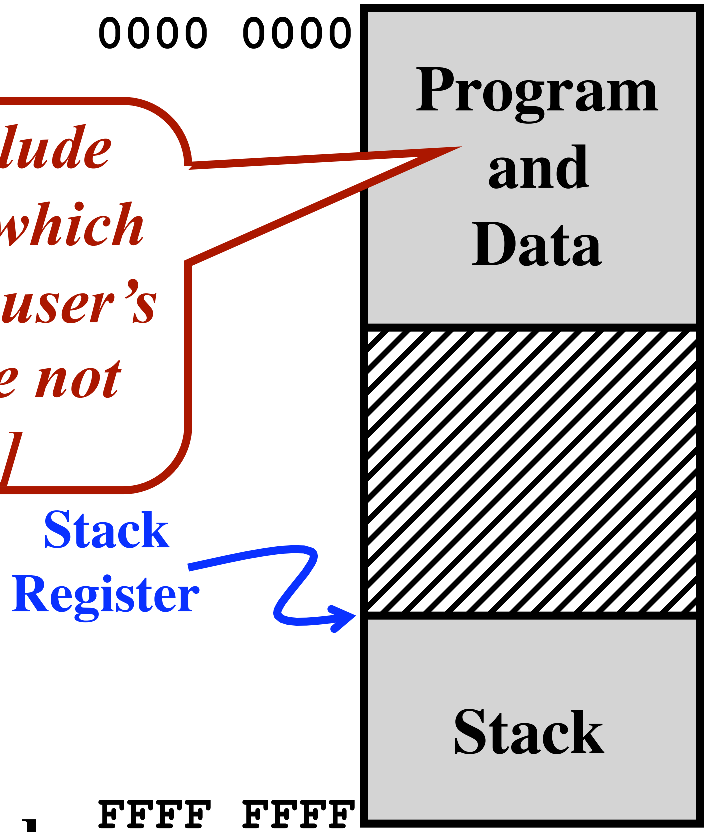
Will usually include library routines, which are "linked" into user's code. [These are not kernel code!]



User Processes

"Virtual Address Space"
"Logical Address Space"
"Core Image"

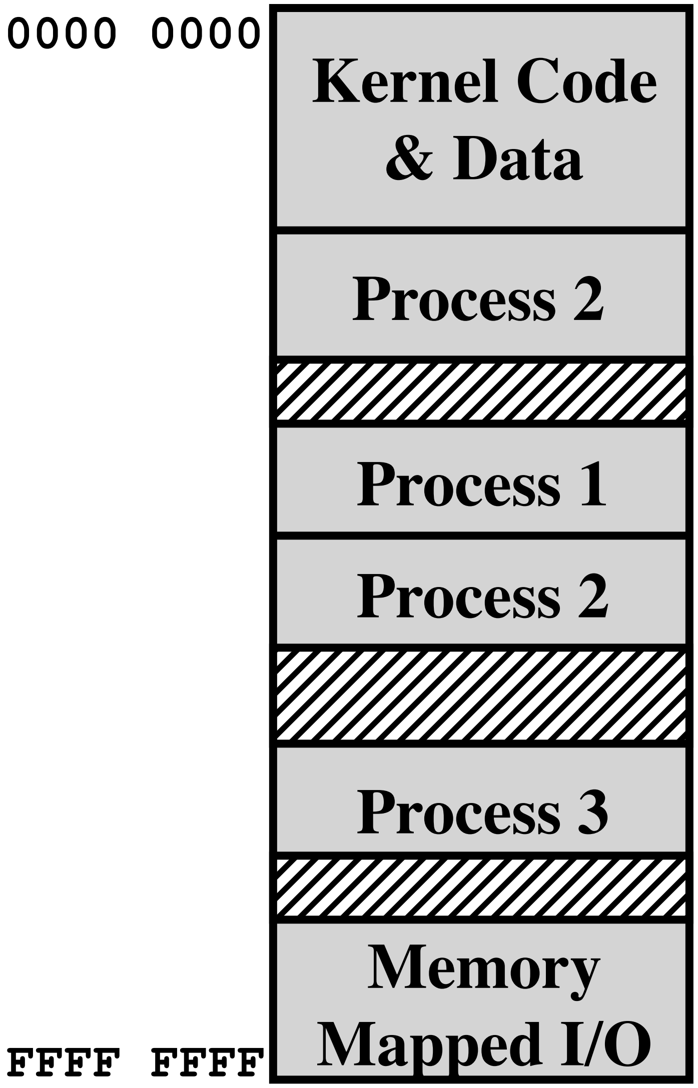
Will usually include library routines, which are "linked" into user's code. [These are not kernel code!]



Has its own set of registers.
Cannot even see the memory of
  kernel or other processes.
Runs in "*user mode*".
  Some instructions are disallowed.
  (I/O, page table, etc.)
Has a single "*thread of execution*".
  Has its own PC ("program counter")

The Kernel

“*Physical Address Space*”
(actual installed main memory)

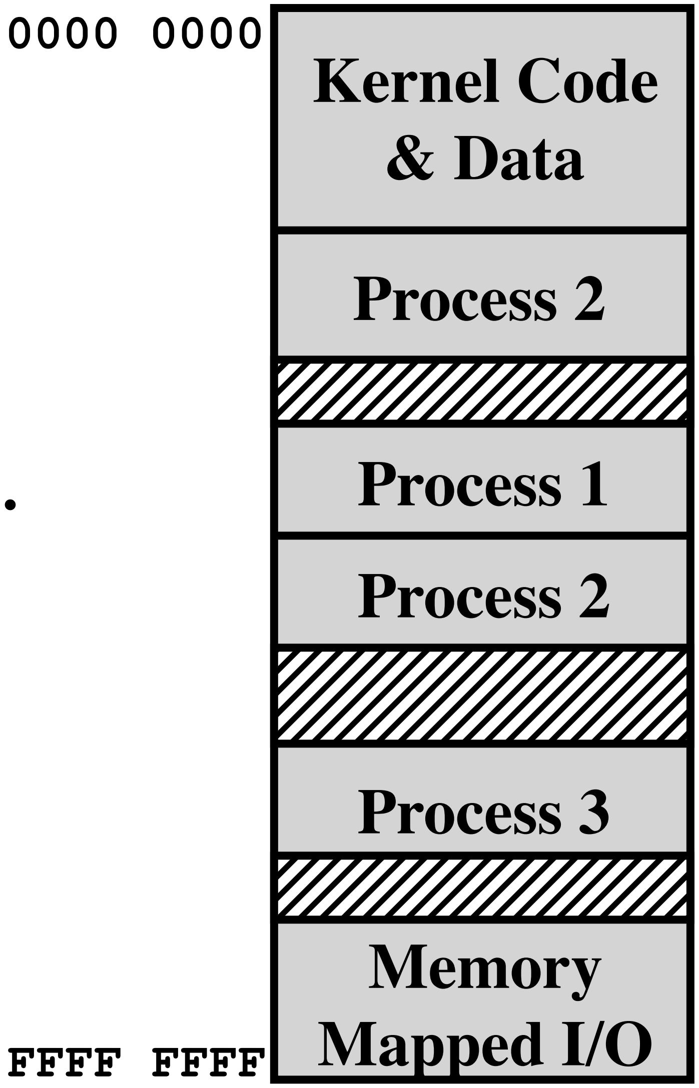


The Kernel

"*Physical Address Space*"
(actual installed main memory)

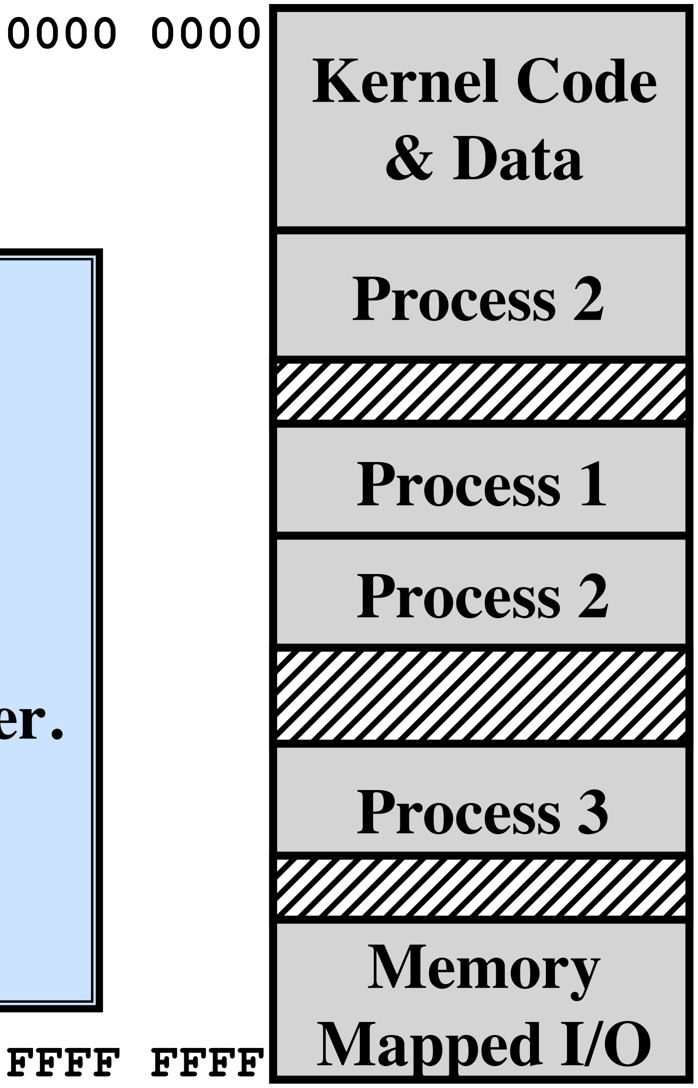
Process 2's Address Space
... is all over physical memory,
... and partly on disk,
... and shared with other processes.

We'll discuss:
"Virtual Memory Management"
"PAGE TABLE hardware"



The Kernel

"*Physical Address Space*"
(actual installed main memory)



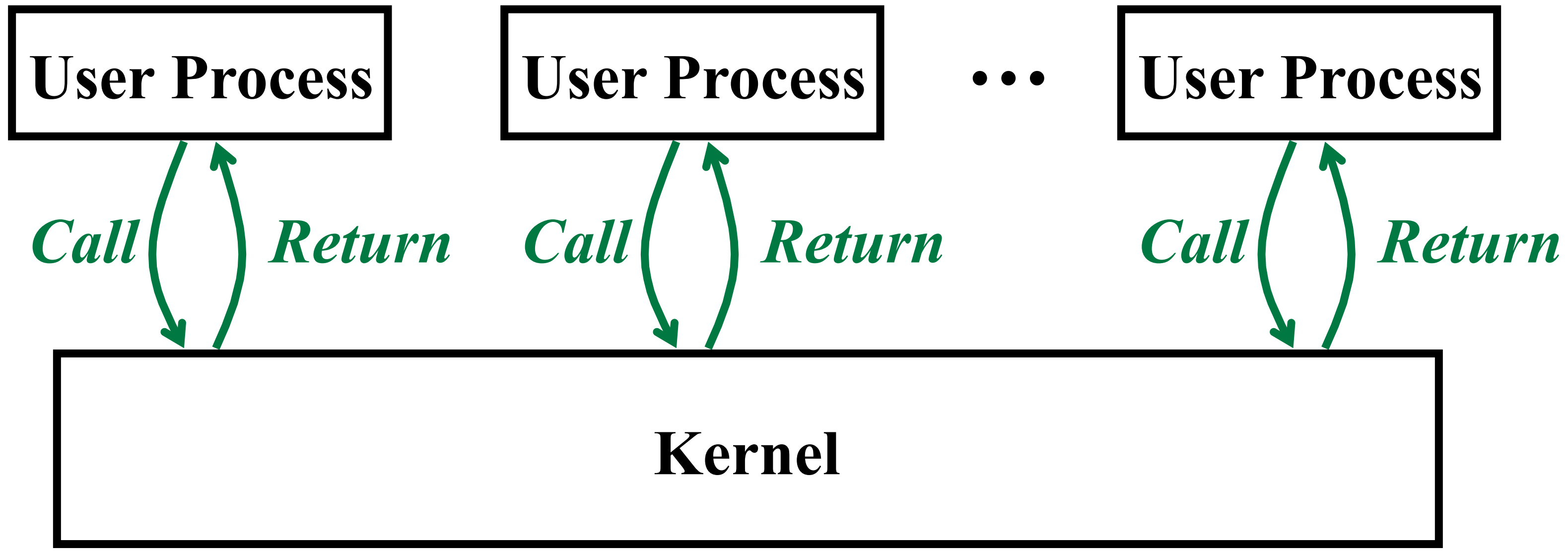
Kernel runs in "*System Mode*".

Must manage user processes.
  Page Tables --> Virtual Memory

Switches from one process to another.
  Time-slicing / multitasking

Must manage I/O devices.

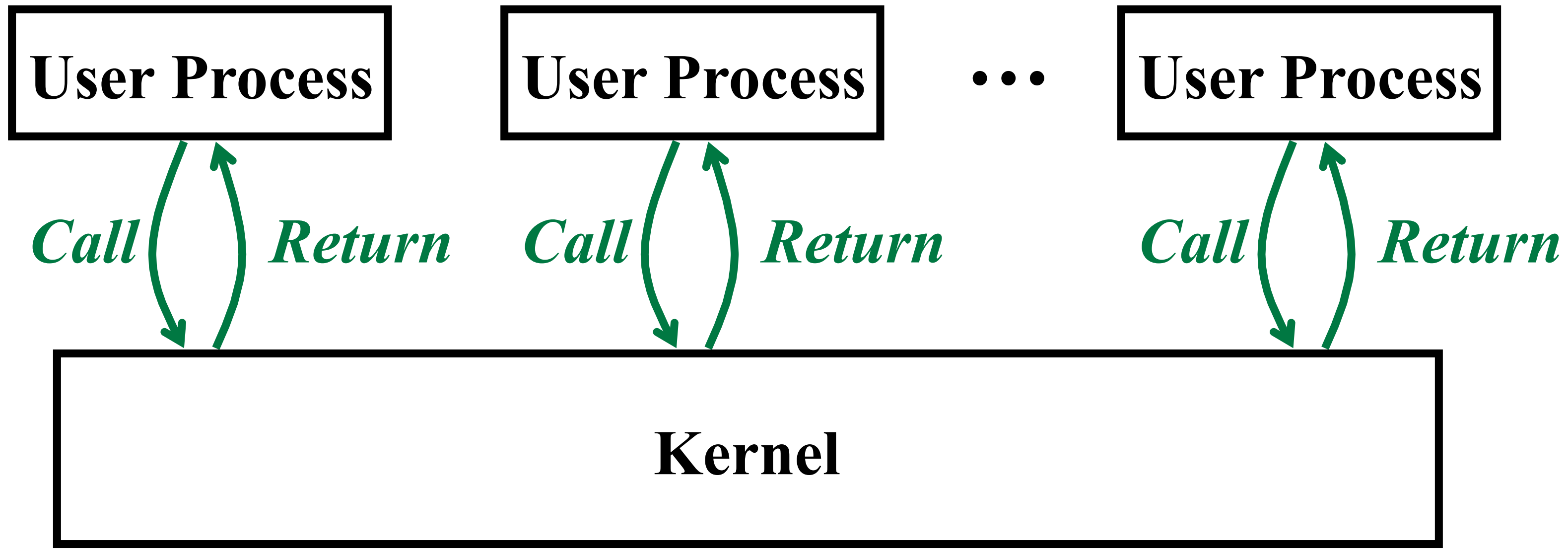
System Calls



User Process

- Runs in its own separate address space
- Runs in "User Mode"
- Can't use a normal CALL instruction

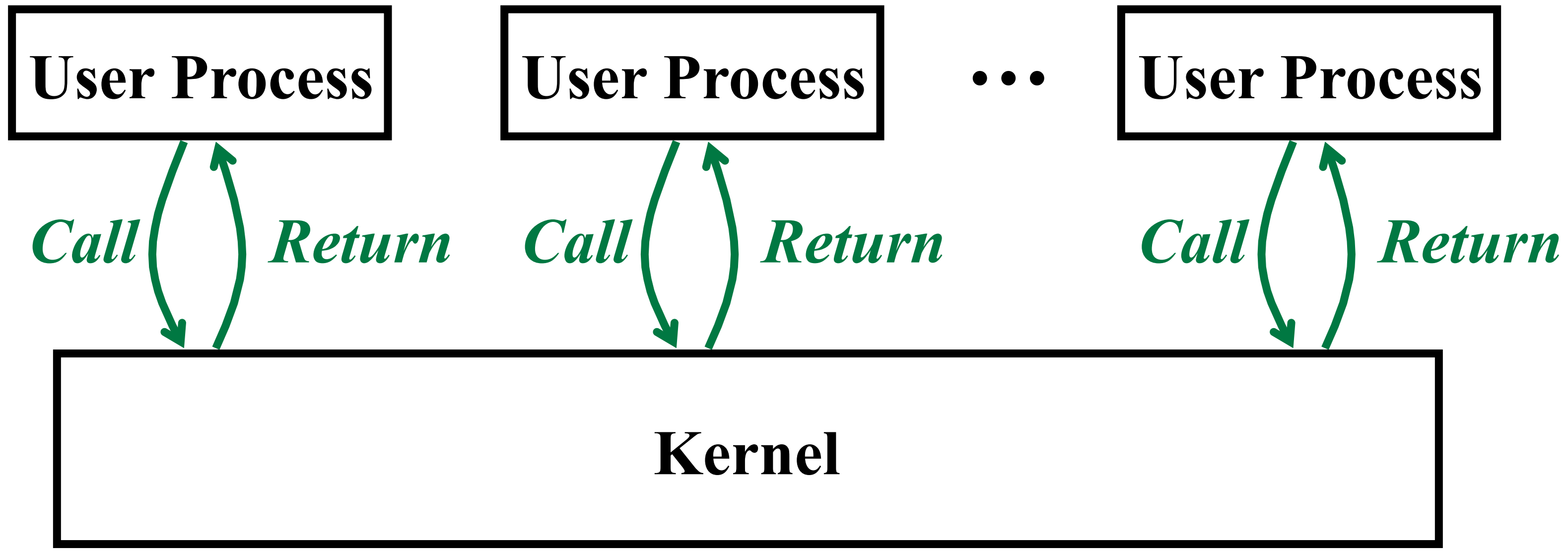
System Calls



The* SYSCALL *Instruction

- **Invoked by user code**
- **Switches into "System Mode"**
- **Transfers control to a kernel routine**
- **Args may be passed to kernel (including a function code)**

System Calls



The Return-From-Interrupt (RETI) Instruction

- Invoked by kernel code
- Switches back to “User Mode”
- Transfers control back to just after the SYSCALL

The “POSIX” Standard Interface

A set of SYSCALL functions
Implemented in all UNIX/LINUX kernels

File Management

```
  fd = open (filename, how, ...)
  x = close (fd)
  n = read (fd, buffer, numBytes)
  n = write (fd, buffer, numBytes)
  position = lseek (fd, offset, whence)
  s = stat (filename, bufferAddr)
```

The "POSIX" Standard Interface

A set of SYSCALL functions
Implemented in all UNIX/LINUX kernels

File Management

```
fd = open (filename, how, ...)
x = close (fd)
n = read (fd, buffer, numBytes)
n = write (fd, buffer, numBytes)
position = lseek (fd, offset, whence)
s = stat (filename, bufferAddr)
```

For each, there is a "stub routine".

- **Included from library**
- **Coded in assembly**
- **Move args into registers**
- **Execute a SYSCALL**
- **After return from kernel, return**

The “POSIX” Standard Interface

Directory Management

```
s = mkdir (name, mode)
s = rmdir (name)
s = link (name1, name2)
s = unlink (name)
s = mount (special, name, flag)
s = unmount (special)
```

The “POSIX” Standard Interface

Misc Syscalls

```
s = chdir (directoryName)
s = chmod (fileName, newModeBits)
s = kill (pid, signalType)
```

Send a “Signal” to a user process.

Somewhat like an “interrupt”.

```
seconds = time (&seconds)
```

Get the current date and time.

The “POSIX” Standard Interface

Misc Syscalls

```
pid = fork ()

s = execve (filename, argv, environp)

exit (status)

pid = waitpid (pid, &statloc, options)
```

The “POSIX” Standard Interface

Misc Syscalls

```
pid = fork ()
s = execve   filename, argv, environp)
exit (status
pid = waitpid         &statloc, options)
```

- Used to create a new process.
- Executed by the “*parent*” process.
- Create a new process, called the “*child*”.
- Make a new copy of parent’s address space.
- In parent, return “*process id*” of the child.
- In child, return 0.

The “POSIX” Standard Interface

Misc Syscalls

```
pid = fork ()
s = execve (filename, argv, environp)
exit (st  tus)
pid = w    oid (pid, &statloc, options)
```

- Read from a new program in from a file.
- Replace this process’s memory image.
- Begin executing the new program.
- Never returns, except when errors.

The “POSIX” Standard Interface

Misc Syscalls

```
pid = fork ()

s = execve (filename, argv, environp)

exit (status)

pid = waitpid (pid, &statloc, options)
```

- Terminate this process.
- Pass “***exit status***” (an integer) to the parent process.
- No return from this syscall!

The “POSIX” Standard Interface

Misc Syscalls

```
pid = fork ()
s = execve (filename, argv, environp)
exit (status)
pid = waitpid (pid, &statloc, options)
```

- Wait for a child to exit.
  (Option: wait for a specific child or any child.)
- Save the child’s “*exit status*” in statloc.
- If the child terminated earlier,
  then return its exit status immediately.

A UNIX Shell Program

```
while (TRUE) {
  type_prompt ();
  read_command (command, parameters);
  if (fork () != 0) {
      waitpid (-1, &status, 0);
  } else {
      execve (command, parameters, 0);
  }
}
```

A UNIX Shell Program

```
while (TRUE) {
  type_prompt ();
  read_command (command, parameters);
  if (fork () != 0) {
      waitpid (-1, &status, 0);
  } else {
      execve (command, parameters, 0);
  }
}
```

Parent's Code

Child's Code

Wait for any child