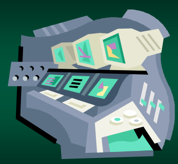
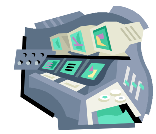


Operating Systems

Part 8

Operating Systems

- Master controller for all of the activities that take place within a computer
- Basic Duties:
  - manage the physical resources of the system
  - load and execute programs
  - controlling I/O devices



What *is* an operating system?

- The operating system is simply another program executing on the processor
- However it…
  - knows about all the hardware in the computer
  - runs in *privileged (or supervisor) mode*
  - which gives it the ability to run special instructions
  - other programs run in *user mode*

Multiprogrammed operating system

- Most computers support *multiprogramming* (aka *multitasking*)
- Presents the illusion that multiple programs are running simultaneously on a computer
  - each program executes for a fixed amount of time, known as a *timeslice*
  - user programs do not know if other programs are running on the system

Context Switch

- When a program's time slice ends…
  1. operating system stops it
  2. copies its register file into memory
  3. removes it from the processor
  4. copies the next program's register file out of memory and into the register file
  5. loads next program into the processor
- This process known as a *context switch*

Multiuser computers

- *Multiuser* systems allow more than one user to be logged in to the computer at one time
- Operating system must…
  - protect programs from accessing other program's data
  - prevent users from accessing other user's private data

Interact with Applications

- Necessary for multi-programmed systems
- *Application Program Interface*
  - application can tell the OS to perform a task
  - makes applications faster and smaller
  - also makes the system more secure since apps do not directly talk to IO
  - Application → Operating System → IO

Vector Tables

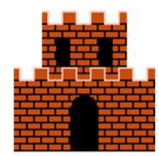
How software and hardware "talk"

Vector Tables

- Often an application (or a piece of hardware) needs to talk to the operating system
- Examples:
  - software needs the OS to output data
  - USB port notifies the OS that a device was plugged in

Vector Tables

- But how does this happen?
- The processor can be *interrupted* - alerted that something must be handled
- Each type interrupt has a unique number – which identifies the type of alert



Vector Table

- When interrupted, the processor looks up the number in the *"vector table"*
- Table contains the address of the subroutine to execute
- The interrupt number is an index into this table

How it Works

- When interrupted, the processor uses the interrupt number (index into the table) and looks up the address
- It then executes that address *(like a function you call in your Java programs)*

How it Works

- These subroutines belong to the *kernal* – the core of the operating system
- So, software can interrupt itself with a specific number *(designated for software to use)* when it needs to talk to the operating system

Instruction: Interrupt

- Interrupt Instruction allows your program to "interrupt" itself and pass information to the operating system *kernal*
- How you use it
  1. fill registers with values that will tell Linux what do to
  2. call Linux by using interrupt 0x80 (or a special *software interrupt* instruction)

Instruction: Interrupt (32-bit)

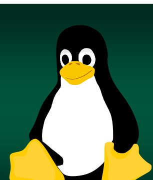
0x80

```
INT number
```


Instruction: syscall (64-bit)

```
SYSCALL
```

Calls vector reserved for OS

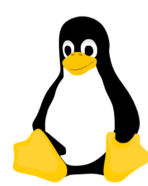


Linux System Calls

How software and hardware "talk"

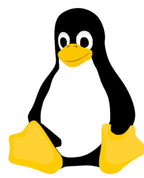
Interrupts on the Linux

- Linux, like other operating systems communicate with applications using *interrupts*
- Applications do not know where (in memory) to contact the kernal – so they ask the processor to do it



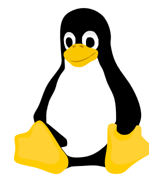
How It Works

1. Fill the registers
2. Interrupt using 0x80 (or the special *software interrupt* instruction)
3. Any results will be stored in the registers



Kernals are Simple!

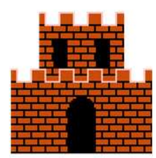
- Linux only has **1** write and **1** read system call
- The location, number of bytes, and device only change
- It basically states *"write x many bytes from y to device z"*
- So, writing to the screen, a file, a port, etc…use the same call!



How to Call Linux – 32 bit

- In the 32-bit version of Linux, the kernal uses eax to identify the system call
- Each call has a unique constant
- Interrupt 0x80 activates the kernal (well, the code that handles app requests)

How to Call Linux – 32 bit



- Often a system call needs additional information to work
- Registers ebx, ecx, edx, etc… will contain this information

Linux 32 – Sys Write

```
mov   $4, %eax
mov   $1, %ebx
mov   $address, %ecx
mov   $length, %edx
int   $0x80
```

Linux 32 call for WRITE

1 = Screen

Call Linux

Linux 32: Sys Read

```
mov   $3, %eax
mov   $0, %ebx
mov   $address, %ecx
mov   $maxBytes, %edx
int   $0x80
```

Linux 32 call for READ

0 = Keyboard

Maximum number of bytes to read

Call Linux

How to Call Linux – 64 bit

- The 64-bit version of Linux changed the system calls
- The rax register still holds the system call number
- However, the "kernal call" numbers are different now
- There are 329 total calls

How to Call Linux – 64 bit

- Different registers are used to hold data
- The order is also quite different: *rdi, rsi, rdx, r10, r8*
- The new instruction syscall talks to the OS

Some Linux 64 Calls

System Call	rax	rdi	rsi	rdx
read	0	fd (device)	address	max bytes
write	1	fd (device)	address	count
open	2	address	flags	mode
close	3	fd (device)		
get pid	39			
exit	60	error code		

Linux 64: Sys Write

```
mov  $1, %rax
mov  $1, %rdi
mov  $address, %rsi
mov  $length, %rdx
syscall
```

Linux command for WRITE

1 = Screen

Call Linux

Linux 64: Sys Read

```
mov  $0, %rax
mov  $0, %rdi
mov  $address, %rsi
mov  $maxBytes, %rdx
syscall
```

Linux command for READ

0 = Keyboard

Maximum number of bytes to read

Call Linux