Kernels & Processes
The Structure of the Operating System

Noah Mendelsohn
Tufts University
Email: noah@cs.tufts.edu
Web: http://www.cs.tufts.edu/~noah
Based on a presentation by Professor Alva Couch
Today

- **The Big Picture:**
  - How the OS is organized
  - How good OS services are designed
- *Process: the most important OS abstraction*
- How the OS interacts with processes
- How processes use memory

Next lecture: how processes are born, how they die, and how to control them while they’re running
Today

- **The Big Picture:**
  - How the OS is organized
  - How good OS services are designed

- *Process: the most important OS abstraction*
- How the OS interacts with processes
- How processes use memory
Why do we need an OS anyway? (brief review)
Operating systems do two things for us:

- They make the computer easier to use
- The facilitate sharing of the computer by multiple programs and users

Interestingly, some of the same abstractions that make things more convenient for one program are just what we need for sharing too.
…actually, Unix & Linux have one more goal:

- To facilitate running the same program (and OS!) on different types of computer

...and guess what, those same abstractions are designed to help with that too!
Why have an operating system?
Why have an operating system?

This computer is doing lots of things at once. We need to:

- Give each program a **useful** environment
- Give each program an **isolated** environment
- Protect the programs from each other
- Let some programs (browser) control others (flash plugin)
- Let the programs access & share system resources
Sharing the Computer
Sharing Memory

Multiple Programs Running at once

All programs share memory

OPERATING SYSTEM

MAIN MEMORY

CPU

Angry Birds

Play Video

Browser
Sharing the CPU

Multiple Programs Running at once

CPU is shared...can only do one thing at a time*

*Actually, modern CPUs can do a few things at a time, but for now assume a simple, traditional computer
Sharing Memory

Multiple Programs Running at once

OPERATING SYSTEM

MAIN MEMORY

CPU

Angry Birds

Play Video

Browser
Protecting the OS Kernel
The operating system is a special, privileged program, with its own code and data. We call the protected, shared part of the OS the "kernel".
We need **help from the hardware** to protect the kernel!

The hardware has memory mapping features that the OS can use to:

- Hide the kernel from other programs
- Hide programs from each other
- Convince each program it’s got its own private memory starting at address zero
We need **help from the hardware** to protect the kernel!

The hardware has memory mapping features that the OS can use to:

- Hide the kernel from other programs
- Hide programs from each other
- Convince each program it’s got its own private memory starting at address zero

Q. What do you mean by Mapping Features?

A. The kernel can tell the hardware to renumber blocks of memory to rearrange it and to hide blocks the program shouldn’t see.
We need **help from the hardware** to protect the kernel!

The hardware has memory mapping features that the OS can use to:

- Hide the kernel from other programs
- Hide programs from each other
- Convince each program it's got its own private memory

**Q. What do you mean by Mapping Features?**

**A.** The kernel can tell the hardware to renumber blocks of memory to rearrange it and to hide blocks the program shouldn’t see.

Each program gets its own *virtual memory.*
We need **help from the hardware** to protect the kernel!

The hardware has memory mapping features that the OS can use to:

- Hide the kernel from other programs
- Hide programs from each other
- Convince each program it's got its own private memory

Q. What do you mean by Mapping Features?

A. The kernel can tell the hardware to renumber blocks of memory to rearrange it and to hide blocks the program shouldn’t see.

Each program gets its own **virtual memory**.

Usually: the kernel memory is not mapped into the program’s space.
We need help from the hardware to protect the kernel!

The hardware has memory mapping features that the OS can use to:

- Hide the kernel from other programs
- Hide programs from each other
- Convince each program it’s got its own private memory starting at address zero

Different computers do this differently (e.g. segments vs. paging) but all except the smallest embedded ones do it.
Privileged instructions only the OS can use

The hardware has special instructions that only the kernel can use to:

* Initiate I/O
* Set clocks and timers
* Control memory mapping

The Kernel runs in “privileged” or “kernel” or “supervisor” state.

Ordinary programs run in “user mode”.

*If a user program tries a privileged operation, the hardware will tell the kernel!*
Abstractions for Convenient, Portable, Shareable I/O
Convenient abstractions for I/O

CPU

OPERATING SYSTEM KERNEL

Angry Birds

Play Video

Browser

Keyboard, mouse, display

Disk

Printer
Convenient *abstractions* for I/O

Devices provide very primitive services (store a block, send network packet, signal a keypress)
Convenient abstractions for I/O

The OS gives applications access through convenient, higher level abstractions.

CPU

Angry Birds    Play Video    Browser

Filesyste, Graphics System, Window system, TCP/IP Networking, etc., etc.

Keyboard, mouse, display

Disk

Printer
Disk data abstracted as *files*.
Linux provides convenient abstractions for

- File access
- Network
- Audio / Video
- Process creation & control
- Timing
- Memory management

- Graphics / Windows
- Monitoring execution
- Thread management
- I/O access
- Inter-process communication
- Etc.
Sharing devices

Programs need **coordinated** access to **shared devices**
What gets *shared*?

**QUESTION?**
Would I want to give the faculty access to disk block number 3752344 or to comp111.grades?
What gets *shared*?

Angry Birds  \(\rightarrow\)  FILES  \(\rightarrow\)  File system, Graphic system, Window system, TCP/IP Networking, etc., etc.

Play Video  \(\rightarrow\)  FILES  \(\rightarrow\)  Browser  \(\rightarrow\)  FILEs

CPU  \(\rightarrow\)  FILES  \(\rightarrow\)  Keyboard, mouse, display  \(\rightarrow\)  OPERATING SYSTEM KERNEL

Disk  \(\rightarrow\)  Printer
What gets shared?

Files are the main abstraction for persistent data storage.

Files are the units of sharing and access control in a Unix/Linux system.

chgrp faculty comp111.grades // give faculty access
  // to grades file

We’re controlling access to a file!!
...and your point is?

Many of the same abstractions that make things easier to use also are the ones we need to control sharing!
Abstraction and Information Hiding in Operating Systems
Abstractions hide unnecessary details

- **The file abstraction hides:**
  - Where each file lives on disk
  - The size of the disk, how many blocks it has, how to program it
  - Whether it’s a spinning disk, SSD, USB key, floppy disk, etc.

- **The tty abstraction hides:**
  - What type of keyboard you have, whether you are on an actual character terminal or a simulated XTerm,

- **The process memory abstraction hides**
  - Details of the computer’s memory mapping hardware structure
  - (lots of other stuff too)

- **By hiding these things, we tend to make your programs portable**
- **We can use the same APIs for many different types of data and devices**
- **We also isolate the parts of the OS that must be changed in order to support new devices (USB key) or to run on a different CPU**

**KEY POINT!!** Hiding information can make your interfaces more powerful, easier to use and more portable
A bit of history

- Unix was designed by Ken Thompson & Dennis Ritchie in 1969 at Bell Labs and recoded in C as a portable system in 1973
- It was the first serious attempt at a *portable* operating system
  - Until that time, each family of hardware had its own OS
  - You couldn’t easily move programs across different brands of computer
- There were arguably two key innovations that led to success
A bit of history

- Unix was designed by Ken Thompson & Dennis Ritchie in 1969 at Bell Labs and recoded in C as a portable system in 1973.
- It was the first serious attempt at a portable operating system.
  - Until that time, each family of hardware had its own OS.
  - You couldn’t easily move programs across different brands of computer.
- There were arguably two key innovations that led to success:
  - The invention and use of “C” as a portable programming language for system use.
A bit of history

- Unix was designed by Ken Thompson & Dennis Ritchie in 1969 at Bell Labs and recoded in C as a portable system in 1973

- It was the first serious attempt at a *portable* operating system
  - Until that time, each family of hardware had its own OS
  - You couldn’t easily move programs across different brands of computer

- There were arguably two key innovations that led to success
  - The invention and use of “C” as a portable programming language for system use
  - The brilliant choice of simple portable, abstractions for processes, file I/O etc.
A bit of history

- Unix was designed by Ken Thompson & Dennis Ritchie in 1969 at Bell Labs and recoded in C as a portable system in 1973
- It was the first serious attempt at a portable operating system
  - Until that time, each family of hardware had its own OS
  - You couldn’t easily move programs across different brands of computer
- There were arguably two key innovations that led to success
  - The invention and use of “C” as a portable programming language for system use
  - The brilliant choice of simple portable, abstractions for processes, file I/O etc.
  - Almost all the file and process system calls we use today are the ones invented by Ritchie and Thompson around 1969

Dennis Ritchie paper on the history of Unix:
http://cm.bell-labs.com/cm/cs/who/dmr/hist.html
PDP-7: the first Unix was written for this

I couldn’t find information about this PDP-7, but its replacement, a PDP-11 used for the next version of Unix had (quoting Ritchie):

* 24K bytes of core memory (16K for the system, 8K for user programs)

* A disk with 1K blocks (512K bytes). Each file was limited to 64K bytes.

Isn’t it astonishing that the abstractions they invented are the ones we are using in COMP 111 on machines that run at > 1 Ghz, have many gigabytes of memory, and disks that each hold terabytes of data???
Avoiding Unnecessary Dependencies in the OS
The OS chooses not to depend on:

- The language in which a user program is written
  - C?, C++?, Java?, Fortran?, Python?
  - The types of variables in a compiled program
  - The types of stack and dynamically allocated variables

- The function of a specific device driver
  - The meaning of driver private data
  - Which driver connects to which physical device

- Etc., etc.

The less the OS constrains what goes on in a process or device driver, the more flexible it can be in supporting new programming languages, I/O devices, etc.

However: we’ll see in a minute that the OS has a few features to help common languages with procedure calls and shareable libraries run efficiently:
Summary: Information hiding and abstraction

- The core abstractions in Unix/Linux have been artfully chosen to support a very broad range of needs, and to evolve with changing hardware for decades. To enable this:
  - The interface provided to processes abstract away non-essential details of the hardware and OS
  - The OS avoids depending unnecessarily on details of the programs to be run or the languages in which they are written
Today

- **The Big Picture:**
  - How the OS is organized
  - How good OS services are designed
- **Process:** the most important OS abstraction
- How the OS interacts with processes
- How processes use memory
Processes
The Fundamental Unit of Work
There is one OS “process” for each running copy of a program.
What is process?

- The fundamental unit of work: a running (or paused) copy of a program

- **Processes have:**
  - A process ID (PID): a unique integer identifier
  - Memory segments for code and data (we’ll discuss later)
  - Machine-level execution state: program counter, registers, etc.
  - Owner - a userid (a process has its owner’s access permissions)
  - Scheduling priority
  - Current working directory (where files with relative names like “xxx” will open)
  - Zero or more open files (e.g. stdin or files opened with “open/fopen”)
  - A parent process
  - (maybe) child processes
  - Accounting information: how much time used
  - Etc., etc.

- If you understand what a process is and how it runs, you know a lot about how the OS works!
Today

- **The Big Picture:**
  - How the OS is organized
  - How good OS services are designed
- **Process:** the most important OS abstraction
- How the OS interacts with processes
- How processes use memory
How can your process “call” the kernel?

What about an ordinary function call?

```c
read(myfile...);
```

That won’t work…your process isn’t even allowed to access kernel memory!
How can your process “call” the kernel?

The answer: A small glue function (e.g. `read()`) prepares its arguments in registers or stack, then executes an instruction that “traps” to the kernel. This is known as a “system call”.

The kernel, running in privileged mode, looks at your arguments, decides what service you want, and calls it.

When work is done, the kernel leaves result info in your registers and memory, and returns to the instruction after the trap, using HW mechanisms to return to “user mode”.
Linux provides system calls for

- Process creation & control
- File access
- Network
- Timing
- Memory management
- Monitoring execution
- Thread management
- I/O access
- Inter-process communication
- Etc.
Libraries vs. System Calls

Library routines like `sqrt` & `fread` are ordinary functions provided with the system...they run in user memory.

Some library routines use system calls.
Today

- The Big Picture:
  - How the OS is organized
  - How good OS services are designed
- Process: the most important OS abstraction
- How the OS interacts with processes
- How processes use memory
Process Memory
The process memory illusion

- Process thinks it's running in a private space
- Separated into segments, from address 0
- Stack: memory for executing subroutines
- Heap: memory for malloc/new
- Global static variables
- Text segment: where program lives

Loaded with your program
The process memory illusion

```c
char notInitialized[10000];
char initialized[] = "I love COMP 111";

int main(int argc, char *argvp[]) {
    float f;
    int i;

    // yes, we should check return codes
    char *cp = malloc(10000);
}
```
The process memory illusion

```c
char notInitialized[10000];
char initialized[] = "I love COMP 111";

int main(int argc, char *argvp[]) {
    float f;
    int i;

    // yes, we should check return codes
    char *cp = malloc(10000);
}
```

The process memory illusion

```c
char notInitialized[10000];
char initialized[] = "I love COMP 111";

int main(int argc, char *argvp[]) {
    float f;
    int i;

    // yes, we should check return codes
    char *cp = malloc(10000);
}
```
The process memory illusion

```c
int main(int argc, char *argv[]) {
  float f;
  int i;

  // yes, we should check return codes
  char *cp = malloc(10000);
}
```

```c
char notInitialized[10000];
char initialized[] = "I love COMP 111";
```
The process memory illusion

```c
char notInitialized[10000];
char initialized[] = "I love COMP 111"

int main(int argc, char *argv[]) {
    float f;
    int i;

    // yes, we should check return codes
    char *cp = malloc(10000);
}
```
The process memory illusion

```c
char notInitialized[10000];
char initialized[] = "I love COMP 111";

int main(int argc, char *argv[], char *envp[]) {
    float f;
    int i;

    // yes, we should check return codes
    char *cp = malloc(10000);
}
```
The process memory illusion

- Process thinks it's running in a private space
- Separated into segments, from address 0
- Stack: memory for executing subroutines

- `malloc/free` are *library* functions that run in user mode to allocate space within the heap....

  ...when they run out of space to play with, they call the `sbrk` system call to ask the OS to grow the heap.

- Global static variables
- Text segment: where program lives
- Stack: memory for executing subroutines
The process memory illusion

```c
char notInitialized[10000];
char initialized[] = "I love COMP 111";

int main(int argc, char *argvp[]) {
    float f;
    int i;

    // yes, we should check return codes
    char *cp = malloc(10000);
}
```
The process memory illusion

```c
char notInitialized[10000];
char initialized[] = “I love COMP 111”;

int main(int argc, char *argvp[]*) {
  float f;
  int i;

  // yes, we should check return codes
  char *cp = malloc(10000);
}
```
Of course, the stack enables recursion

```c
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

The diagram illustrates the stack with different sections:
- **Stack**:
  - n=4
  - n=3
  - n=2
  - n=1
- **Heap** (malloc’d):
- **Static uninitialized**
- **Static initialized**
- **Text (code)**

The stack is used to store the function calls and variables for the recursive calls to `factorial`. The stack frame for each call is pushed onto the stack, and the variables are initialized within each frame.
Of course, the stack enables recursion

```c
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

Loaded with your program
The process memory illusion

In Linux, there is a system call to grow the heap, but not the stack. When the system faults on an access to the next page below the end of the stack, the OS maps more memory automatically.

(up to configurable limit).

More on program calls and the stack can be found in Prof. Couch’s notes at: http://www.cs.tufts.edu/comp/111/notes/The_Visible_OS.pdf
The process memory illusion

- Process thinks it's running in a private space.
- Separated into segments, from address 0:
  - Text segment: where program lives.
  - Global variables.
  - Heap: memory for malloc/new.
  - Stack: memory for executing subroutines.

Memory between the stack and the heap is not mapped by the OS.

As far as the process is concerned, it doesn’t exist. Access causes a segfault.
The process memory illusion

- Process thinks it's running in a private space.
- Separated into segments, from address 0:
  - Text segment: where program lives.
  - Global variables.
  - Heap: memory for malloc/new.
  - Stack: memory for executing subroutines.

Surprisingly: the kernel actual lives in every process space at the “top”.

The maps are set so ordinary user code segfaults on access, but...

...when in the kernel executing a system call, the system can access its own kernel segments as well as the user’s.
Summary

- The Big Picture:
  - How the OS is organized
  - How good OS services are designed
- Process: the most important OS abstraction
- How the OS interacts with processes
- How processes use memory
Thank you!