threads, virtual memory, devices, process isolation, windows

processors, memory, interrupts, modes, displays
OS vs. Kernel

Operating System
OS vs. Kernel

System Libraries & Applications

Operating System

Kernel
Kernel Features

• **Processes** for running multiple programs/instances
• **Threads** for managing CPUs
• **Virtual memory** for allocating memory
• **Sockets** for networking
• **Filesystems**\(^\dagger\) for persistent storage
• **Device drivers** for plugging in new functionality
• **Users and groups** for controlling permissions
• **Windows**\(^*\) for managing screen real estate and input

\(^\dagger\) usually pluggable for different formats and devices

\(^*\) to varying degrees
Kernel vs. User Code

When you turn on a processor, instructions can do anything: the processor starts in **privileged mode**

```
mov 42, 0x75462
```

Details here are inspired by x86, but not true-to-life
Kernel vs. User Code

One of the things you can do in priviledged mode is change the way that \textbf{virtual addresses} are mapped to physical memory.

\texttt{mov 43, 0x75462}

So, you can hide memory from unpriviledged code but, before you do that...
Kernel vs. User Code

A certain area in memory, not normally made accessible, contains a table of functions called for special events:

```
0x100
  on_divide_by_zero
  on_sysenter
  ...
```

The `sysenter` instruction jumps to one of those

The jump ignores address remappings and switches back to privileged mode

Control the table, and you control the way back to privileged mode
System Calls

A process asks the OS to do something by making a **system call:**

\[
\text{mov } $57, \%EAX \\
\text{sysenter}
\]

This is a kind of function call, while also switching to privileged mode

Instead of assembly code, you normally use a wrapper C function
System Calls

Some (C wrappers for) typical system calls on Unix:

• create a process: `fork()`
• open a file: `open()`
• allocate memory: `mmap()`
• create a network connection: `connect()`

A system call’s `man` page will say ``` (2)`"
Some System Applications

A command-line **shell** is just a program:

- It uses **fork()** to create new processes
  
  Windows: `CreateProcess()`

- A new processes uses **execve()** to load a program into the process
  
  Windows: `CreateProcess()` does that, too

- The **execve()** system call also handles command-line arguments
  
  Windows: `CreateProcess()` does that, too

  see `exec.c`
Some System Applications

A *desktop GUI* is just a program:

- It uses `open()` to read directory and file information
- It uses other system calls\(^\dagger\) to open windows, draw on them, and receive mouse events
  \(^\dagger\) or communicates with a semi-privileged window-manager program
- If you double-click an application, it uses `fork()`, etc.

see `dir.c`
Some System Applications

A **debugger** like **gdb** is just a program:

- Of course, it uses **fork()** ...
- It uses a system call to attach to a process
  
  Based the process’s user, the request may be declined
- It uses various system calls to inspect a process
- It uses various system calls to receive **signals**
  
  ◦ e.g., “the process seg faulted”
  
  each process has a table of signal callbacks

see **signal.c**
Some System Applications

A **web browser** is just a program:

- It uses system calls like `connect()` to contact a server
- It uses other system calls\(^\dagger\) to open windows, draw on them, and receive mouse events
  \(^\dagger\) or communicates with a semi-privileged window-manager program
- It runs Javascript program in the same way that our interpreter runs MiniRacket programs

see `connect.c`
Writing Portable Applications

```c
fopen("data.txt", "r");
```

```c
FILE* fopen(char *name, char *mode)
{
    ....
    open(name, flag)
    ....
}
```

**unix_file.c**

```c
FILE* fopen(char *name, char *mode)
{
    ....
    CreateFile(name, ....)
    ....
}
```

**win_file.c**
Writing Portable Applications

```c
#ifdef _WIN32
    .... VirtualAlloc(....) ....
#endif
#ifdef linux
    .... mmap(....) ....
#endif
#ifdef OS_X
    .... vm_allocate(....) ....
#endif
main.c

#ifdef is a last resort
```
Applications on Linux

Linux “proper” is just the kernel:

• Processes, users and groups, filesystems, etc.

• New devices and features are exposed through the filesystem

  e.g., `cat /proc/cpuinfo`

The kernel does not include graphics
Applications on Linux

A *distribution* pairs the kernel with particular applications and libraries

- Ubuntu
- Debian
- Fedora
- Gentoo

These differ in look-and-feel, but they’re about the same to an application developer
Applications on Linux

Core graphics functionality is provided by the X Windowing System, a.k.a. X11

X11 is just a program, and others connect to it

Program connections can even go across a network

see x11.c
Applications on Linux

The X11 primitive layer:

- Drawing:

  ![X11 vs Modern Diagram]

  - GUIs: XCreateWindow()

  no buttons, menus, ...
Applications on Linux

- Application
- Gtk
- Pango
- Cairo
- X11
- D-Bus
- glib
- libc
- Kernel
Applications on Linux

- Application
- Gtk
- Pango
- Cairo
- X11
  - D-Bus
  - libc
- Kernel

*System call wrappers, traditional C functions*
Applications on Linux

Application

Gtk

Pango

Cairo

X11

D-Bus

glib

libc

Kernel

Communication among applications
Applications on Linux

Application

Gtk

Pango

Cairo

X11  D-Bus  glib

libc

Kernel

Modern C: reference counting, objects, text
Applications on Linux

- Application
- Gtk
- Pango
- Cairo
  - Drawing
- X11
- D-Bus
- glib
- libc
- Kernel
Applications on Linux

Application

Gtk

Typesetting

Pango

Cairo

X11

D-Bus

glib

libc

Kernel
Applications on Linux

- Application
- GTK
- Pango
- Cairo
- X11
- D-Bus
- glib
- libc
- Kernel

GUI widgets
Applications on Linux

In practice:

• First, you pick a set of libraries to build on

  Gtk is just one option for GUls, though probably the most popular

• Documentation is distributed among producers of different libraries

• Usually, you can look at a library’s source code

  With respect to documentation quality, this is both good and bad
Applications on Windows

Everything is built into Windows:

• Processes, users and groups, filesystems, etc.
• Graphical windows also primitive kernel objects
• Unicode wired deeply into the kernel

The Windows OS API is traditionally called **Win32**
Applications on Windows

Creating a button in Win32:

```
CreateWindow("BUTTON", "Click Me",
            WS_CHILD | WS_CLIPSIBLINGS,
            0, 0, 100, 50,
            container, NULL, NULL, NULL);
```

Creating a Chinese button in Win32:

```
CreateWindowW(L"BUTTON", L"打这里",
             WS_CHILD | WS_CLIPSIBLINGS,
             0, 0, 100, 50,
             container, NULL, NULL, NULL);
```
Applications on Windows
Applications on Windows

Processes and files

kernel32  gdi32  user32  ole32  libc

Win32
Applications on Windows

kernel32  gdi32  user32  ole32  libc

Drawing and text

Win32
Applications on Windows

Win32

kernel32  gdi32  user32  ole32  libc

Application  GUI
Applications on Windows

Application communication via **COM**

- kernel32
- gdi32
- user32
- ole32
- libc

Win32
Applications on Windows

kernel32  gdi32  user32  ole32  libc

Traditional C library

Win32
Applications on Windows

In practice:

- “Everything” is built in, but there are some choices
  - Win32: C API
  - MFC: C++ wrapper on Win32

  Non-C languages are more common on Windows

- Documentation is centralized at MSDN

- COM is sometimes used to glue together applications

  In contrast, stdio-based subprocesses are more common in Unix
Applications on Mac OS X

The Mac OS X kernel is called **Mach**

- Processes, memory management, message passing
- New devices/features accessed via message passing

The goal was to make the kernel as small as possible
Applications on Mac OS X

**XNU** is a Unix-like kernel layer on Mach

- Adds filesystems, users and groups, etc.
- Based on BSD
Applications on Mac OS X

Add system libraries and applications to XNU, and you get *Darwin*

- Darwin
- XNU
- Mach

This layer makes application development on Mac OS X feel like Unix
Applications on Mac OS X

Application

Cocoa

libc Core Foundation Quartz

Darwin
Applications on Mac OS X

Application

Traditional C functions

libc  Core Foundation  Quartz

Darwin
Applications on Mac OS X

Application

Cocoa

Modern C functions

libc

Core Foundation

Quartz

Darwin
Applications on Mac OS X

Application

Drawing and windowing

libc  Core Foundation  Quartz

Darwin
Applications on Mac OS X

- Applications
  - GUI controls
- Cocoa
- libc
- Core Foundation
- Quartz
- Darwin
Applications on Windows

In practice:

• Major libraries packaged by Apple, usually one per goal
  but legacy libraries are commonly in use: Carbon, QuickDraw, ATSUI

• Documentation is centralized at Apple’s developer site

• Library layers (e.g., Core Foundation) are commonly referenced
  feels more like Linux, less like Win32

• GUIs usually written in Objective-C
  ... which is a hybrid of C and Smalltalk
# Portable GUI Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gtk</td>
<td>Gtk</td>
</tr>
<tr>
<td>Pango</td>
<td>Pango</td>
</tr>
<tr>
<td>Cairo</td>
<td>Cairo</td>
</tr>
<tr>
<td>X11</td>
<td>kernel32</td>
</tr>
<tr>
<td>D-Bus</td>
<td>gdi32</td>
</tr>
<tr>
<td>glib</td>
<td>user32</td>
</tr>
<tr>
<td>libc</td>
<td>ole32</td>
</tr>
<tr>
<td>Kernel</td>
<td>libc</td>
</tr>
</tbody>
</table>

Other options in place of Gtk/Pango/Cairo include Qt and wxWidgets