Operating System Structure

Disclaimer: some slides are adopted from the book authors’ slides with permission
Recap

• Architectural support for OS
  – Interrupt, timer
  – User/kernel mode, privileged instructions
  – MMU
  – Synchronization instructions
Today

• OS services
  – User’s perspective
  – What are the major functionalities of an OS?

• OS interface
  – How applications interact with the OS?

• OS structure
  – What are possible structures of an OS?
A View of Operating System Services

User Mode
- User and other system programs
  - GUI
  - Batch
  - Command line
  - User interfaces

Kernel Mode
- System calls
  - Program execution
  - I/O operations
  - File systems
  - Communication
  - Resource allocation
  - Accounting
  - Error detection
  - Protection and security

Operating system

Hardware
Operating System Services

• User interface
  – Command-Line Interface (CLI) vs. Graphical User Interface (GUI)
Command-Line Interface (CLI)

• Command-line interpreter (shell)
  – Many flavors: bash, csh, ksh, tcsh, ...
  – Usually not part of the kernel, but an essential system program

• Allow users to enter text commands
  – Some commands are built-in
    • E.g., alias, echo, read, source, ...
  – Some are external programs
    • E.g., ls, find, grep, ...

• Pros and Cons.
  + Easy to implement, use less resources, easy to access remotely
  + Easy to automate
    • E.g., $ grep bandwidth /tmp/test.txt | awk '{ print $2 }'
  – Difficult to learn
Graphic User Interface (GUI)

• GUI
  – Mouse, keyboard, monitor
  – Invented at Xerox PARC, then adopted to Mac, Window,…

• Pros and Cons
  + Easy to use
  - Use more h/w resources

• Many systems support both CLI and GUI
Operating System Services

• File-system service
  – Read/write /create/delete/search files and directories
  – See file information (e.g., file size, creation time, access time, ...)
  – Permission management (read only, read/write, ...)

• Communications
  – Share information between processes in the same computer (Inter-process communication - IPC) or between computers over a network (TCP/IP)
Operating System Services

• Resource allocation
  – CPU cycles, main memory space, file space, I/O devices

• Accounting
  – Keeping track of who uses what for how much

• Security
  – Login, administrators vs. normal users vs. guests
Operating System Services

• Protection
  – Prevent memory corruption between multiple user programs and between user programs and the kernel
  – Detect and report errors
    • Divide by zero, access violation, hardware faults, ...

A problem has been detected and Windows has been shut down to prevent damage to your computer.

DRIVER_IRQL_NOT_LESS_OR_EQUAL

If this is the first time you’ve seen this Stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup Options, and then select Safe Mode.

Technical information:
*** STOP: 0x000000D1 (0x000000C,0x00000002,0x00000000,0xF86B5A89)
***

Beginning dump of physical memory
Physical memory dump complete. Contact your system administrator or technical support group for further assistance.
System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C)
- Most programmers do not directly use system calls
  - They use more high level APIs (i.e., libraries)
  - But system programmers (or you) do use system calls
- Two most popular system call APIs
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
Example

• Copy the contents of one file to another file

```c
int main(int argc, char *argv[])
{
    int src_fd, dst_fd; char buf[80]; int len;

    src_fd = open(argv[1], O_RDONLY);
    dst_fd = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC);

    while ((len = read(src_fd, buf, 80)) > 0) {
        write(dst_fd, buf, len);
    }

    printf("Done\n");
    return 0;
}
```
Example

• Copy the contents of one file to another file

```c
int main(int argc, char *argv[]) {
    int src_fd, dst_fd; char buf[80]; int len;

    src_fd = open(argv[1], O_RDONLY);
    dst_fd = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC);

    while ((len = read(src_fd, buf, 80)) > 0) {
        write(dst_fd, buf, len);
    }

    printf("Done\n");
    return 0;
}
```

Syscalls: open, read, write
Non-syscall: printf
System Call API Description

- $ man 2 read

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

**DESCRIPTION**

*read()* attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

If *count* is zero, *read()* returns zero and has no other results. If *count* is greater than *SSIZE_MAX*, the result is unspecified.

**RETURN VALUE**

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because *read()* was interrupted by a signal. On error, -1 is returned, and *errno* is set appropriately. In this case it is left unspecified whether the file position (if any) changes.

**ERRORS**

- **EAGAIN** The file descriptor *fd* refers to a file other than a socket and has
API - System Call - OS

user application

open()

user mode

system call interface

kernel mode

open()

Implementation of open()

system call

return
Standard C Library Example

- C program invoking `printf()` library call, which calls `write()` system call
Types of System Calls

• Process control
  – Create/terminate process, get/set process attributes, wait for time/event, allocate and free memory

• File management
  – create, delete, open, close, read, write, reposition
  – get and set file attributes

• Device management
  – request device, release device, read, write, reposition, get device attributes, set device attributes

• Communications
  – create, delete communication, send, receive messages

• Protection
  – Control access to resources, get/set permissions, allow and deny user access
# Examples of Windows and Unix System Calls

<table>
<thead>
<tr>
<th></th>
<th>Windows</th>
<th>Unix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Process Control</strong></td>
<td>CreateProcess()</td>
<td>fork()</td>
</tr>
<tr>
<td></td>
<td>ExitProcess()</td>
<td>exit()</td>
</tr>
<tr>
<td></td>
<td>WaitForSingleObject()</td>
<td>wait()</td>
</tr>
<tr>
<td><strong>File Manipulation</strong></td>
<td>CreateFile()</td>
<td>open()</td>
</tr>
<tr>
<td></td>
<td>ReadFile()</td>
<td>read()</td>
</tr>
<tr>
<td></td>
<td>WriteFile()</td>
<td>write()</td>
</tr>
<tr>
<td></td>
<td>CloseHandle()</td>
<td>close()</td>
</tr>
<tr>
<td><strong>Device Manipulation</strong></td>
<td>SetConsoleMode()</td>
<td>ioctl()</td>
</tr>
<tr>
<td></td>
<td>ReadConsole()</td>
<td>read()</td>
</tr>
<tr>
<td></td>
<td>WriteConsole()</td>
<td>write()</td>
</tr>
<tr>
<td><strong>Information Maintenance</strong></td>
<td>GetCurrentProcessID()</td>
<td>getpid()</td>
</tr>
<tr>
<td></td>
<td>SetTimer()</td>
<td>alarm()</td>
</tr>
<tr>
<td></td>
<td>Sleep()</td>
<td>sleep()</td>
</tr>
<tr>
<td><strong>Communication</strong></td>
<td>CreatePipe()</td>
<td>pipe()</td>
</tr>
<tr>
<td></td>
<td>CreateFileMapping()</td>
<td>shmget()</td>
</tr>
<tr>
<td></td>
<td>MapViewOfFile()</td>
<td>mmap()</td>
</tr>
<tr>
<td><strong>Protection</strong></td>
<td>SetFileSecurity()</td>
<td>chmod()</td>
</tr>
<tr>
<td></td>
<td>InitializeSecurityDescriptor()</td>
<td>umask()</td>
</tr>
<tr>
<td></td>
<td>SetSecurityDescriptorGroup()</td>
<td>chown()</td>
</tr>
</tbody>
</table>
Operating System Structure

- Simple structure – MS-DOS
- **Monolithic kernel** – UNIX
- Microkernel – Mach
MS-DOS Structure

- Written to provide the most functionality in the **least space**
- Minimal functionalities
- Not divided into modules
- Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated
UNIX: Monolithic Kernel

- Implements CPU scheduling, memory management, filesystems, and other OS modules all in a single big chunk

Pros and Cons

+ Overhead is low
+ Data sharing among the modules is easy
  - Too big. (device drivers!!!)
  - A bug in one part of the kernel can crash the entire system
Loadable Kernel Module

• Dynamically load/unload new kernel code
  – Linux and most today’s OSes support this

Pros and Cons
  + Don’t need to have every driver in the kernel.
  + Easy to extend the kernel (just like micro kernel. See next)
  – A bug in a module can crash the entire system
Microkernel

- Moves as much from the kernel into user space
- Communicate among kernels and user via message passing

**Pros and Cons**

+ Easy to extend (user level driver)
+ More reliable (**less code** is running in kernel mode)
- **Performance overhead** of user space to kernel space communication
Hybrid Structure

• Most modern operating systems are actually not one pure model
  – Hybrid combines multiple approaches to address performance, security, usability needs
  – Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  – Windows mostly monolithic, plus microkernel for different subsystem personalities

• Apple Mac OS X
  – hybrid, layered
  – Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called kernel extensions)
OS Structure Comparison

Source: http://en.wikipedia.org/wiki/Monolithic_kernel