Operating System Structure

Heechul Yun

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

Fast, Expensive

Slow, Inexpensive
Recap

• OS needs to understand architecture
  – Hardware (CPU, memory, disk) trends and their implications in OS designs

• Architecture needs to support OS
  – Interrupts and timer
  – User/kernel mode and 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

Kernel Mode

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

The first commercial GUI from Xerox Star workstation. (source: Wikipedia)
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: 0x00000001 (0x0000000C,0x00000002,0x00000000,0xF86B5A89)

*** gv3.sys - Address F86B5A89 base at F86B5000, DateStamp 3dd991eb

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`
API - System Call - OS

user application

system call interface

user mode

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

Heechul Yun

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

• OS services
  – Resource (CPU, memory) allocation, filesystem, communication, protection, security, I/O operations

• OS interface
  – System-call interface

• OS structure
  – Monolithic, microkernel
  – Loadable module
Roadmap

• Beginning of a series of **important** topics:
  – Process
  – Thread
  – Synchronization

• Today
  – Process concept
  – Context switching
Process

• Process
  – An OS abstraction represents a running application

• Three main components
  – Address space
    • The process’s view of memory
    • Includes program code, global variables, dynamic memory, stack
  – Processor state
    • Program counter (PC), stack pointer, and other CPU registers
  – OS resources
    • Various OS resources that the process uses
    • E.g.) open files, sockets, accounting information
Process Address Space

- **Text**
  - Program code

- **Data**
  - Global variables

- **Heap**
  - Dynamically allocated memory
    - i.e., `malloc()`

- **Stack**
  - Temporary data
  - Grow at each function call
Process Address Space

- Each process has its own **private** address space
  - $2^{32}$ (4GB) of **continuous memory** in a 32bit machine
  - Each has same address range (e.g., 0x0 ~ 0xffffffff)
  - How is this possible?
    - What if you have less than 4GB physical DRAM?
    - What if you have 100 processes to run?

- **Virtual memory**
  - An OS mechanism providing this **illusion**
  - We will study it in great detail later in the 2\textsuperscript{nd} half of the semester
Virtual Memory vs. Physical Memory

Virtual Memory

Physical Memory
– **running**: Instructions are being executed
– **waiting**: The process is waiting for some event to occur
– **ready**: The process is waiting to be assigned to a processor
Process Control Block (PCB)

• Information associated with each process
  – Process id
  – Process state
    • running, waiting, etc.
  – Saved CPU registers
    • Register values saved on the last preemption
  – CPU scheduling information
    • priorities, scheduling queue pointers
  – Memory-management information
    • memory allocated to the process
  – Accounting information
    • CPU used, clock time elapsed since start, time limits
  – OS resources
    • Open files, sockets, etc.
Process in Linux

Represented by the C structure `task_struct` (include/linux/sched.h)

```c
pid t_pid;          /* process identifier */
long state;        /* state of the process */
u64 vruntime;      /* CFS scheduling information */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
cputime_t utime, stime; /* accounting information */
struct thread_struct thread; /* CPU states */
```

(very big structure: 5872 bytes in my desktop *)

(*) # cat /sys/kernel/slab/task_struct/object_size
Process Scheduling

• Decides which process to run next
  – Among ready processes

• We cover in much more detail later in the class
  – but let’s get some basics

• OS maintains multiple scheduling queues
  – Ready queue
    • ready to be executed processes
  – Device queues
    • processes waiting for an I/O device
  – Processes migrate among the various queues
Ready Queue and I/O Device Queues
Process Scheduling: Queuing Representation

- Ready queue
- CPU
- I/O
- I/O queue
- I/O request
- Time slice expired
- Child executes
- Fork a child
- Interrupt occurs
- Wait for an interrupt
Context Switching

• Suspend the current process and resume a next one from its last suspended state
Context Switching

• Overhead
  – Save and restore CPU states
  – Warm up instruction and data cache
    • Cache data of previous process is not useful for new process

• In Linux 3.6.0 on an Intel Xeon 2.8Ghz
  – About 1.8 us
  – ~ 5040 CPU cycles
  – ~ thousands of instructions
Process Creation

• **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes

• Generally, process identified and managed via a **process identifier (pid)**
A Process Tree in Linux
‘pstree’ output
Process Creation

- UNIX examples
  - `fork()` system call creates new process
  - `exec()` system call used after a `fork()` to replace the process’ memory space with a new program
Example: Forking a Process in UNIX

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }

    else if (pid == 0) { /* child process */
        execlp("/bin/ls","ls",NULL);
    }

    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
                        "C:\WINDOWS\system32\mspaint.exe", /* command */
                        NULL, /* don’t inherit process handle */
                        NULL, /* don’t inherit thread handle */
                        FALSE, /* disable handle inheritance */
                        0, /* no creation flags */
                        NULL, /* use parent’s environment block */
                        NULL, /* use parent’s existing directory */
                        &si,
                        &pi))
    {
        fprintf(stderr, "Create Process Failed");
        return -1;
    }

    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
Process Termination

• Normal termination via `exit()` system call.
  – Exit by itself.
  – Returns status data from child to parent (via `wait()`)
  – Process’s resources are deallocated by operating system

• Forced termination via `kill()` system call
  – Kill someone else (child)

• **Zombie** process
  – If no parent waiting (did not invoke `wait()`)

• **Orphan** process
  – If parent terminated without invoking `wait`
  – Q: who will be the parent of a orphan process?
  – A: Init process
Mini Quiz

int count = 0;
int main()
{
    int pid = fork();
    if (pid == 0){
        count++;
        printf("Child: %d\n", count);
    }
    else{  
        wait(NULL);
        count++;
        printf("Parent: %d\n", count);
    }  
    count++;
    printf("Main: %d\n", count);
    return 0;
}

• Hints
  – Each process has its own private address space
  – Wait() blocks until the child finish

• Output?
  Child: 1
  Main: 2
  Parent: 1
  Main: 2
Mini Quiz

• What are the three components of a process?
  – Address space
  – CPU context
  – OS resources

• What are the steps of a context switching?
  – Save & restore CPU context
  – Change address space and other info in the PCB