Lecture I
Introduction to Operating Systems

§1. Mechanics of Course

This is an introductory course on operating systems. We stress basic principles with a hands-on approach via programming.

¶1. Immediate Action:

1. READING for first 2 weeks: read PART ONE (first 2 chapters) of the book on your own time. There is a lot of information, and some can be skimmed for now.

   Also begin to read Chap.3 (process management). Read up to Section 3.4.

2. Bookmark the class webpage: http://cs.nyu.edu/ yap/classes/os. In particular, homework and lecture notes can be found here.

3. All other class information should be accessible through the class webpage. In particular, you can go to the NYU BBoard page from this page. We mainly use the BBoard for keeping track of course grades (it is your responsibility to check that this is in order), and for you to maintain class discussion on various topics, including homework.

¶2. Goals of this course We want to study the organizational principles of modern operating systems (abbrev. OS). Part of of this knowledge will be acquired through programming assignments and projects.

Besides Windows, we want students to also become familiar with various aspects of Unix (or its derivatives). There is a very easy way to get a free Unix-like environment via Cygwin. Go to my website http://cs.nyu.edu/ yap/prog/cygwin/ for more information and download. Unix development is based on the concept of modular units (programs) called "tools". For instance, we will learn to use the Make program for organizing programming projects.

¶3. General Remarks about READING. Be curious! Try to read ahead of our lectures. Even reading for 10 minutes (just the introduction or section 1 of a chapter) is useful to get an orientation before class lectures. It does not matter if you don't understand all the concepts you read – you actually do gain some rough impressions and recognize the buzzwords of the topic. This alone will improve your understanding of lectures.

We will supplement the text book with our lecture notes. THEY COMPLEMENT EACH OTHER.

¶4. Grade Grade is curved:

- Class Participation (5%). This means you must come to class regularly and participate in discussions.
- Written Homework (20%). Typical homework will have a written and a programming component.
- Programming Homework/Project (30%). This is a key part of this course.
- Midterm (15%). We view this as a dry-run for the final exam. Closed-book with one 8"x11" sheet of notes.
- Final (30%). Closed-book with two 8"x11" sheets of notes.

¶5. Computing Environment: We assume a Unix-like environment. Our default instructions will assume Cygwin environment which is a Unix-like environment, but embedded in the Windows environment. These instructions should apply also to the MAC OS, Linux or other UNIX environments. Cygwin can be freely downloaded from the web, and easy to install.

Although we work in Unix-like Environments for our programming and projects, it does NOT mean that this course is about the Unix OS, and say, to the exclusion of Windows OS's. Rather, we teach general OS principles that are usually applicable to any OS systems.

¶6. Programming Language: We will use the C language. It is natural choice for learning operating systems because (1) low level (bit-level) operations can be achieved very efficiently in C, and (2) many operating systems and low-level programs are written in C. Since all students know Java, the transition to C is relatively easy since Java evolved from C. In any case, we will provide the necessary C information. The default compiler is the free GNU C compiler (gcc) which can be downloaded as part of the cygwin suite.

Along with programming languages, we want you to learn an important software development tool called "make". This tool is useful in organize large software projects. It also serves as special kind of software documentation: namely, a "make file" documents file dependencies, compilation flags, standard tasks and actions.

Our class webpage provides links and additional information of getting started with Cygwin, make and C programming.

¶7. Programming Tools: You will also learn two important software, Make and CVS in the course of doing programming and our projects.

¶8. Random Self-Test Questions The first 2 weeks is aimed at getting you up to speed on working in a unix like environment and programming in C. Here are some random self-test questions.

1. What are the environmental variables PWD, PATH and HOME, and illustrate how each is used in UNIX?
2. What do these special symbols mean in unix:

   ”.”, ”..”, ” ”, ”&””|”

3. What are targets and dependencies in makefiles? How do you use them?
4. What is the difference between a text (ASCII) file in Unix and in Windows?
5. ...

See answers at the end of this lecture.

§2. Overview of OS

¶9. A. Layered View of a Computer (3 layers)

- Hardware/architecture – CPU, MM, IO/Physical Devices
- OS
- Application Programs – word processors, compilers, browsers

¶10. B. What is an OS? Book says an OS is like a ”goverment”, that does no useful function of its own, but to enable others to do useful function.

Four main characterizations of an OS:

- OS as a manager of processes.

  From a user-viewpoint, this is perhaps the most natural characterization of an OS.

  E.g., each user (including the system or root user) owns a set of process. When you log into the machine you have at least one process, but in general, you will have one or more for each task that you are doing. For instance, an editor process might have a spell checker running in the background. These processes must be relatively isolated from each other, must be scheduled fairly, given access to shared resources.

- OS as a resource manager:

  This is just the flip side of the view of manager of processes.

  E.g., CPU time, memory space, storage space, hardware devices (printer, networks, mouse, sound, etc). The goal of resource (and process) management is efficiency, fairness, protection, as well as convenience.

- OS as an extended or virtual machine.

  The OS extends the hardware's raw capability in simple, natural and standard way. E.g., read/write to disk is available in hardware, but the OS adds various conveniences in the system calls.

  From this viewpoint, the full set of these convenient functions (i.e., OS's system calls or API's) constitute the OS.

  E.g., all Unix-derived OS's implement a set of about 100 system calls. This set is defined by an international IEEE standard called POSIX (= Portable Operating System Interface for uniX).

- OS as an abstract machine.

  This is related to the Virtual machine idea, but calling an OS an "abstract machine" stresses the fact that the OS is machine- or hardware-independent. Thus, it allows users to see OS services from a natural viewpoint that does not depend on hardware detail.

  E.g., An OS handles many kinds of I/O devices, each with its peculiarities. By using device drivers with a specific interface, these peculiarities are hiddene peculiarities are hidden

- NOTE: Notice how the core topics in OS are related to the above characterizations: Process Management, Memory management, Storage Management.

¶11. C. OS Architectural Elements

- CPU

  Instruction set, registers

  Program Counter (PC) (or IP, Instruction Pointer)

  Stack Pointer (SP)

  Stack contains a frame for each procedure that is entered

  Program Status Word (PSW) – for system calls, errors, etc

- RAM (or Main Memory, MM)

  Accessed directly by CPU thru LOAD/STORE

  MMU (Memory Management Unit) between CPU and RAM

  MMU to map (i.e., generate physical addr from virtual/logical addr) and do protection checking

- Disk (or Secondary Memory)

  Magnetic technology

  Connected via I/O Bus (ATA,SCSI, etc)

  Disc Controllers

- Other IO Devices (Floppy, CD, Videocam, USB devices, etc)
- Device controllers
- Busses

¶12. Interrupts/Events/System Calls

- CPU can execute in one of two Modes: user/kernel

  Kernel mode has more previleges than User mode. Normally, the OS routines are executed in kernel mode an user routines are executed in user mode.

- Interrupt (aka trap, exception, system call) – generated by user or system or hardware

- E.g., to make a system call for system service

  E.g., I/O controller signals completion of its task

  E.g., Divide by zero error

- Interrupt Processing:

  1. CPU disables further interrupts.

  2. Stops current process and stores the context of the process.

  3. Determines the type of interrupt that has occurred: "polling" or "vectored interrupt system".

  3. Transfers control to a fixed location, which has the starting address of service routine. This address is called the INTERRUPT Vector.

  4. Executes the service routine in KERNEL MODE.

  5. Returns to the interrupted process (in USER MODE).

- FORMS of interrupts:

  We can think of "INTERRUPTS" as a generic name for any action that causes the CPU to instantly respond. But often, it is associated with such actions which are initiated by hardware devices. E.g., I/O Interrupt.

  TRAP: a machine level instruction which is executed during an interrupt.

  SYSTEM CALL: user-initiated interrupts provided by the OS to provide kernel mode service. The set of such calls essentially defines the OS.

  EXCEPTION: error-generated interrupts (e.g., divide-by-zero exception)

- EXAMPLE: "read" system call in UNIX.

  In C program, we can make the system call

  count = READ(fd, buffer, nbytes);

  (1) Calling program program pushes parameters onto stack and then calls the library procedure.

(2) Library procedure, probably written in assembly, puts the system call number in a standard place (some register), and executes a TRAP instruction.

(3) This switches from User to Kernel mode, and begins execution at a fixed address in the kernel.

(4) The kernel code examines the System Call Number and dispatches to the system call handler.

(5) After the system call handler returns, control MAY return to the user-space at the instruction following TRAP instruction.

WHY "MAY" return? Because the system may block the caller, so some other process may be run. In this case, the current process is also blocked.

(6) This library procedure returns to the user program.

(7) User program then cleans up the usual way after a procedure call.

¶13. Memory Hierarchy WLOG, all memory is divided into fixed units called words (32 bits or 64 bits).

Many considerations in memory management are based on the existence of the different types of memory that form a linear **Memory Hierarchy**:

- register (< 20)
- cache ($< 1MB$)
- electronic disk ($< 1GB$, e.g., flash memory, can be volatile or non-volatile)
- magnetic disc (RAM) ($<$ 1 TeraByte)
- (optical disk)
- tape.

The hierarchy is characterized by the capacity-speed tradeoff: as a particular memory type increases in speed, its capacity is decreased.

It may appear paradoxical, but it is possible that we might consider two memory devices that do exhibit the capacity-speed tradeoff. That is because there might be other tradeoffs between these devices: physical size, price, heat dissipation, etc.

¶14. Multiprogramming: why we need it We take multiprogramming for granted today. But it is instructive to see why it is useful.

What is the alternative to multiprogramming? A sequential job queue (perhaps modified by priority). This is old-fashion (up to late 70's) way of running a computer system. Users submit jobs (a stack of punched cards!) in an input-tray, and return an hour later to pick up the output of running their job. If the job did not compile or fail for some reason, you fixed the punched cards and resubmit.

Consider how multiprogramming wins over this scenario. There are two basic SCENARIOS:

(1) Processes can be classified as I/O-bound or CPU-bound.

I/O-bound, e.g., reading and writing files, or interactive programs. CPU-bound, e.g., pure number-crunching application (prime factorization)

While running I/O bound processes, the CPU would be under-utilized while waiting for I/O to complete. Thus, we want to have other processes when this happens. Note that in the pre-1980 era, there were no interactive programs, but there are still I/O bound jobs in printing jobs.

Thus, by introducing multiprogramming, we greatly improve the utilization of CPU cycles, for a slight overhead of slowing down I/O-bound processes.

(2) Processes can be classified into big jobs and small ones. We do not want short jobs to be held up a long time because a long job was just a little ahead of it. So we want to give fixed quantums to each job in the current queue. This way, the user's wait time is roughly proportional to the size of their jobs, all other things equal. This seems fair.

Thus, by introducing multiprogramming, we greatly improve the fairness to small jobs, for a slight overhead of slowing down the big jobs. The swapping of jobs incurs a slight waste of CPU cycles.

Note that the motivation for multiprogramming in these 2 cases are different: the former has to do with optimizing resource usage, the latter has to do with fairness.

§3. OS FUNCTIONS

Perhaps the 3 core functions of an OS are process management, memory management and storage management:

¶15. A. Processes Management

- Perhaps the most important concrete concept!
- Process = program in execution (PC, Image, etc)
- OS has to create/delete/suspend/resume processes
- OS provide for synchronization + communication of processes

¶16. B. Memory Management

- An array of addressable words, faster than secondary memory
- Processes are swapped in/out of MM
- DMA driven I/O requires no CPU intervention

¶17. C. Storage (or file) Management

- Different physical media (disk, tape, CD, etc)
- files and directories
- create/delete/modify
- organization of file (mapping)

¶18. D. Other issues

- I/O Devices
- Networking
- Protection
- Security
- *etc*, ...

§4. VARIETIES OF OS

- Multiprogrammed systems. Rationale: jobs waiting for I/O. Scheduling needed.
- Timesharing systems. Extension of multiprogramming. More complex – need virtual memory for instance.
- Desktop systems. Single user, many jobs.
- Multiprocessor systems. VERY LARGE VARIETY (how tightly coupled, communication, topology, etc). Symmetric multiprocessing (SMP) where each processor runs a copy of OS. Asymetric multiprocessing is more master-slave relations.
- Distributed sytems. Loosely coupled, geographically distributed. Computer networks (LAN, WAD, MAN). Client-Server systems. Peer-to-peer systems. CLUSTERS.
- Realtime systems. Physical process controllers (e.g., Nuclear reactor). Embedded system. Multimedia applications. Soft-realtime.
- Handheld systems. PDAs.

§5. Metric Prefixes

1. Positive Powers of 1000: Kilo (3), Mega (6), Giga (9), Tera (12), Peta (15), Exa (18).

   E.g., one Kilobyte is $1000 = 10^3$ bytes, and one Terabyte is 10^{12} bytes.

2. Negative Powers of 1000: milli (-3), micro (-6), nano (-9), pico (-12), femto (-15).

   E.g., one nanometer is 10^{-9} meters. So if each wire is 1 nanometer, you can lay one million (mega) wires side by side to form a wide of one millimeter.

   E.g., as fractions of a second, we write $1s, 1ms, 1\mu s, 1ns$ for 1 second, 1 millisecond, 1 microsecond, and 1 nanosecond.

3. Fortuitiously, powers of 2 can closely approximate powers of 1000:

   $2^{10} = 1024$ and we consider this to be approximately 1000 (kilo).

   Then $2^{20} = (2^{10})^2$ is approximately $1,000,000$ (mega).

   It is important to realize that, depending on the context, we may regard the metric units (powers of ten) to be the exact values, or the powers of 2 to be the exact values. In general we do not really care about this difference, so you can freely switch between them.

4. "One order of magnitude" means "a multiple of 10".

   E.g., CPU speed is 3 orders of magnitude faster than I/O speed means CPU speed is 1000 times faster than I/O.

   E.g., Moore's Law says: "chip density doubles every 18 months" This usually taken to imply that CPU speeds doubles every 18 months, but this correlation is not direct. Assuming that computing speeds does double every 18 months, it means that we can get 3 orders of magnitude in speedup every 15 years.

§6. Short History

First digital computer: Charles Babbage (1850's)

1950s: mainframe with no OS. batch processing. introduction of punched card programs.

Strachy (1959) proposed time-sharing system. CTSS (Compatible Time-Sharing System) at MIT [Corbato 1962].

1960s: IBM System/360, designed for scientific (number crunching) as well as commercial use (word processing). Multiprogramming introduced. MULTICS (Multiplexed Information and Computing Services) [Corbato, Vyssotsky, 1965].

1970s: Unix from Bell Labs (Ken Thomson)

1980: Personal Computers, DOS ("Disk Operating System") from Microsoft. Windows 95.

1990: Windows 2000, Linux

Exercises

Exercise 6.1: (Device Drivers)
a. What are device drivers?
b. When you add a new device to the system, why do some OS require you to reboot the system?
c. What does it take to implement on-the-fly (hot pluggable) device drivers? ♢

End Exercises

¶19. ANSWERS to Self-Test Questions

1. What are the environmental variables PWD, PATH and HOME, and illustrate how each is used in UNIX?

   A: PWD is the current directory (when your create a new directory or file will be placed in the PWD). PATH is a list of directory paths (when you issue a command, unix will search for the executable command in the paths in these directors). HOME is your home directory (initially, PWD is set to HOME).

2. What do these special symbols mean in unix:

   ".", "..", " ", "&""|"

   A: current directory, parent of current directory, home directory, run in background, pipe.

3. What are targets and dependencies in makefiles? How do you use them?

   A: Targets are names for a series of actions. If "test1" is the name of a target in the Makefile, and you type "make test1", then these actions will be taken.

   Dependencies of a given target "test1" are just a list of other targets. It means that in order to achieve the target "test1", you must first achieve the dependency targets.

4. What is the difference between a text (ASCII) file in Unix and in Windows?

   A: In a Unix file, each new line is represented by a single ASCII character called CR (carriage return). In a Windows file, each new line is represented by a sequence of two ASCII characters called CR and LF (line feed). So window files are slightly more wasteful in space.