# CPS221 Lecture: Operating System Protection

last revised 9/5/12

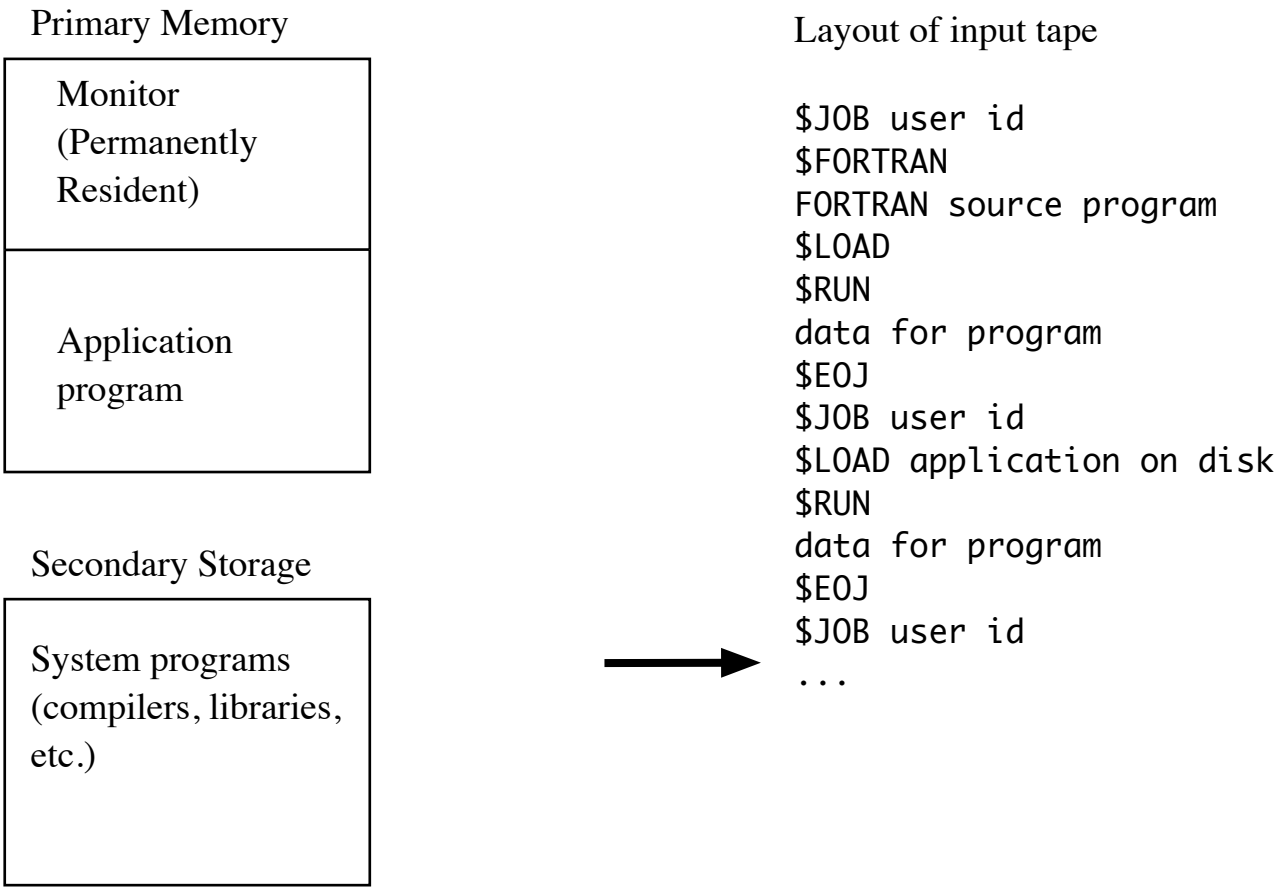
*Objectives*

1. To explain the use of two CPU modes as the basis for protecting privileged instructions and memory
2. To introduce basic protection facilities found in operating systems

## I. CPU Modes

A. In our survey of operating system history we saw that the precursor of today's operating systems was the stacked job batch monitor. When such a system was used, the input stream (card reader or tape) would contain a series of user jobs, structured like this

1. The advent of stacked job monitors created some interesting problems that called for additional hardware features



2. Since a series of jobs would follow one another on the tape, it was vital that one job not be able to do anything that would interfere with the other jobs on the tape:

    a) If one job - through error or maliciousness - damaged the monitor, then the remaining jobs on the tape could be processed incorrectly or not at all.

    b) If one job contained an error it might not stop reading with its own data, but might begin reading the control cards, program, and data for the next job. Thus, when the monitor regained control it would not see some or all of the next job.

    c) A program might go into an infinite loop, blocking all subsequent jobs.

    d) These problems were solved by adding a feature to the CPU architecture.

B. That feature was the concept of **processor mode**: a provision that, at any given time, the CPU would be running in one of two different modes.

    1. The basic idea is this:

        a) When the concept was first introduced, the two modes were called “monitor” and “user”; but today it is more common to call them “kernel” and “user”, and we will use these two terms.

        b) The CPU mode is specified by a single bit in the CPU state.

        c) When the CPU is running code that is part of the operating system, it is in kernel mode; when it is running any other code, it is in user mode. (We’ll talk about how mode switching is accomplished shortly).

2. A stacked job batch monitor would use these two modes as follows:

   a) The monitor region of memory was made "off limits" when the processor was in user mode. If the monitor resided in the lowest memory addresses, a simple way to do this for a stacked job monitor was with a "fence" register - addresses below the fence are accessible (or at least alterable) only when in monitor mode. The fence would be set so that the monitor could not be touched by a program running in user mode. This solved the first problem.

   b) To solve the second problem, it was required that all input and output be done by the monitor, rather than by user programs, by making IO instructions privileged - i.e. executable only in kernel mode. (An attempt to execute one of these in user mode would be treated as a fatal error terminating the program.)

      This means that a user program had to transfer control to the monitor whenever it wants IO done. The monitor could then take steps to be sure that the IO operation is legitimate - e.g. that the user is not attempting to read a control card belonging to the next job (or worse yet to write to the job input tape!)

   c) A timer was used to solve the third problem. The monitor would set it to a suitable interval (often specified by the user on the first control card in the job) before passing control to the user program. If the timer ran out before the user program is finished, it interrupted the CPU, causing transfer to a timer interrupt handler routine which is part of the monitor. This routine would then terminate the offending job with a suitable error dump and then advance the input tape to the next job. (Of course, the instruction to set the timer was privileged!)

      Note: In practice, especially on current machines, timers are often implemented by a device that interrupts the CPU at regular intervals - with a counter in the monitor memory that is decremented by 1 on each interrupt. Timeout is deemed to occur when the counter reaches 0.

3. This marked the introduction of what has become a key concept in operating system design: **protection.**

   a) In a system that is shared in some way among multiple users, it is necessary to protect each user from accidental or intentional interference by others. (This is true whether the users access the system one after another as on a stacked job batch system, or at the same time as on a multiprogrammed system.)

   b) The operating system (which has unlimited access to the system) must be protected from damage by users. This is mandatory on a multiple user system, but even desireable on a one user system

      (1) Example: Gordon's old PDP-8: during one programming project we had a rash of accidental erasures of part of the disk containing the operating system due to a bug that many students seemed to share.

      (2) Example: Software crashes due to errors in one piece of software were quite common on early PC operating systems such as earlier versions of Windows or pre OS-X Macintoshes.

   c) Protection typically requires some hardware facilities - e.g. a two mode CPU with privileged instructions for IO and certain other operations, and some form of memory partitioning. There is thus a strong interaction between operating systems and computer architecture.

   d) Note that protection is not the same thing as security - though the two are closely related:

(1) Protection is concerned with mechanisms that help to ensure correct operation even in the face of erroneous or malicious code.

(2) Security is concerned with taking measures to ensure that the protection mechanisms are not bypassed (e.g. to obtain access to information one is not authorized to see, or to alter information illegitimately, or just to crash the system.)

4. In a multiprogrammed system, memory protection is achieved by arranging for the tables used by the memory management unit to only be writeable when the CPU is in kernel mode.

   a) Thus, only the operating system is able to control the memory mapping for user processes.

   b) An easy way to do this is to have the mapping tables reside in a region of memory that only the operating system can write.

C. But how is the CPU mode changed?

1. For the use of these two modes to achieve the desired result, it is essential that we ensure that only code that is part of the operating system be executed with the CPU in kernel mode. (When the CPU is in user mode, it doesn't matter what code it is executing, since any instruction that could cause a violation of protection is disallowed.)

   a) The CPU's instruction set will include some mechanism for allowing the CPU, when in kernel mode, to change mode to user.

   b) How do we go the other way?

2. The key is the notion of an "interrupt" or "trap".

a) The two terms “interrupt” and “trap” have distinct technical meanings, but are handled similarly, and the one term interrupt is often used collectively to cover both.

(1) An interrupt (in the narrow sense of the word) is an event that is triggered by an external device - e.g. when an IO operation to a device is complete, or when a hardware clock “ticks”.

In the case of an IO interrupt, there is typically some process in the waiting state that is waiting for the operation to complete.

(2) A trap is a CPU event that is triggered by the program currently running on the CPU. There are two sources for traps.

(a) It may be the result of some kind of fault such as executing an illegal instruction or attempting to access a protected region of memory or an IO device.

(b) It may be deliberately triggered by a system call instruction. All CPU architectures include some instruction like this, which the hardware treats the same way it treats an illegal instruction (thus initiating fault processing), but the OS recognizes as a system call rather than a fault.

b) Either sort of event is handled by the hardware in essentially the same way:

(1) The CPU hardware saves minimal information about the current state of the system - at least the program counter and its current mode (user or kernel), and sometimes a bit of other information. (Details vary from architecture to architecture).

(2) The CPU hardware changes mode to kernel

(3) The CPU hardware starts executing appropriate OS code.

(a) On some hardware architectures, the code that is executed is a dispatcher routine in the OS that analyzes the cause of the interrupt/trap and passes control to the correct handler routine in the OS. The CPU architecture specifies where this handler must reside in memory, and the operating system code is set up to ensure that the appropriate dispatcher code resides there.

(b) On other architectures, this analysis is effectively done by the hardware.

(c) In either case, the end result is that the CPU hardware begins executing an appropriate "handler" routine in the OS. There is one handler for each external device that might cause an interrupt, and one handler for each possible cause for a trap.

c) However, what the OS handler routine does depends on the nature of the interrupt or trap.

(1) In the case of interrupts from an IO device, the typical actions include:

(a) Changing the state of the process waiting for the IO operation to complete from waiting to ready.

(b) Possibly initiating a new IO operation on the same device, if there is one waiting. (On a shared device like a disk, one process may request an operation while the device is busy serving another process - in fact, there may be a whole queue of pending requests for a device.)

(2) An internal clock "tick" may result in some process that was waiting a specific period time becoming ready.

(3) In the case of traps resulting from a fault, one of the following is done:

(a) Initiating appropriate exception processing in the process

(b) Terminating the offending process.

(4) In the case of a trap resulting from a request for a system service:

(a) Prior to initiating the trap, the process will have placed information about what it wants in some appropriate place, as specified by the CPU architecture (typically in registers or some specified region of memory).

(b) The system service handler will analyze this information to determine what the process wants and to see if it is legitimate. If it is legitimate, processing will be initiated with the CPU in kernel mode; if it is not legitimate, the request will be handled as if it were a fault.

d) In any case, after processing the interrupt or trap, the OS handler will have to decide whether to resume running the process that was running at the time the trap occurred, or to run some other process - e.g. because a waiting process of higher priority has just become ready, or because the running process has requested an operation for which it must wait.

3. Thus, when code running in user mode needs to perform some privileged operation (such as IO), it sets up parameters to describe its request and then executes the architecture-specific system call instruction. This causes a trap to OS code that runs in kernel mode, which examines the request and performs it if appropriate.

D. The basic idea of a two mode CPU has continued to be a fundamental part of most CPU architectures, with the operating system making use of these modes to protect itself and user code. There are, however, a few refinements that have been added.

1. An operating system may be structured in such a way as to have only a portion of it actually running in kernel mode. The rationale for this is that erroneous code running in kernel mode can do more damage than erroneous code running in user mode, so it is safer to minimize the danger by using user mode for functions that do not require kernel mode privileges.

2. Some CPU architectures provide more than two modes (4 is typical). In this case, the most privileged mode (kernel) is reserved for those portions of the OS that need it, while the modes between kernel and user mode are used for other portions of the OS and/or for middleware.

## II. **Additional Protection Facilities**

A. Operating systems typically protect a number of other resources

1. The right to access the system in the first place - typically by some sort of login mechanism.

2. Access to various files .

   a) Many operating systems use a model similar to that found in traditional Unix., where a file is owned by a specific user, and access rights are specified for the user who owns the file, for others in the same group as the user who created the file, and for the world at large.

(1) In Unix, the password file specifies a user id for each user. When that user logs in, the user's shell (and all processes it creates) normally run with the permissions of that user id.

(2) The file /etc/group specifies various groups. The password file specifies a primary group for each user, but a user may be a member of other groups as specified by /etc/group.

(3) The unit of protection is the file, which may be

  (a) An ordinary file

  (b) A directory (containing a listing of other files)

  (c) A "special file" representing an IO device or an area in kernel memory

(4) Each file has a user and group owner - normally the user and group who created it.

(5) Each file specifies access rights for its user owner, its group owner, and everyone else.

  (a) If a process is running under the user id of the file's owner, the owner access rights govern.

  (b) If the process is running on behalf of a user who is a member of the group owning the file - but not the user owner - the group access rights govern

  (c) In all other cases, the "other" access rights govern

(6) Access rights are specified separately for each category:

  (a) Read access permitted - a process can read a file or list the contents of a directory or read from an IO device or memory

  (b) Write access permitted - a process can write to a file or create or delete files in a directory or write to an IO device or memory

(c) Execute permitted - a process can execute a binary file (to which it also has read access) or perform operations on a directory.

(Recall how you needed to do this with the shell scripts you created in lab)

(7) It is also possible to set bits called "set user id" and "set group id" on a binary executable program. In this case, the program runs with the identity of the file's owner user or group, and therefore has that user/group's access permissions.

b) Some operating systems (including Windows and newer versions of Unix, including Linux), offer a model known as access-control lists, which allows an owner to grant access to specific users - a more flexible protection mechanism than the simple user - group - others model of basic Unix.

3. Various management functions

a) Unix has a rather broad-brush approach to this. There is one user (conventionally known as root or "superuser") who is allowed to do everything - shutdown the system, change ownership or protection of files, etc.

(1) This can create a problem, because the only way to give a user the privilege to perform some "system" operation is to give them all privileges.

(2) However, it is possible to give a user the ability to run a specific program with "system" privileges via the setuid mechanism. (For example, /bin/ps does this to allow access to the kernel's process table.)

(3) In addition, running as the root user is dangerous, because a typographical error ion a command can have disastrous consequences since the system allows the root user to do anything

Example: Barry Reinhold's experience as a system manager

(4) For the latter reason, most Unix-like systems today use a command called sudo which executes a single command with superuser privileges. When first given, sudo requires the user to enter a password; if used again within 5 minutes, it doesn't require a password again.

Most systems require that the user who executes sudo be a listed in a file that specifies who is allowed to use sudo -

(On some Linux systems, the password required is that of root; on many other systems (including Ubuntu and MacOS) it is the password of a user who is declared to be a system administrator in the list of user accounts.

b) On systems designed to use a graphical user interface, such as Ubuntu or MacOS, management utilities may require the user to enter the password of an administrator account before a critical function is done. (What is actually happening behind the scenes is effectively the same as what is done by sudo).

c) Other systems use a more fine-grained approach. For example, recent versions of Windows have 35 distinct privileges which can be assigned to an individual user at login. This allows the use of the principle of minimum privilege - a user should be given the privileges to do what the user needs to do, but nothing more.