Privileged Instructions

Computer instructions are usually divided into two classes:
user instructions and privileged instructions.

User instructions are those that are not privileged.

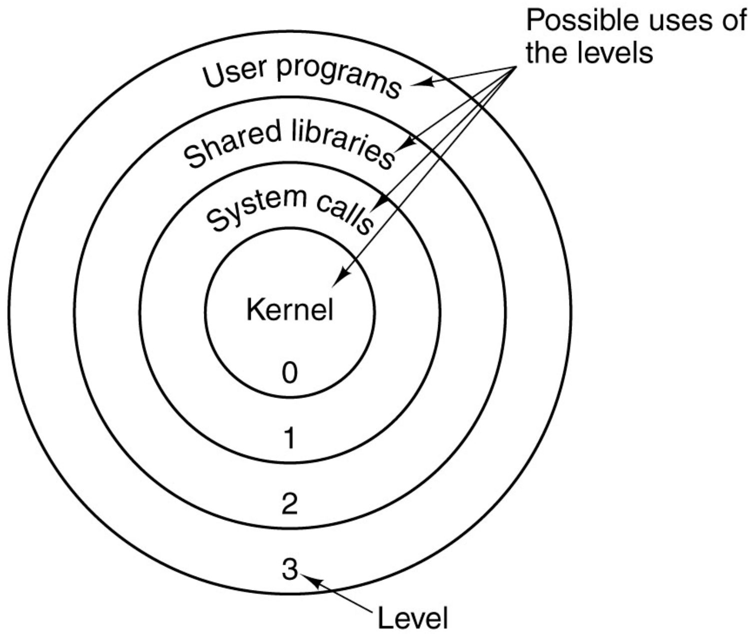
Instructions can be labeled as **privileged** for a number of reasons.

Confusion	Instructions such as input / output instructions can cause difficulties if executed directly by the user. Consider output to a shared print device.
Security	Instructions such as memory management can cause severe security problems if executed by the user. I can directly read and corrupt your program memory.

Rings of Protection

The simple security models in a computer call for rings of protection.

The protection rings offered by the Pentium 4 architecture (IA–32) are fairly typical.



Attempts to read data at higher (less protected) rings are permitted.

Attempts to read data at lower (more protected) rings are not permitted and cause traps to the operating system.

Implementing Rings of Protection

There are two options for these rings of protection.

1. Implementation in software
2. Direct implementation in hardware with sufficient hardware support.

Early experience with the MULTICS operating system showed that direct hardware implementation is necessary.

The reason for this is the efficiency of cross–ring procedure calls.
With protection rings emulated in software, these calls take too much time.

Experience with MULTICS, with protection rings implemented in software, showed that system programmers placed non–security–critical software into the kernel, just to avoid the time delays associated with calling kernel code from the outside.

Experience with MULTICS, running on a computer with hardware support for protection rings, showed that cross–ring calls were not noticeably less efficient than calls within the same protection ring.

The PSW (Program Status Word)

Often called the PSL (Program Status Longword), as with the VAX–11/780.

Not really a word or longword, but a collection of bits associated with the program being executed.

Some bits reflect the status of the program under execution.

N	the last arithmetic result was negative
Z	the last arithmetic result was zero
V	the last arithmetic result caused an overflow
C	the last arithmetic result had a “carry out”

The security–relevant parts of the PSW relate to the protection ring that is appropriate for the program execution.

The VAX–11/780 and the Pentium 4 each offered four protection rings.
The ring number was encoded in a two–bit field in the PSW.

The VAX stored both the ring for the current program and the previous program in the PSW. This allowed a program running at kernel level (level 00) to determine the privilege level of the program that issued the trap for its services.

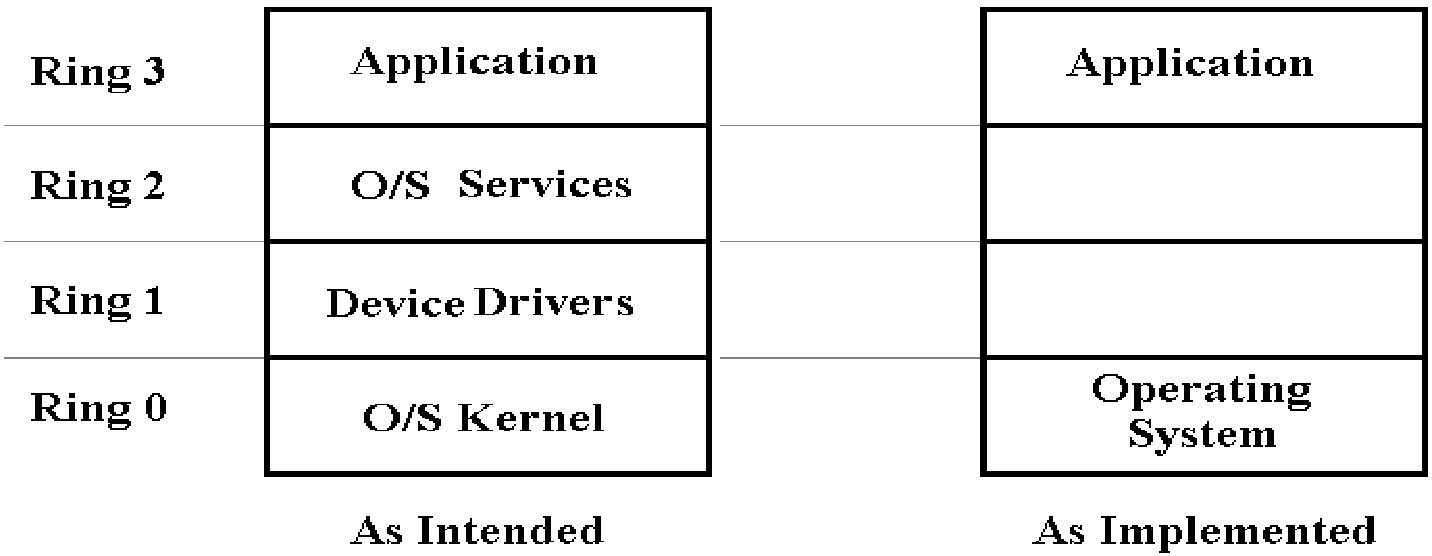
Commercialization of the Protection Rings

Early computers had operating systems tailored to the specific architecture.

Examples of this are the IBM 360 and OS/360; VAX–11/780 and VMS.

More modern operating systems, such as UNIX, are designed to run on many hardware platforms, and so use the “lowest common denominator” of protection rings.

This figure shows the IA–32 protection rings as intended and as normally implemented.

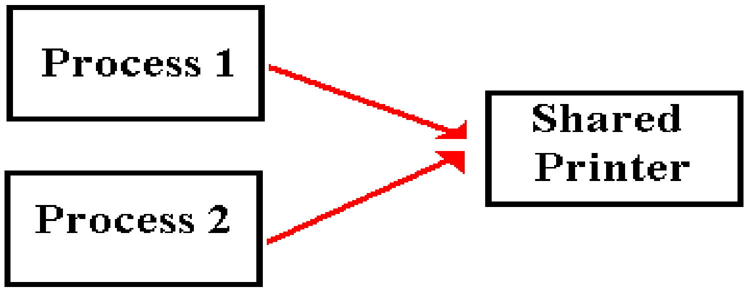


The problem here is that any program that executes with more than user privileges must have access to all system resources. This is a security vulnerability.

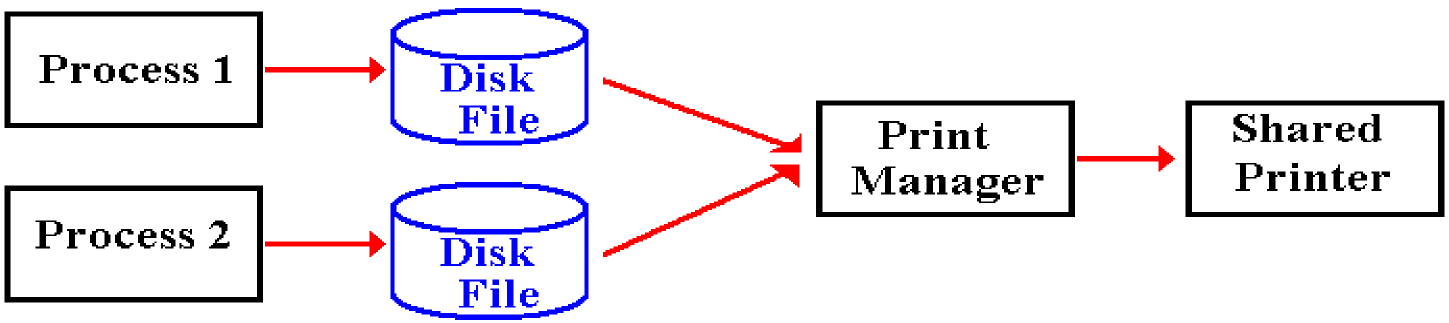
SPOOLING: A Context for Discussing O/S Services

The term “SPOOL” stands for “**S**ystem **P**eripheral **O**peration **O**n–**L**ine”.

Direct access to a shared output device can cause chaos.



The Spooling approach calls for all output to be to temporary files. The print manager has sole control of the shared printer and prints in order of closing the files.



Privileges: User vs. SPOOL

A **User Program** must be able to create a file and write data to that file.
It can read files in that user's directory, but usually not in other user's directory.
It cannot access the shared printer directly.

The **Print Manager** must be able to:

Read a temporary file in any user's directory and delete that file when done.
Access a printer directly and output directly to that device.
It should not be able to create new files in any directory.

The Print Manager cannot be run with Level 3 (Application) privilege, as that would disallow direct access to the printer and read access to the user's temporary files.

Under current designs, the Print Manager must be run with "Superuser Privileges", which include the ability to create and delete user accounts, manage memory, etc.

This violates the principle of least privilege, which states that an executing program should be given no more privileges than necessary to do its job.

We need at least four fully–implemented rings of privilege, as well as specific role restrictions within a privilege level.

Memory Segmentation

Memory paging divides the address space into a number of equal sized blocks, called **pages**. The page sizes are fixed for convenience of addressing.

Memory segmentation divides the program's address space into **logical segments**, into which logically related units are placed. As examples, we conventionally have code segments, data segments, stack segments, constant pool segments, etc.

Each segment has a **unique logical name**. All accesses to data in a segment must be through a <*name*, *offset*> pair that explicitly references the segment name.

For addressing convenience, segments are usually constrained to contain an integral number of memory pages, so that the more efficient paging can be used.

Memory segmentation facilitates the use of security techniques for protection.

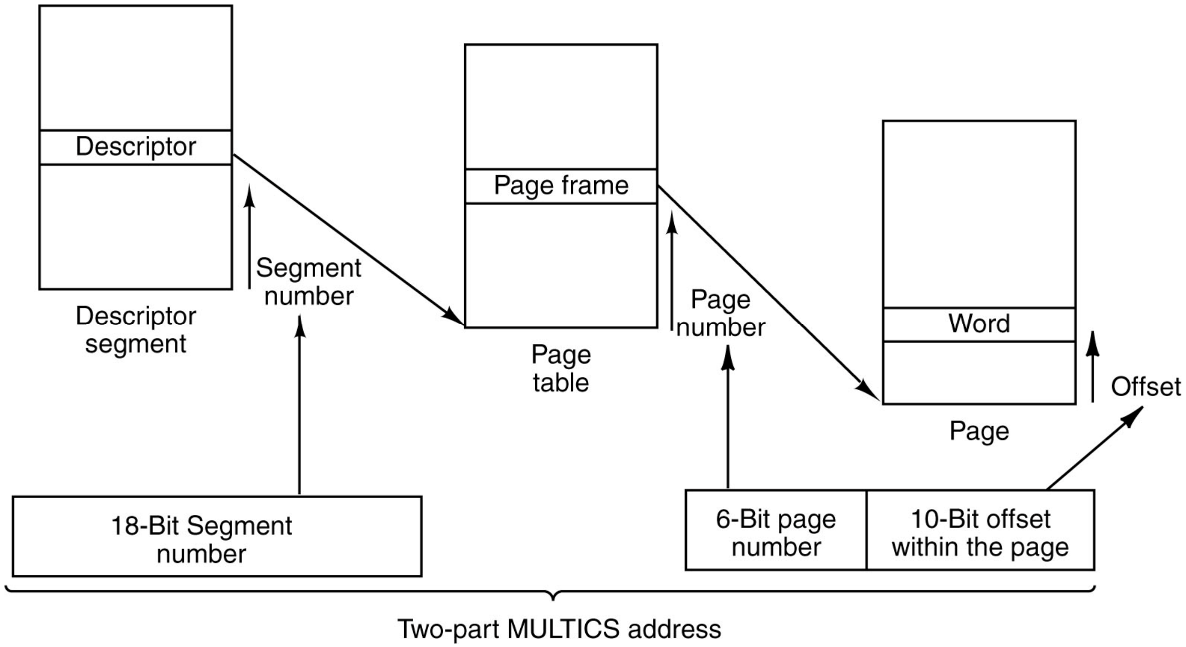
All data requiring a given level of protection can be grouped into a single segment, with protection flags specific to giving that exact level of protection.

All code requiring protection can be placed into a code segment and also protected.

It is not likely that a given segment will contain both code and data. For this reason, we may have a number of distinct segments with identical protection.

Segmentation and Its Support for Security

The segmentation scheme used for the MULTICS operating system is typical.



Each segment has a number of pages, as indicated by the page table associated with the segment. The segment can have a number of associated security descriptors.

Modern operating systems treat a segment as one of a number of general objects, each with its **A**ccess **C**ontrol **L**ist (ACL) that specifies which processes can access it.

More on Memory Protection

There are two general protections that must be provided for memory.

Protection against unauthorized access by software can be provided by a segmentation scheme, similar to that described above.

Each memory access must go through each of the segment table and its associated page table in order to generate the physical memory address.

Direct Memory Access (DMA) provides another threat to memory.

In DMA, an input device (such as a disk controller) can access memory directly without the intervention of the CPU. It can issue physical addresses to the MAR and write directly to the MBR. This allows for efficient Input / Output operations.

Unfortunately, a corrupted device controller can write directly to memory not associated with its application. We must protect memory against unauthorized DMA.

Some recent proposals for secure systems provide for a **"NoDMA Table"** that can be used to limit DMA access to specific areas of physical memory.

Securing Input and Output

Suppose that we have secured computing. How can we insure that our input and output are secure against attacks such as key logging and screen scraping?

Input Sequence.
Suppose that we want to input an “A”. We press the shift key and then the “A” key.

The keyboard sends four scan codes to the keyboard handler, operating in user mode. This might be 0x36, 0x1E, 0x9E, 0xB6 – for pressing the Shift Key, then pressing the “A” key, then releasing the “A” key, then releasing the Shift Key.

This sequence is then translated to the ASCII code 0x41, which is sent to the O/S.

Either the scan codes or the ASCII code can be intercepted by a key logger.

Output Sequence.
When the program outputs data, it is sent to the display buffer.

The display buffer represents the bit maps to be displayed on the screen. While it does not directly contain ASCII data (just its “pictures”), it does contain an image that can be copied and interpreted. This is called “screen scraping”.

Protecting the Code under Execution

We can wrap the CPU in many layers of security so that it correctly executes the code.

How do we assure ourselves that the code being executed is the code that we want?

More specifically, how do we insure that the code being executed is what we think it is and has not been maliciously altered?

One method to validate the code prior to execution is called a **cryptographic hash**.

One common hash algorithm is called “SHA–1” for “**S**ecure **H**ash **A**lgorithm 1”.

This takes the code to be executed, represented as a string of 8–bit bytes, and produces a 20 byte (160 bit) output associated with the input.

The hardware can have another mechanism that stores what the 20–byte hash should be. The hardware loads the object code, computes its SHA–1 hash, and then compares it to the stored value. If the two values match, the code is accepted as valid.

What Is a Cryptographic Hash?

First, we begin with the definition of a **hash function**. It is a many–to–one function that produces a short binary number that characterizes a longer string of bytes.

Consider the two characters "AC" with ASCII codes 0100 0001 and 0100 0011.

One hash function would be the parity of each 8–bit number: 0 and 1 (even and odd).

Another would be the exclusive OR of the sequence of 8–bit bytes.

A		0100 0001
C		0100 0011
	$\oplus$	0000 0010

The hash function must be easy to compute for any given input.

A **cryptographic hash** function has a number of additional properties.

1. A change of any single bit in the input being processed changes the output in a very noticeable way. For the 160–bit SHA–1, it changes about 80 of the bits.
2. While it is easy to compute the SHA–1 hash for a given input, it is computationally infeasible to produce another input with the identical 20–byte hash.

Thus, if a code image has the correct hash output; it is extremely probable that it is the correct code image and not some counterfeit.

Virtual Machines and Virtual Machine Monitors

A VM (Virtual Machine) is defined as an “efficient, isolated duplicate of a real machine”.

A VMM (Virtual Machine Monitor) is the control program that implements a VM.

Basically, the VM and VMM monitor “trick” the user program into the appearance that it has access to the full instruction set and hardware of the platform, while intercepting all security sensitive instructions and implementing those, usually in microcode.

The VMM approach can provide a high level of isolation between various user programs. This is definitely a security asset.

The ISA–32 architecture defines two new instruction classes to implement a virtual machine.