CS 162 Operating Systems and Systems Programming

Professor: Anthony D. Joseph

Spring 2003

Lecture 12: Protection: Kernel and Address Spaces

12.0 Main Points:

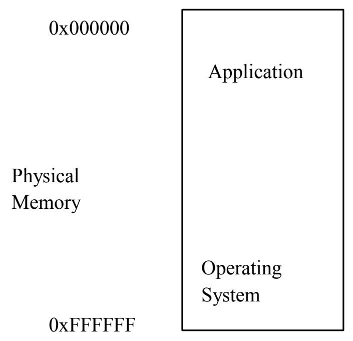
- Kernel vs. user mode
- What is an address space?
- How is it implemented?

Physical memory	Abstraction: virtual memory
No protection	Each program isolated from all others and from the OS
Limited size	Illusion of infinite memory
Sharing visible to programs	Transparent – can't tell if memory is shared
Easy to share data between programs	Ability to share code, data

12.1 Operating system organizations

12.1.1 Uniprogramming without protection

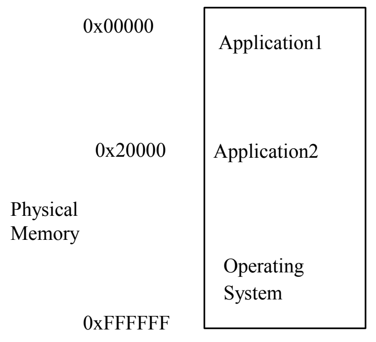
Early personal computer operating systems: application always runs at the same place in physical memory, because each application runs one at a time (application given illusion of dedicated machine, by giving it reality of a dedicated machine). For example, load application into low memory, operating system into high memory. Application can address any physical memory location.



12.1.2 Multiprogramming without protection: Linker-loader

Can multiple programs share physical memory, without hardware translation?

Yes: when a program is copied into memory, its addresses are changed (loads, stores, jumps) to use the addresses of where the program lands in memory. This is done by a **linker-loader**. Used to be very common.



UNIX **ld** does the linking portion of this (despite its name deriving from loading!): compiler generates each `.o` file with code that starts at location 0. How do you create an executable from this? Scan through each `.o`, changing addresses to point to where each module goes in larger program (requires help from compiler to say where all the relocatable addresses are stored).

12.1.3 Multiprogrammed OS with protection

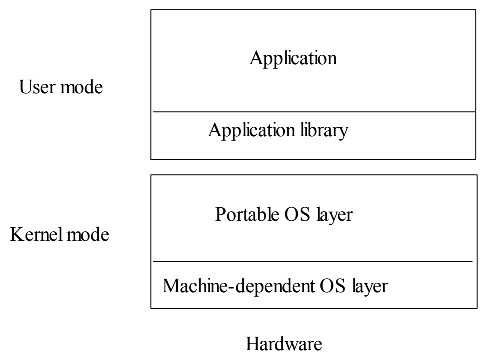
Goal of **protection**:

- Keep user programs from crashing/corrupting OS
- Keep user programs from crashing/corrupting each other

How is protection implemented?

Hardware support:

- Address translation
- Dual mode operation: kernel vs. user mode



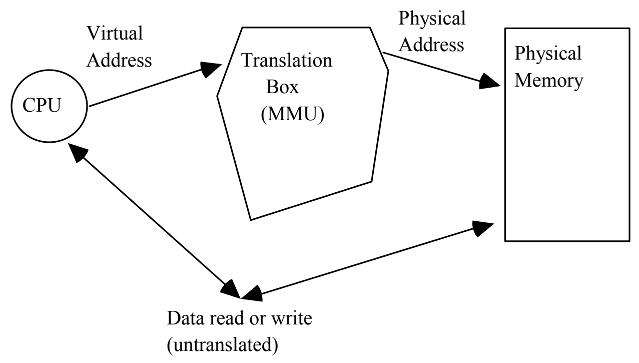
Typical Operating System Structure

12.2 Address translation

Address space: literally, all the addresses a program can touch. All the state that a program can affect or be affected by.

Restrict what a program can do by restricting what it can touch!

Hardware translates every memory reference from virtual addresses to physical addresses; software sets up and manages the mapping in the translation box.



Address Translation in Modern Architectures

Two views of memory:

- View from the CPU – what program sees, virtual memory
- View from memory – physical memory

Translation box converts between the two views.

Translation helps implement protection because there's no way for programs to even talk about other programs' addresses; no way for them to touch operating system code or data.

Translation can be implemented in any number of ways – typically, by some form of table lookup (we'll discuss various options for implementing the translation box later). Separate table for each user address space.

12.3 Dual mode operation

Can application modify its own translation tables? If it could, could get access to all of physical memory. Has to be restricted somehow.

Dual-mode operation

- When in the OS, can do anything (*kernel-mode*)
- When in a user program, restricted to only touching that program's memory (*user-mode*)

Hardware requires CPU to be in **kernel-mode** to modify address translation tables.

In Nachos, as well as most OS's:

- OS runs in kernel mode (untranslated addresses)
- User programs run in user mode (translated addresses)

Want to isolate each address space so its behavior can't do any harm, except to itself.
A couple issues:

1. How to share CPU between kernel and user programs
2. How do programs interact?
3. How does one switch between kernel and user modes when the CPU gets shared between the OS and a user program?
    - OS -> user (kernel –> user mode)
    - User -> OS (user mode –> kernel mode)

12.3.1 Kernel -> user:

To run a user program, create a thread to:

- Allocate and initialize address space control block
- Read program off disk and store in memory
- Allocate and initialize translation table (point to program memory)
- Run program (or to return to user level after calling the OS with a system call):
    - Set machine registers

    - Set hardware pointer to translation table
    - Set processor status word (from kernel mode to user mode)
    - Jump to start of program

12.3.2 User-> kernel:

How does the user program get back into the kernel?

Voluntarily user->kernel: **System call** – special instruction to jump to a specific operating system handler. Just like doing a procedure call into the operating system kernel – program asks OS kernel, please do something on procedure's behalf.

Can the user program call any routine in the OS? No. Just specific ones the OS says are OK. Always start running handler at same place, otherwise, problems!

How does OS know that system call arguments are as expected? It can't – OS kernel has to check all arguments – otherwise, bug in user program can crash kernel.

Involuntarily user->kernel: **Hardware interrupt**, also **program exception**

Examples of program exceptions:

- Bus error (bad address – e.g., unaligned access)
- Segmentation fault (out of range address)
- Page fault (important for providing illusion of infinite memory)

On system call, interrupt, or exception: hardware atomically

- Sets processor status to kernel mode
- Changes execution stack to an OS kernel stack
- Saves current program counter
- Jumps to handler routine in OS kernel
- Handler saves previous state of any registers it uses

Context switching between programs: same as with threads, except now also save and restore pointer to translation table. To resume a program, re-load registers, change PSL (hardware pointer to translation table), and jump to old PC.

How does the system call pass arguments? Two choices:

a. Use registers.
   Can't pass very much that way.
b. Write into user memory, kernel copies into its memory.
   Except:
   - User addresses – translated
   - Kernel addresses – untranslated

Addresses the kernel sees are not the same addresses as what the user sees! We'll explore this riddle in a later lecture.

12.3.3 Communication between address spaces

How do two address spaces communicate? Can't do it directly if address spaces don't share memory.

Instead, all inter-address space (in UNIX, inter-process) communication has to go through kernel, via system calls.

Models of inter-address space communication:

- Byte stream producer/consumer. For example, communicate through pipes connecting `stdin`/`stdout`.

- Message passing (send/receive). Will explain later how you can use this to build remote procedure call (RPC) abstraction, so that you can have one program call a procedure in another.

- File system (read and write files). File system is shared state! (Even though it exists outside of any address space.)

- "Shared Memory" -- Alternately, on most UNIXes, can ask kernel to set up address spaces to share a region of memory, but that violates the whole notion of why we have address spaces – to protect each program from bugs in the other programs.

In any of these, once you allow communication, bugs from one program can propagate to those it communicates with, unless each program verifies that its input is as expected.

So why do UNIXes support shared memory? One reason is that it provides a cheap way to simulate threads on systems that don't support them:

Each UNIX process = Heavyweight thread.

12.4 An Example of Application – Kernel Interaction: Shells and UNIX fork

Shell – user program (not part of the kernel!)

- Prompts users to type command
- Does system call to run command

In Nachos, system call to run command is simply "exec". But UNIX works a bit differently than Nachos.

UNIX idea: separate notion of fork vs. exec

- Fork – create a new process, exact copy of current one
- Exec – change current process to run different program

To run a program in UNIX:

- Fork a process
- In child, exec program
- In parent, wait for child to finish

UNIX fork:

- Stop current process
- Create exact copy
- Put on ready list
- Resume original

Original has code/data/stack. Copy has exactly the same thing!

Only difference between child and parent is: UNIX changes one register in child before resume.

Child process:

Exec program:

- Stop process
- Copy new program over current one
- Resume at location 0

Justification was to allow I/O (pipes, redirection, etc.), to be set up between fork and exec. Child can access shell's data structures to see whether there is any I/O redirection, and then sets it up before exec.

Nachos simply combines UNIX fork and exec into one operation.

12.5 Protection without hardware support

Does protection require hardware support? In other words, do we really need hardware address translation and an unprivileged user mode?

No! Can put two different programs in the same hardware address space, and be guaranteed that they can't trash each other's code or data.

Two approaches: strong typing and software fault isolation.

12.5.1 Protection via strong typing

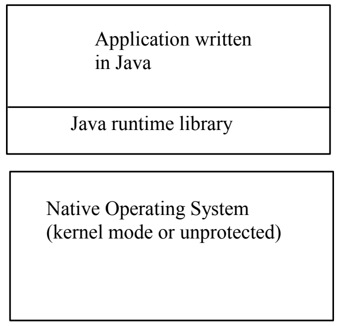
Restrict programming language to make it impossible to misuse data structures, so can't express program that would trash another program, even in same address space.

Examples of strongly typed languages include LISP, Cedar, Ada, Modula-3, and most recently, Java.

Note: nothing prevents shared data from being trashed; *which includes the data that exists in the file system*.

Even in UNIX, there is nothing to keep programs you run from deleting all your files (but at least can't crash the OS!)

Java's solution: programs written in Java can be downloaded and run safely, because language/compiler/runtime prevents the program (also called an applet) from doing anything bad (for example, can't make system calls, so can't touch files).


Application written in Java
Java runtime library
Native Operating System (kernel mode or unprotected)

Java Operating System Structure

Java also defines portable virtual machine layer, so any Java programs can run anywhere, dynamically compiled onto native machine.

Problem: requires everyone to learn new language. Any code not in Java can't be safely downloaded.

12.5.2 Protection via software fault isolation

Language independent approach: Have compiler generate object code that provably **can't** step out of bounds – programming language independent.

Easy for compiler to statically check that program doesn't do any native system calls.

How does the compiler prevent a pointer from being misused, or a jump to an arbitrary place in the (unprotected) OS?

Insert code before each "store" and "indirect branch" instruction; check that address is in bounds.

For example:

```
store r2, (r1)
```

becomes

```
assert "safe" is a legal address
copy r1 into "safe"
check safe is still legal
store r2, (safe)
```

Note that I need to handle case where malicious user inserts a jump past the check; "safe" always holds a legal address, malicious user can't generate illegal address by jumping past check.
Key to good performance is to apply aggressive compiler optimizations to remove as many checks as possible statically. Research result is protection can be provided in language independent way for < 5% overhead.

12.5.3 Example applications of software protection

Safe downloading of programs onto local machine over Web: games, interactive advertisements, etc.

Safe anonymous remote execution over Web: Web server could provide not only data, but also **computing**.

Plug-ins: Complex application built by multiple vendors (example: Netscape support for new document formats). Need to isolate failures in plug-in code from killing main application, but slow to put each piece in separate address space.

Kernel plug-ins. Drop application-specific code into OS kernel, to customize its behavior (ex: to use a CPU scheduler tuned for database needs, or CAD needs, etc.)