Memory Management

Arvind Krishnamurthy
Spring 2004

Memory Management

To support multiprogramming, we need “Protection”

- How to implement an address space?

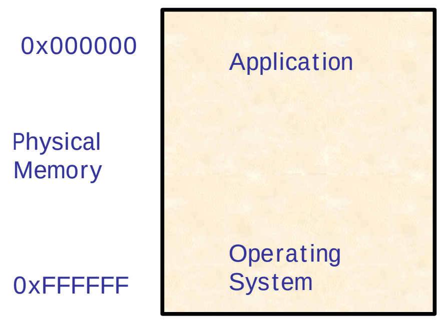
Physical memory	**Abstraction: virtual memory**
No protection	Each program isolated from all others and from the OS
Limited size	Illusion of infinite memory
Sharing visible to programs	Transparent --- can’t tell if memory is shared
Easy to share data between programs	Controlled sharing of data

OS Evolutionary Path

- Uniprogramming without protection
- Multiprogramming without protection: linker-loader
- Multiprogrammed OS with protection:
  - hardware-based approach
    - address translation (support of address space)
    - dual mode operation: kernel vs. user mode
  - software-based approach
    - type-safe languages
    - software fault isolation

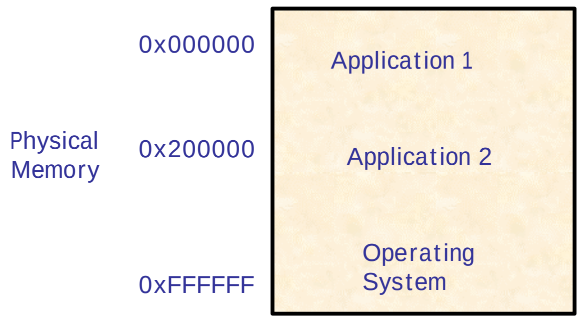
Uniprogramming (no protection)

- Application runs at the same place in physical memory
  - load application into low memory
  - load OS into high memory
  - application can address any physical memory location
  - application can corrupt OS and even the disk



Multiprogramming

- Multiple programs share physical memory
  - when copying a program into memory, use the **linker-loader** to change the addresses for all load/store/jump instructions



Linker-loader Approach

- How the linker-loader works?
  - Compiler generates .o file with code starting at location 0.
    - Also record all the re-locatable addresses
  - Linker (ld in Unix) scans through each .o, changing addresses to point to where each module goes in larger program
  - Loader loads the executable (a.out) to the memory and program runs

Linker-Loader example

```
int y;
extern int z;

int foo() {
      int x;

      bar();
      y = 1;
      z = 1;
}
```
[foo.c]

```
int z;


void bar() {
      z = 99;
}

main() {
      foo();
}
```
[bar.c]

Link: Need to patch references to "bar", "foo", and "z"

Load: Need to relocate all addresses based on what programs are running

- Problem of linker-loader --- still no protection: bugs in any program can cause other programs to crash, even OS
- Goal: how to support protection?

Incorporating Protection

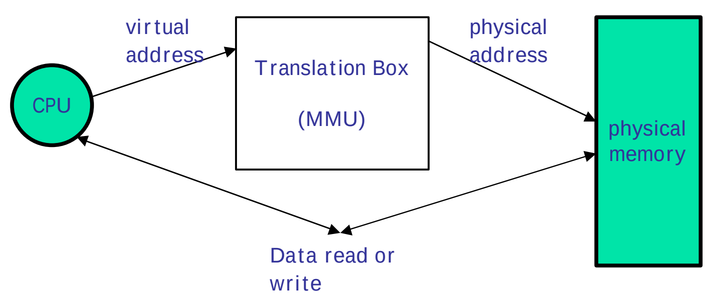
- Goal of protection
  - keep user programs from crashing/corrupting OS
  - keep user programs from crashing/corrupting each other

How is protection implemented?

- Almost all OS today use hardware-based approach
  - address translation
  - dual mode operation: kernel vs. user mode
- Other approaches: software-based solutions
  - type safe languages
  - software fault isolation

Address translation (1)

- **Address space**: state of an active program
  - Hardware translates every memory reference from virtual addresses to physical addresses
  - Software sets up and manages the mapping in the translation box



- Protection: there is no way for programs to talk about other program's addresses

Why dual mode operation ?

- If application can modify its own translation tables --- then it can access all of physical memory --- protection is lost!

- Solution: use "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)***

  HW can require CPU to be in kernel-mode to modify address translation table

- In Nachos (as well as most OS's):
  - OS runs in kernel mode (untranslated addresses)
  - User programs run in user mode (translated addresses)

- How does one switch between kernel and user modes?

Kernel → user

- The most basic of kernel-to-user transitions occur when a new user program is started by the system
- In a traditional OS, what steps are involved in starting a new user program?

User → kernel

How does the user program get back into the kernel ?

- **Hardware interrupt** (involuntarily)
  - timer interrupt, IO interrupt, etc.
- **Program exception** (involuntarily)
  - bus error (bad address --- e.g., unaligned access)
  - segmentation fault (out of range address)
  - page fault (important for providing illusion of infinite memory)
- **System call** (voluntarily)
  - special instruction to jump to a specific OS handler – just like doing a procedure call into the OS kernel
    - on MIPS, it is called "**OP_SYSCALL**"

Issues with system call

- Can the user program call any routine in the OS ?
  - No. Only the specific ones that the OS says are ok.
- How to pass arguments on a system call?
  - via registers
  - write data into user memory, kernel copies into its memory

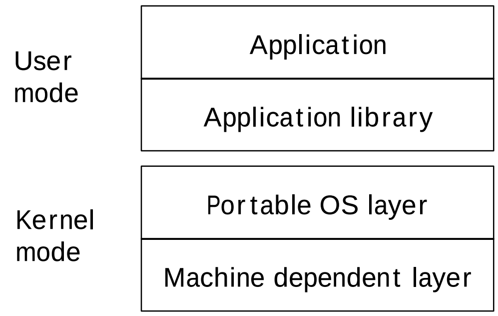
    except: user addresses --- translated

    kernel addresses --- untranslated
  - **main problem**: addresses the kernel sees are not the same addresses as what the user sees
- What if user programs does a system call with bad arguments? OS must check everything

User → kernel: how to switch

- On system call, interrupt exception and the system
  - 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 register it uses
- Context switches between programs:
  - same as with threads, except
  - also save and restore pointer to translation table
  - to resume a program: reload registers, change PSW, and jump to old PC.

OS Structure



- How does Nachos's structure fit into this model?
  - Nachos is the portable OS layer – it simulates the hardware and machine-dependent layer, and it simulates the execution of user programs running on top
    - Can still use debugger, printf, etc.
    - Can run normal UNIX programs concurrently with Nachos
  - Could run Nachos on real hardware by writing a machine-dependent layer