Memory Protection: Kernel and User Address Spaces

Sarah Diesburg

Operating Systems

CS 3430

Up to This Point

- Threads provide the illusion of an infinite number of CPUs
  - On a single processor machine
- Memory management provides a different set of illusions
  - Protected memory
  - Infinite amount of memory
  - Transparent sharing

Physical vs. Virtual Memory

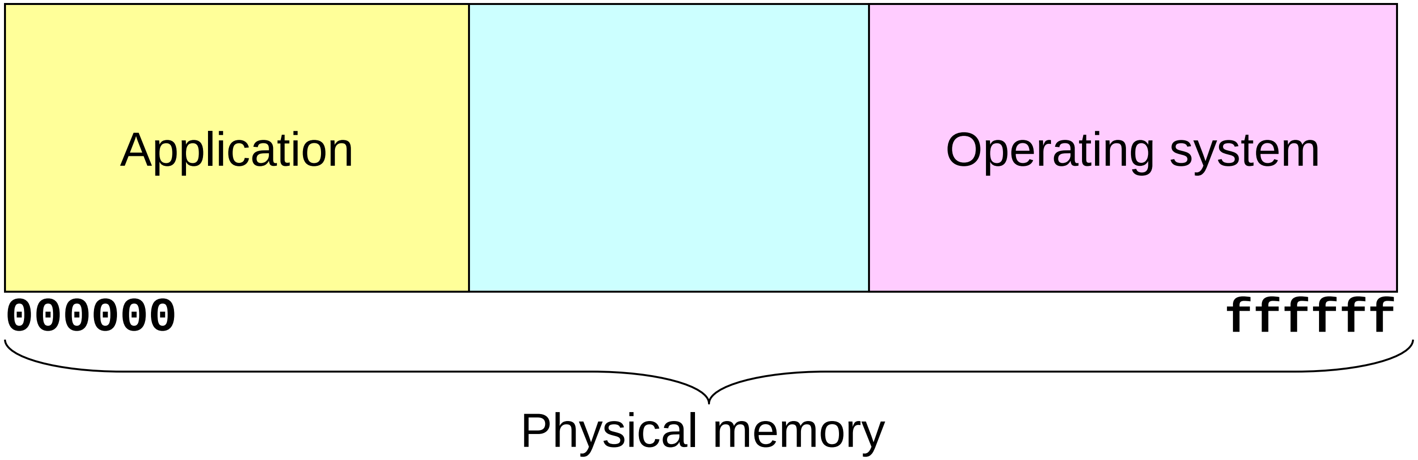
Physical memory	Virtual memory
No protection	Each process isolated from all others and from the OS
Limited size	Illusion of infinite memory
Sharing visible to processes	Each process cannot tell if memory is shared

Memory Organizations

- Simplest: ***uniprogramming without memory protection***
  - Each application runs within a hardwired range of physical memory addresses
- One application runs at a time
  - Application can use the same physical addresses every time, across reboots

Uniprogramming Without Memory Protection

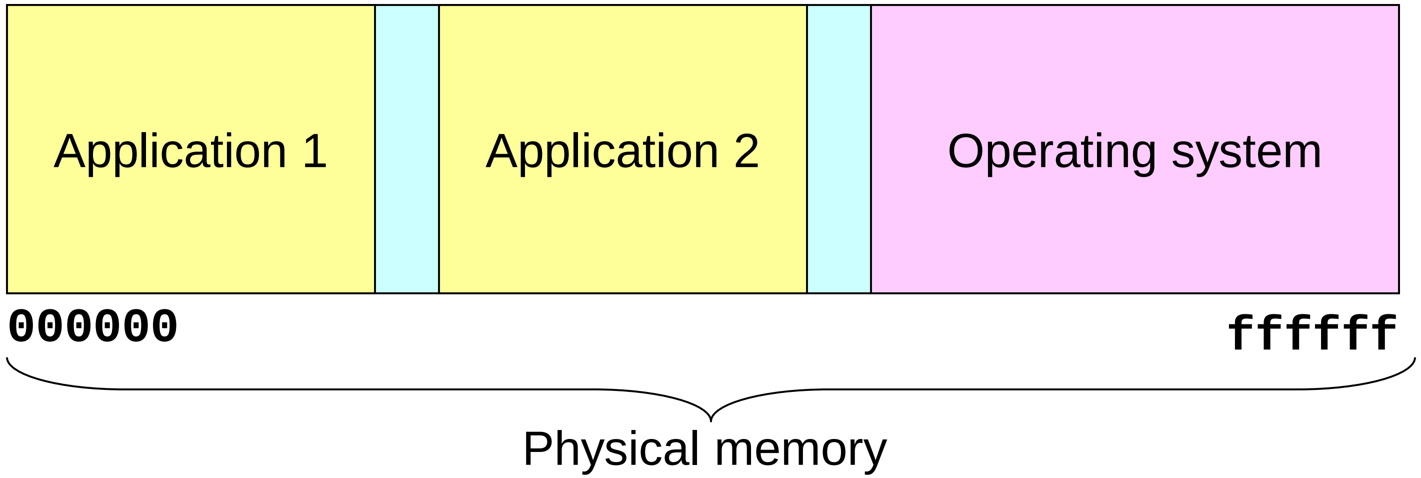
- Applications typically use the lower memory addresses
- An OS uses the higher memory addresses
- An application can address any physical memory location



Multiprogramming Without Memory Protection

- When a program is copied into memory, a ***linker-loader*** alters the code of the program (e.g., loads, stores, and jumps)
  - To use the address of where the program lands in memory

- Bugs in any program can cause other programs to crash, even the OS



Linker-loaders today

- Getting a C program to run (steps)
  1. Compile
  2. Link to shared libraries
  3. Loader puts program into memory when program is run
- Only one version of each shared library is in memory (like function defined in stdio.h)
  - But many programs can link to them!

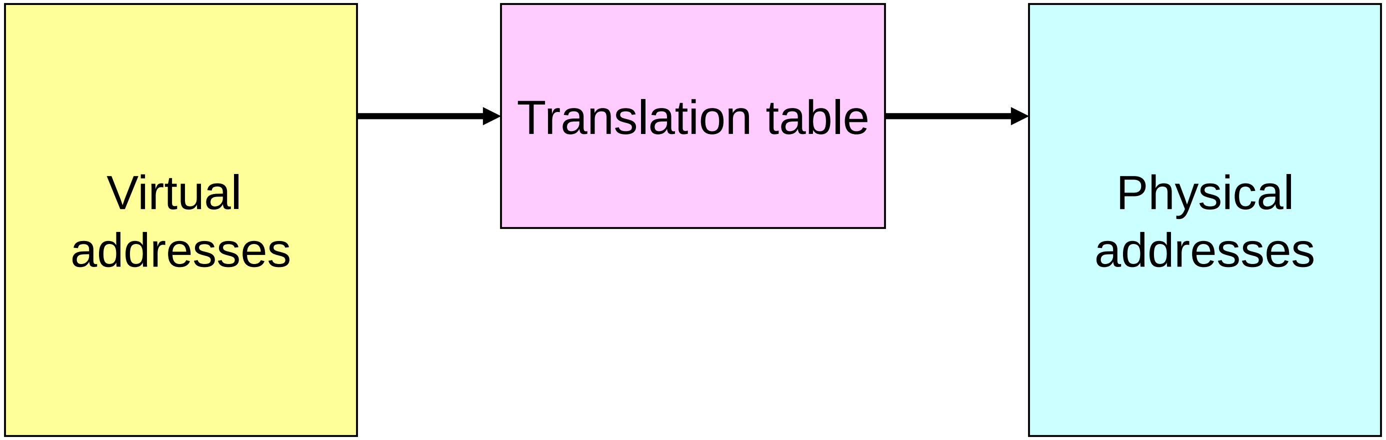
Multiprogrammed OS With Memory Protection

- ***Memory protection*** keeps user programs from crashing one another and the OS
- Two hardware-supported mechanisms
  - Address translation
  - Dual-mode operation

Address Translation

- Recall that each process is associated with an ***address space***, or all the *physical* addresses a process can touch
- However, each process believes that it owns the entire memory, starting with the *virtual* address 0
- The missing piece is a translation table to translate every memory reference from virtual to physical addresses

Address Translation Visualized



More on Address Translations

- Translation provides protection
  - Processes cannot talk about other processes' addresses, nor about the OS addresses
  - OS uses physical addresses directly
    - No translations

Dual-Mode Operation Revisited

- Translation tables offer protection if they cannot be altered by applications
- An application can only touch its address space under the user mode
- Hardware requires the CPU to be in the kernel mode to modify the address translation tables

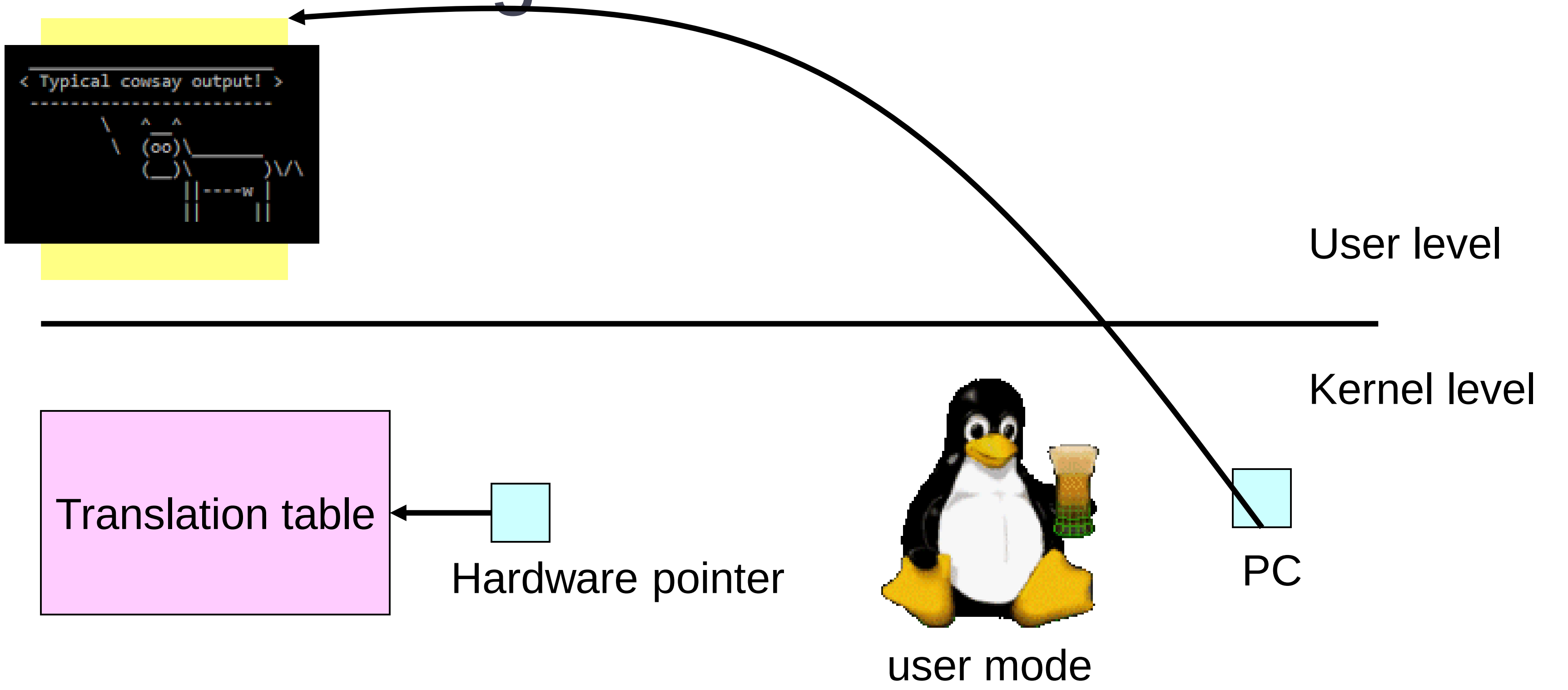
Details of Dual-Mode Operations

- How the CPU is shared between the kernel and user processes
- How processes interact among themselves

Switching from the Kernel to User Mode

- To run a user program, the kernel
  - Creates a process and initialize the address space
  - Loads the program into the memory
  - Initializes translation tables
  - Sets the hardware pointer to the translation table
  - Sets the CPU to user mode
  - Jumps to the entry point of the program

To Run a Program



Switching from User Mode to Kernel Mode

- Voluntary
  - ***System calls***: a user process asks the OS to do something on the process's behalf
- Involuntary
  - Hardware interrupts (e.g., I/O)
  - Program exceptions (e.g., segmentation fault)

Switching from User Mode to Kernel Mode

- For all cases, hardware atomically performs the following steps
  - Sets the CPU to kernel mode
  - Saves the current program counter
  - Jumps to the handler in the kernel
    - The handler saves old register values

Switching from User Mode to Kernel Mode

- Unlike context switching among threads, to switch among processes
  - Need to save and restore pointers to translation tables
- To resume process execution
  - Kernel reloads old register values
  - Sets CPU to user mode
  - Jumps to the old program counter

Communication Between Address Spaces

- Processes communicate among address spaces via ***interprocess communication (IPC)***
  - Byte stream (e.g., **`pipe`**)
  - Message passing (send/receive)
  - File system (e.g., read and write files)
  - Shared memory
- Bugs can propagate from one process to another

Protection Without Hardware Support

- Hardware-supported protection can be slow
  - Requires applications be separated into address spaces to achieve fault isolation
- What if your applications are built by multiple vendors? (e.g., Firefox plug-ins)
  - Can we run two programs in the same address space, with safety guarantees?

Protection via Strong Typing

- Programming languages may disallow the misuse of data structures (casting)
  - e.g., LISP and Java
- Java has its own virtual machines
  - A Java program can run on different hardware and OSes

- Need to learn a new language

Protection via Software Fault Isolation

- Compilers generate code that is provably safe
  - e.g., a pointer cannot reference illegal addresses
- With aggressive optimizations, the overhead can be as low as 5%

Protection via Software Fault Isolation

Original instruction	Compiler-modified version
`st r2, (r1)`	**`safe`** = a legal address
	`safe = r1`
	Check **`safe`** is still legal
	`st r2, (safe)`

- A malicious user cannot jump to the last line and do damage, since **`safe`** is a legal address