Lecture 13

<2016-05-25 Wed>

Contents

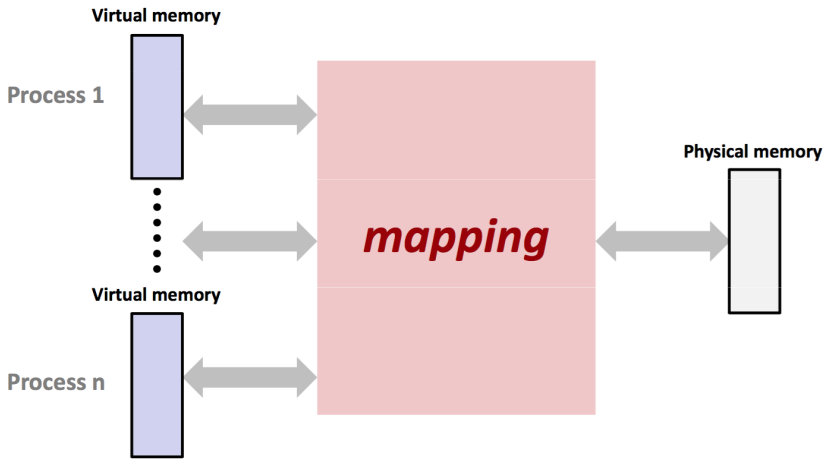
1 Operating System

1.1 Virtual Memory, Page

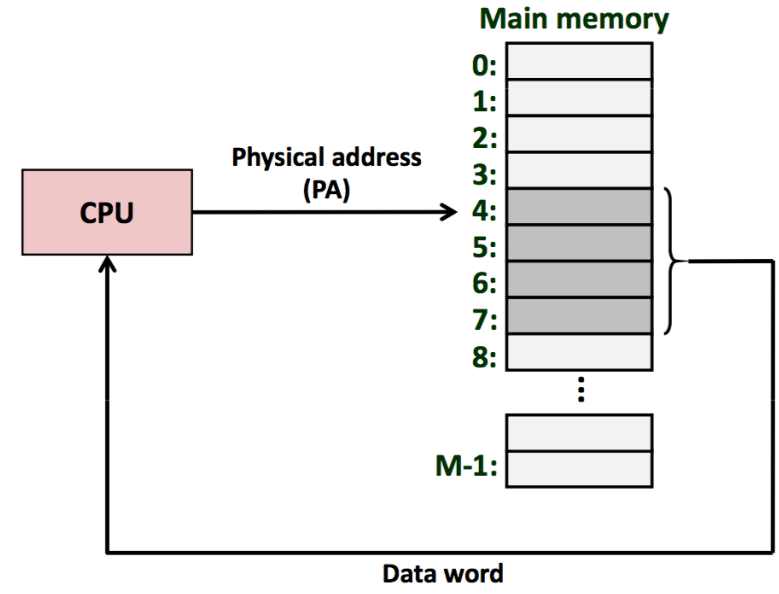
- "programs" refer to virtual memory address
  - `movl (%ecx), %eax`
  - conceptually very large array of bytes

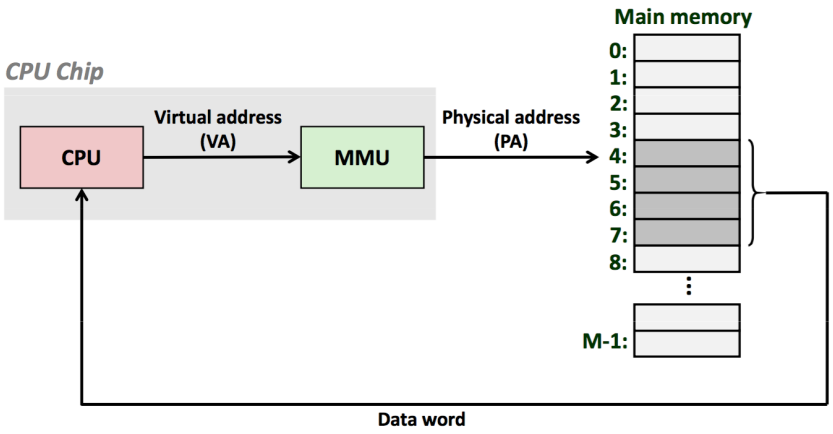
  - each bytes have its own address
  - actually implemented with hierarchy of different memories
  - system provides address space private to particular "program"
- allocation: compiler and run-time system
  - where different program objects should be stored
  - add allocation within single virtual address space
- why virtual memory
  - address does not fit
    - 64-bit address space(16 exabyte) » physical main memory(few gigabytes)
  - memory management
  - how to protect
    - how to prevent different processes from using the same physical memory
  - how to share

1.1.1 Solution: Virtual Addressing



- each process gets its own private memory space
- solves the previous problems





1.1.2 Address Space

- virtual address space
  - set of 2^n virtual addresses
- physical address space
  - set of 2^m physical addresses

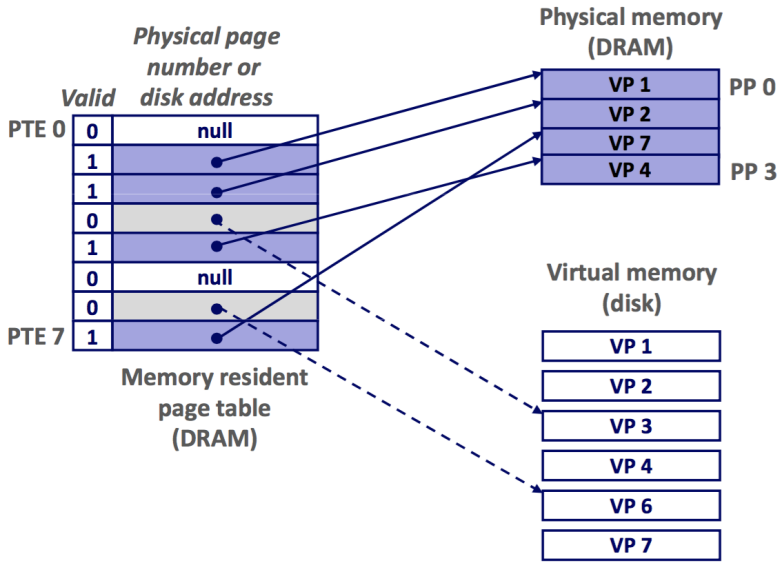
- clean distinction between data (bytes) and their attributes (addresses)
- each object can now have multiple addresses
- every byte in main memory
  - 1 physical address
  - 1 (or more) virtual addresses

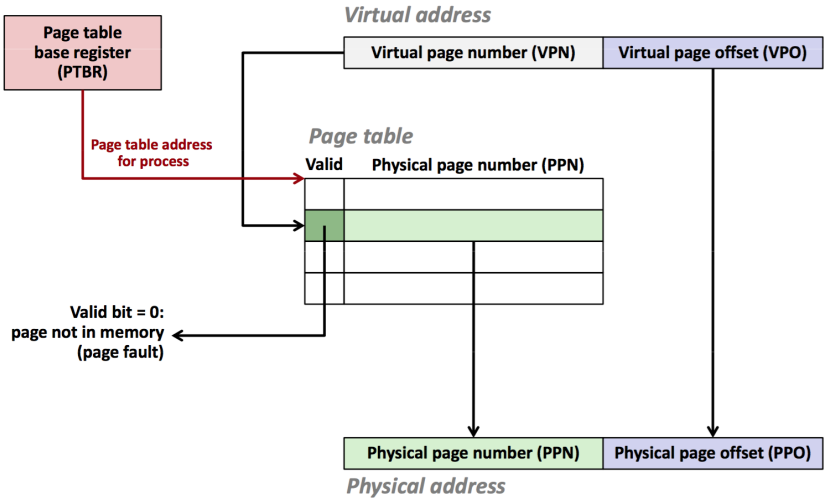
1.1.3 Why Virtual Memory

- efficient use of limited main memory (RAM)
  - use RAM as cache for the parts of a virtual address space
    - some non-cache parts stored on disk
    - some (unallocated) non-cache parts stored nowhere
  - keep only active of areas of virtual address space in memory
    - transfer data back and forth as needed
- simplifies memory management for programmers
  - each program gets the same full, private linear address space
- isolates address space
  - one process can't interfere with another's memory
    - because they operate in different address space
  - user process can't access privileged information
    - different sections of address spaces have different permissions

1.1.4 Address Translation: Page Table

- a page table is an array of page table entries (PTEs) that maps virtual pages to physical pages





1.2 Exception

1.2.1 Altering Control Flow

- control flow
  - a CPU simply reads and executes a sequence of instructions
  - this sequence is the system's physical control flow

- alter the control flow: 2 mechanisms
  - jumps and branches
  - call and return using the stack discipline
  - both react to changes in program state
  - insufficient for a useful system
    - difficult for the CPU to react to changes in system state
      - data arrive from a disk or network adapter
      - instructions divides by 0
      - user hits control-c at the keyboard
      - system timer expires
- needs exceptional control flow

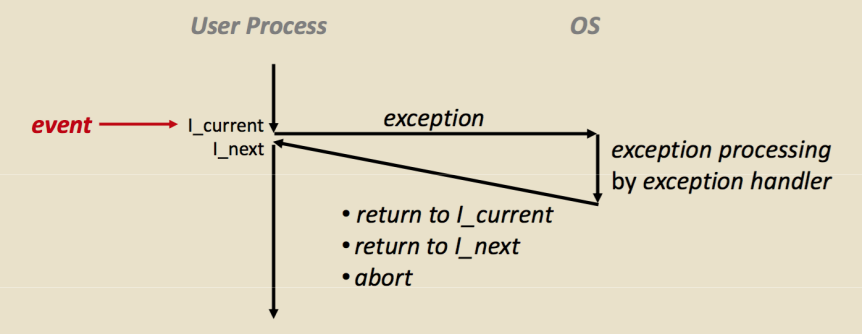
1.2.2 Exceptional Control Flow

- mechanisms for exceptional control flow exists at all level of a computer system
- low level mechanism
  - exceptions
    - change in control flow in response to system event
    - combination of hardware and OS software
- high level mechanisms
  - process context switch
  - signals
  - nonlocal jumps
  - implemented by either
    - OS software: context switch and signals
    - C language runtime library: nonlocal jump

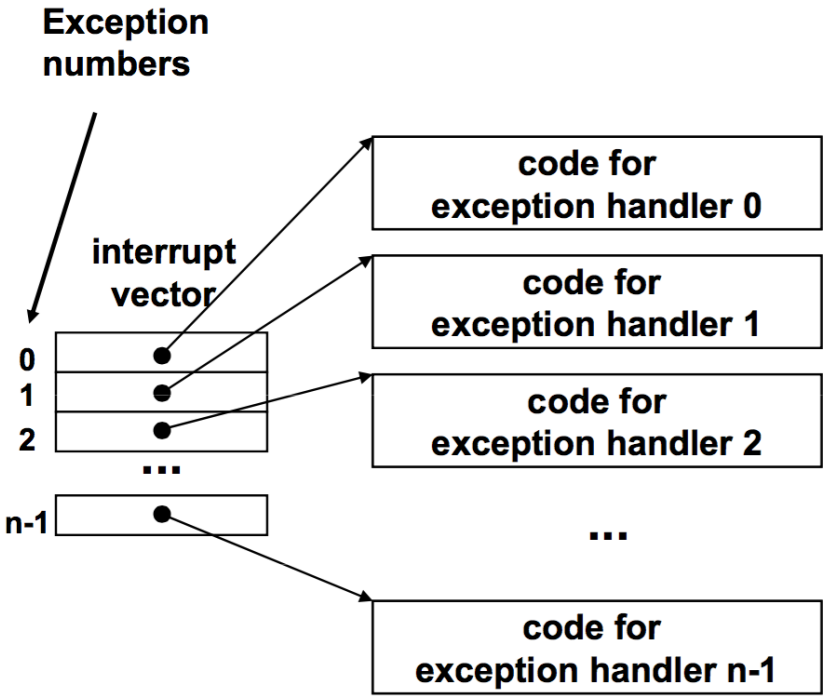
1.2.3 Exceptions

- an exception is a transfer of control to the OS in response to some event
- examples

- divide by 0
- arithmetic overflow
- page fault
- I/O request conpletes
- Ctrl-C



1.2.4 Interrupt Vectors



- each type of event has a unique exception number k
  - k as an index into jump table (interrupt vector)
- jump table entry k points to an exception handler
  - handler k is called each time exception k occurs

1.2.5 Asynchronous/Synchronous Exception

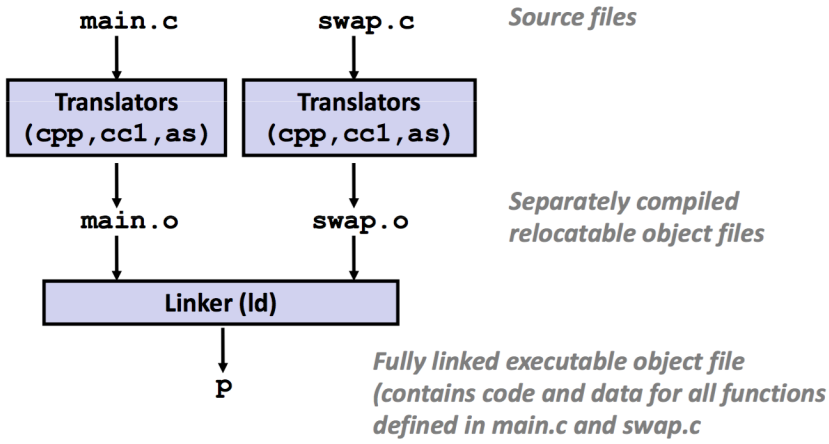
- asynchronous
  - caused by events external to the processor
    - indicated by setting the processor's interrupt pin
    - handler returns to next instruction
  - examples
    - I/O interrupts
      - hitting Ctrl-C at the keyboard
      - arrival of a packet from a network
      - arrival of a data sector from a disk
    - hard reset interrupt
      - hitting the reset button
    - soft rest interrupt
      - hitting Ctrl-Alt-Delete on a PC
- synchronous
  - caused by events that occur as a result of executing an instruction
    - traps
      - intentional
      - example: system calls, breakpoint traps, special instructions
      - returns control to next instruction
    - faults
      - unintentional but possibly recoverable
      - example: page fault (recoverable), protection faults (unrecoverable)
      - either re-executes faulting ("current") instruction or aborts
    - aborts

- unintentional and unrecoverable
- example: parity error, machine check
- aborts current program

1. Read A Disk Sector (asynchronous)
2. Trap Example (synchronous)
   - open a file
     - user call `open` (system call)
     - OS must find or create file, get it ready for reading or writing
     - returns integer file descriptor
3. Fault Example (synchronous)
   - memory reference

1.3 Linking

1.3.1 Static Linking



- programs are translated and linked using a compiler driver

1.3.2 Linker

- modularity

  - program can be written as a collection of smaller source files, rather than one monolithic mass
  - can build libraries of common functions
    * e.g., math library, standard C library

- efficiency
  - time: separate compilation
    * change one source file, compile, and then relink
    * no need to recompile other source files
  - space: libraries
    * common functions can be aggregated into a single file
    * yet executable files and running memory images contain only code for the functions they actually use

1.3.3 Three Kinds of Object Files

- relocatable object file (`.o`)
  - contain code and data in a form that can be combined with other relocatable object files to form executable obejct files
  - each `.o` file is produced from exactly one source file
- executable object file
  - contains code and data in a form that can be copied directly into memory and then executed
- shared object file (`.so`)
  - special type of relocatable object file that can be loaded into memory and linked dynamically, at either load-time or run-time
  - called dynamic link libraries (dll) by windows

1.3.4 Static Libraries (`.a` archives) vs. Shared Libraries (`.so`)

1. Static Libraries
   - concatenate related relocated object files into a single file with an index

- enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives
- if an archive member file resolves reference, link into executable
- disadvantage:
  - duplication in the stored executables (every function need stdlibc)
  - duplication in the running executables
  - minor bug fixes of system require each application to explicitly relink

2. Shared Libraries

   - object files that contain code and data that are loaded and linked into an application dynamically, at either load-time or run-time
   - also called: dynamic link libraries, DLLs, .so files

3. Shared Libraries and Dynamic Linking

   - dynamic linking occur when executable is first loaded and run (**load-time linking**)
     - common case for Linux, handled automatically by the dynamic linker
     - standard C library (libc.so) usually dynamically linked
   - dynamic linking can also occur after program has begun (**run-time linking**)
     - in unix, this is done by calls to the `dlopen` interface

1.3.5 Dynamic Linking