CS 537
Lecture 7
Memory

Michael Swift

1

Memory Management Topics

- Goals of memory management
  - convenient abstraction for programming
  - isolation between processes
  - allocate scarce memory resources between competing processes, maximize performance (minimize overhead)
- Mechanisms
  - physical vs. virtual address spaces
  - page table management, segmentation policies
  - page replacement policies

2

Virtual Memory from 10,000 feet

- The basic abstraction that the OS provides for memory management is virtual memory (VM)
  - VM enables programs to execute without requiring their entire address space to be resident in physical memory
    - program can also execute on machines with less RAM than it "needs"
  - many programs don't need all of their code or data at once (or ever)
    - e.g., branches they never take, or data they never read/write
    - no need to allocate memory for it, OS should adjust amount allocated based on its run-time behavior
  - virtual memory isolates processes from each other
    - one process cannot name addresses visible to others; each process has its own isolated address space
- VM requires hardware and OS support
  - MMU's, TLB's, page tables, …

3

Virtualizing Resources

- Physical Reality:
  Different Processes share the same hardware
  - Need to multiplex CPU (finished earlier: scheduling)
  - Need to multiplex use of Memory (Today)
  - Need to multiplex disk and devices (later in term)
- Why worry about memory sharing?
  - The complete working state of a process and/or kernel is defined by its data in memory (and registers)
  - Consequently, cannot just let different threads of control use the same memory
    - Physics: two different pieces of data cannot occupy the same locations in memory
  - Probably don't want different threads to even have access to each other's memory (protection)

4

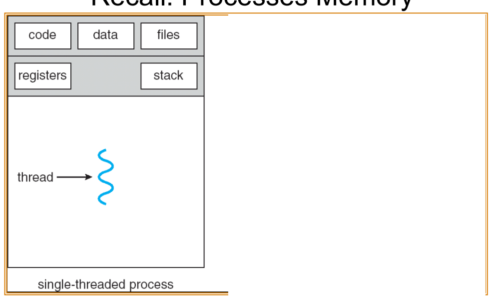
In the beginning…

- First, there was batch programming
  - programs used physical addresses directly
  - OS loads job, runs it, unloads it
- Then came multiprogramming
  - need multiple processes in memory at once
    - to overlap I/O and computation
  - memory requirements:
    - protection: restrict which addresses processes can use, so they can't stomp on each other
    - fast translation: memory lookups must be fast, in spite of protection scheme
    - fast context switching: when swap between jobs, updating memory hardware (protection and translation) must be quick

Virtual Addresses

- To make it easier to manage memory of multiple processes, make processes use virtual addresses
  - virtual addresses are independent of location in physical memory (RAM) that referenced data lives
    - OS determines location in physical memory
  - instructions issued by CPU reference virtual addresses
    - e.g., pointers, arguments to load/store instruction, PC, …
  - virtual addresses are translated by hardware into physical addresses (with some help from OS)
- The set of virtual addresses a process can reference is its address space
  - many different possible mechanisms for translating virtual addresses to physical addresses
    - we'll take a historical walk through them, ending up with our current techniques
- In reality, an address space is a **data structure** in the kernel

Recall: Processes Memory



- Threads encapsulate activity
  - "Active" component of a process
- Address spaces encapsulate protection
  - Keeps buggy program from trashing the system
  - "Passive" component of a process

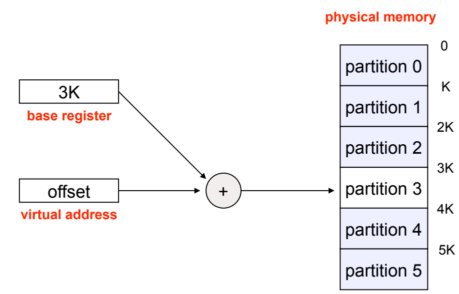
Important Aspects of Memory Multiplexing

- Translation:
  - Ability to translate accesses from one address space (virtual) to a different one (physical)
  - When translation exists, processor uses virtual addresses, physical memory uses physical addresses
  - Side effects:
    - Can be used to avoid overlap
    - Can be used to give uniform view of memory to programs
- Protection:
  - Prevent access to private memory of other processes
    - Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc).
    - Kernel data protected from User programs
    - Programs protected from themselves

Old technique #1: Fixed Partitions

- Physical memory is broken up into fixed partitions
  - all partitions are equally sized, partitioning never changes
  - hardware requirement: base register
    - physical address = virtual address + base register
    - base register loaded by OS when it switches to a process
  - how can we ensure protection?
- Advantages
  - simple, ultra-fast context switch
- Problems
  - internal fragmentation: memory in a partition not used by its owning process isn't available to other processes
  - partition size problem: no one size is appropriate for all processes
    - fragmentation vs. fitting large programs in partition

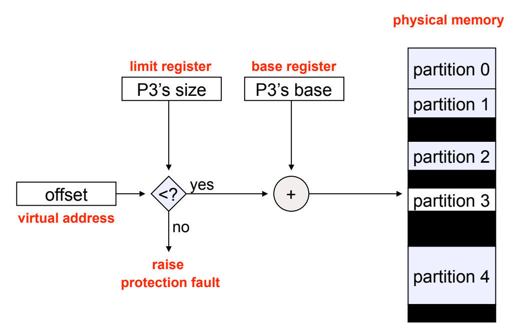
Fixed Partitions (K bytes)



Old technique #2: Variable Partitions

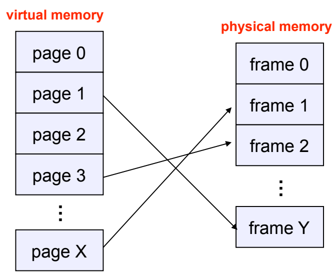
- Obvious next step: physical memory is broken up into variable-sized partitions
  - hardware requirements: base register, limit register
  - physical address = virtual address + base register
  - how do we provide protection?
    - if (physical address > base + limit) then… ?
- Advantages
  - no internal fragmentation
    - simply allocate partition size to be just big enough for process
    - (assuming we know what that is!)
- Problems
  - external fragmentation
    - as we load and unload jobs, holes are left scattered throughout physical memory

Variable Partitions



Modern technique: Paging

- Solve the external fragmentation problem by using fixed sized units in both physical and virtual memory



13

User's Perspective

- Processes view memory as a contiguous address space from bytes 0 through N
  - virtual address space (VAS)
- In reality, virtual pages are scattered across physical memory frames
  - virtual-to-physical mapping
  - this mapping is invisible to the program
- Protection is provided because a program cannot reference memory outside of it's VAS
  - the virtual address 0xDEADBEEF maps to different physical addresses for different processes

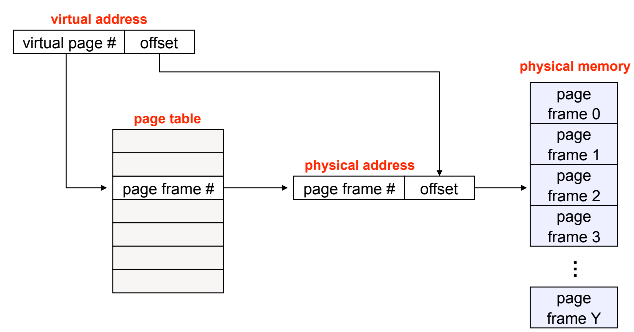
14

Paging

- Translating virtual addresses
  - a virtual address has two parts: virtual page number & offset
  - virtual page number (VPN) is index into a page table
  - page table entry contains page frame number (PFN)
  - physical address is PFN::offset
- Page tables
  - managed by the OS
  - map virtual page number (VPN) to page frame number (PFN)
    - VPN is simply an index into the page table
  - one page table entry (PTE) per page in virtual address space
    - i.e., one PTE per VPN

15

Paging



16

Paging example

- assume 32 bit addresses
  - assume page size is 4KB (4096 bytes, or 2^{12} bytes)
  - VPN is 20 bits long (2^{20} VPNs), offset is 12 bits long
- let's translate virtual address `0x13325328`
  - VPN is 0x13325, and offset is 0x328
  - assume page table entry 0x13325 contains value 0x03004
    - page frame number is 0x03004
    - VPN 0x13325 maps to PFN 0x03004
  - physical address = PFN::offset = 0x03004328

17

Page Table Entries (PTEs)

1	1	1	2	20
V	R	M	prot	page frame number

- PTE's control mapping
  - the valid bit says whether or not the PTE can be used
    - says whether or not a virtual address is valid
    - it is checked each time a virtual address is used
  - the reference bit says whether the page has been accessed
    - it is set when a page has been read or written to
  - the modify bit says whether or not the page is dirty
    - it is set when a write to the page has occurred
  - the protection bits control which operations are allowed
    - read, write, execute
  - the page frame number determines the physical page
    - physical page start address = PFN << (#bits/page)

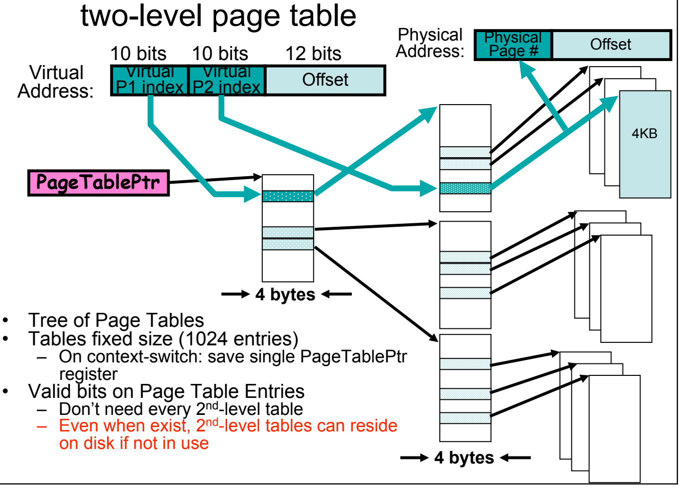
18

Multi-level Translation

- What about a tree of tables?
  - Lowest level page table⇒memory still allocated with bitmap
- Could have any number of levels
  - x86 has 2
  - x64 has 4

19

Another common example: two-level page table



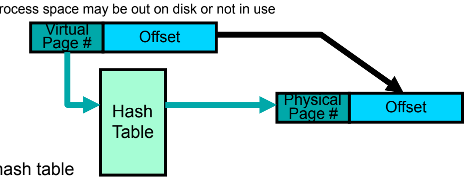
- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Valid bits on Page Table Entries
  - Don't need every 2^{nd}-level table
  - Even when exist, 2^{nd}-level tables can reside on disk if not in use

Multi-level Translation Analysis

- Pros:
  - Only need to allocate as many page table entries as we need for application
    - In other wards, sparse address spaces are easy
  - Easy memory allocation
  - Easy Sharing
    - Share at segment or page level (need additional reference counting)
- Cons:
  - One pointer per page (typically 4K – 16K pages today)
  - Two (or more, if >2 levels) lookups per reference
    - Seems very expensive!

Inverted Page Table

- With all previous examples ("Forward Page Tables")
  - Size of page table is at least as large as amount of virtual memory allocated to processes
  - Physical memory may be much less
    - Much of process space may be out on disk or not in use



- Answer: use a hash table
  - Called an "Inverted Page Table"
  - Size is independent of virtual address space
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces
- Cons: Complexity of managing hash changes
  - Often in hardware!

Paging Advantages

- Easy to allocate physical memory
  - physical memory is allocated from free list of frames
    - to allocate a frame, just remove it from its free list
  - external fragmentation is not a problem!
    - complication for kernel contiguous physical memory allocation
      - many lists, each keeps track of free regions of particular size
      - regions' sizes are multiples of page sizes
      - "buddy algorithm"
- Easy to "page out" chunks of programs
  - all chunks are the same size (page size)
  - use valid bit to detect references to "paged-out" pages
  - also, page sizes are usually chosen to be convenient multiples of disk block sizes

Paging Disadvantages

- Can still have internal fragmentation
  - process may not use memory in exact multiples of pages
- Memory reference overhead
  - 2 references per address lookup (page table, then memory)
  - solution: use a hardware cache to absorb page table lookups
    - translation lookaside buffer (TLB) – next class
- Memory required to hold page tables can be large
  - need one PTE per page in virtual address space
  - 32 bit AS with 4KB pages = 2^{20} PTEs = 1,048,576 PTEs
  - 4 bytes/PTE = 4MB per page table
    - OS's typically have separate page tables per process
    - 25 processes = 100MB of page tables
  - solution: page the page tables (!!!)
    - (ow, my brain hurts…more later)