Administrivia

- Midterm – grades available soon; Adnan should have bluebooks available this evening in office hours

- Last time: contiguous memory allocation
  - Compiling; address binding; loading; linking; dynamic loading
  - Logical address space vs. physical address space
  - Requirements for memory management – Need for hardware support
    - Fixed partitions
    - Dynamic partitions
    - Buddy system

- Today
  - Non-contiguous memory allocation

Compiling, Linking and Loading

• Problem: what to do with memory references/branches; will the program go in the same place in memory every time?

Memory Management Requirements

- Relocation: we do not know beforehand where the program will actually go in memory
- Protection: firewalls are needed to protect programs from interfering with the OS or with each other
- Sharing: the protection mechanism should be flexible enough to allow portions of the memory to be shared (instructions or data)

Logical vs. Physical Address Space

- Each process has a logical address space that is bound to a separate physical address space
  - Logical address generator by the CPU
  - Physical address address seen by the memory
- Impossible to expensive to translate at run-time using software
- Memory Management Unit translates
Contiguous Allocation

- Memory allocated in one chunk
- Fixed partitions:
  - fixed in size (internal fragmentation)
  - Limits degree of concurrency (number of processes that are active)
  - Simple to manage (translation and protection)
- Dynamic Partitions:
  - External Fragmentation
  - More flexible
  - Harder to manage
- Buddy system: allocate variable size but in powers of two – happy medium?

Paging

- Now we relax the restriction of contiguous allocation
- Idea:
  - Divide physical memory into equal size small parts called frames
  - Divide each process' logical memory into pages, each with the same size as the frames
  - To run a program of size n pages, need to find n free frames and load the program
- This is the memory management model in NachOS
- OS keeps track of free frames
- Internal Fragmentation? External?
- Frames do not have to be contiguous
- Design Issues

Size of the Frame

- What size should the frames be?
- How to translate from logical to physical address?
- Placement? Replacement?

- Paging is like fixed partitions (one size), but you allow each process to have multiple partitions
  - Again, if you were printing a single bill, which bill would you print?
- If the size is too large, we risk internal fragmentation
- If it is too small, the overhead to manage it is big
- The page size is a power of 2 (simplifies translation), usually in the range 1k to 32k (older machines used smaller pages; newer have superpages)
Address translation – Logical to Physical

- Assume the page/frame size is $2^k$; assume the virtual address is $n$ bits long
  - The high order $(n-k)$ bits within the virtual address represent the page number
  - The low order $k$ bits represent the offset inside the page

- Each process keeps a page table to keep track of the location of its pages
  - Each page in the process address space has an entry in the page table
  - An entry is a mapping between a virtual page number and a physical frame that contains it
  - Page table entry also holds any attributes of that page (e.g., shared, read only, etc.)

- Translation:
  1. From page number, find frame number in table
  2. Replace the page number with the frame number

Example

Paging Hardware

![Diagram of paging hardware]

1. From page number, find frame number in table
2. Replace the page number with the frame number
• Memory is 32-bytes, page is 4-bytes

• Free frames are a property of the physical memory; one table called the frame table can keep track of the state of the memory
  – Frame table keeps track of whether frames are full or empty
  – In NachOS you are using the Bitmap object for that

• How does paging provide protection?
• The page table for each process is managed by the OS; user cannot modify it directly
• When the OS allocate pages, it marks the entries in the page table. OS sets valid bit
• As long as the logical address is valid, the physical address will be (trap to OS if invalid page)
Discussion

- Paging is similar to fixed partitions, but allows many partitions for each process
  - Advantages: Much less internal fragmentation; no external fragmentation; will later allow us to only keep a portion of the process memory resident; easy to protect and share
  - Disadvantages: more complex to implement than contiguous allocation

- Steps for memory access:
  1. From logical address find page number
  2. Access the page table to get the corresponding frame
  3. Physical address = frame number : offset

- How does it do in terms of our 3 requirements (relocation, protection and sharing)?
- How is the page table implemented? In memory?

Page Table Organization

- What is involved in address translation?
  - From the address find the page number
  - Look in the page table to find the appropriate entry
    - how should the page table be implemented?
  - Get the physical frame number and generate the physical address

- How big is the page table? Example: assume an address space of 4 gigabytes ($2^{32}$) and a page size of 1kbyte ($2^{10}$)

- Where should the page table be kept? In hardware? In memory (real or virtual)?

- Many instructions needed for every memory reference?? This is inefficient!

Sharing Pages

- It is possible to share one or more pages (by having them appear in more than one page table)
- Do you share data or code? Why?
  - Code can be shared if it is re-entrant; it is not self-modifying
  - Does the code have to appear in the same logical pages for each process?

Making it Real

- To Recap, we have two problems:
  1. Inefficient to have several instructions, and two accesses to memory just to read the page table entry
  2. The size of the page table can be very big, how to store it?

- This is especially true when we get to virtual memory
- SUNY - BINGHAMTON – CS350 SPRING ’08 Lec. #14
Hardware Support

- Evaluate based on: access time; cost when we context switch; hardware/memory overhead required

- Implement the page table in a special register set? Good or bad?

- Page table in memory, have a page-table base register (PTBR) – better?
  - Still need two memory accesses to get the value

- Use a cache for the page table: Translation Lookaside Buffer (TLB)
  - A fully associative cache; usually 32-64 entries
  - Idea: most recently used entries will be in the TLB quick translation

- If not in TLB (TLB miss), get it from the page table in memory (using the PTBR scheme above)

TLB and Paging

- Cache organization – associative caches most expensive, but have highest hit-ratio
- If the TLB hit ratio is high enough, we almost eliminate the cost of the page-table access
  - Almost the same performance as the page-table in registers scheme but cheaper and without limitations on page table size
- TLB solves memory access time problem; we will delay solutions for page table size until we discuss virtual memory basics

TLB Management

- TLB Misses
  - Miss occurs when a reference is generated to a page whose page table entry is not in the TLB
  - Must get the page table entry from memory, place the entry in the TLB and translate
    * Software TLB: TLB miss generates a trap to the OS – OS handles the miss.
    * Hardware TLB: CPU handles the miss in hardware
  - Software TLB most common; another example of the hardware/software tradeoff

- What happens on a context switch?
  - TLB entries are no longer valid (they refer to the old processes page entries)
  - Two options: (1) clear the TLB, or (2) save/restore the TLB entries into the PCB as part of the context switch