Outline

- Announcements
  - Lab 4 due Monday, April 7th
    - Please sign up for demos on April 7th and 8th
- Memory Management
  - Paging (cont’d)
  - Segmentation
- Virtual Memory
  - Introduction
  - Demand paging

[Silberschatz/Galvin/Gagne: Sections 9.4 – 9.6, 10.1 – 10.2]

Multi-level Page Tables

- Rationale: Modern systems support a very large logical address space
  - page tables themselves become very large
    - e.g., for a system with 32-bit logical addresses and 4K pages
      - we need 2^20 page table entries (4 bytes per PTE implies 4 MB of space)
- Solution: page the page table itself
  - cost: additional memory accesses (but caching helps)
Advantages of Multi-level Page Tables

Consider a system with 32-bit logical addresses, 4 KB pages, 4-byte PTEs.

**Number of PTEs using a single-level page table?**

128 MB / 4 KB = 2^27 / 2^12 = 2^15 pages
Therefore, 2^15 PTEs, occupying 2^17 bytes of space (128 KB)

**Application process:**

- 2 MB: Text + Data
- 2 MB: Stack
- 124 MB: Virtual memory
- 128 MB: Virtual address space

**Number of PTEs using a two-level page table?**

How do we split up address bits?
One option: Minimize space for first-level table
One page (4KB) can store 4 KB / 4 byte = 2^10 PTEs

Therefore, view logical address as:

```
10 (p1) 10 (p2) 12
512 PTEs
```

Number of useful PTEs:
```
2 + 512 + 512
```

Space occupied:
```
3 pages (12 KB)
```

Page Tables and Sharing

- Page tables permit different virtual addresses (frames of different processes) to map to the same physical address
  - convenient sharing of common code (dynamically-linked system libraries)
  - shared data segments for IPC

```
Logical Memory (Pages)  Physical Memory (Frames)
```

```
0 code 0
1 code 1
2 code 2
3 data p1
4 code 0
5 code 1
6 code 2
7 data p2
```

Inverted Page Tables

- Observation
  - usually, only a portion of all the pages from the system's memory can be stored in the physical memory
  - so while the required page table for all of logical memory might be massive, only a small subset of it contains useful mappings
- We can take advantage of this fact in both TLB and page table design

```
Logical address
```

```
Physical address
```

```
Inverted TLB
```

Inverted Page Tables (cont’d)

- Efficiency considerations
  - the inverted page table is organized based on physical addresses via frame numbers
    - searching for the frame number can be very slow
  - use a hash table based on
    - the PID and logical page number as keys
  - recently located entries of the inverted page table can be stored in a TLB-like structure based on associative registers
- Main disadvantage of inverted page tables: sharing
  - each process that shares an object will have its own (disjoint) space where the shared object is mapped
  - not possible to maintain with standard inverted page tables
    - since space for only one <PID, page number> tuple
Protection Issues with Paging

- Partition protection scheme
  - Check that address lies between base and base+limit
  - Cannot be used on page-based systems: WHY?

- Physical memory can only be accessed through page table mappings
  - all addresses are interpreted by the MMU
  - OS intervention required to manipulate page tables and TLBs

- Special bits in the page table entry enforce per-frame protection
  - an accessibility bit
    • whether a page is invalid, readable, writable, executable
  - a valid/invalid bit to indicate whether a page is in the user's (logical) space

- Sometimes, the hardware may support a page-table length register
  - specifies size of the process page table
    • trailing invalid pages can be eliminated
    • useful when processes are using a small fraction of available address space

Memory Mapping (3): Segmentation

- A segment is a logical piece of the program
  - e.g., the code for the program functions, its data structures, symbol tables

- Segmentation views logical memory as broken into such segments
  - segments are of variable size (unlike pages)

- Accessing a segment
  - the logical address is regarded as two-dimensional
    • a segment pointer to an entry in the segment table
    • a displacement into the segment itself

- Allocating a segment
  - a segment is a partition with a single base-limit pair
    • the limit attribute stores the segment length
      • prevents programs from accessing locations outside the segment space
    • differs from partitioning in that there can be multiple segments/process

Memory Mapping: Segment Table Lookup

- Mapping logical addresses to physical addresses
  - the mapping is maintained by the segment table
  - the segment number s# is used to index into the (process') segment table
    where the corresponding segment size and base address are stored

Memory Mapping: Segmentation Hardware

- Segment registers
  - some designs (e.g., Intel x86) provide registers to identify segments
    • loading a segment register loads a (hidden) segment specification register from
      the segment table
    • construction of the logical address is done explicitly

- TLBs
  - some designs, such as the MIPS 2000, only provide a TLB
    • the OS is responsible for loading this, and doing appropriate translation

- Traditional approach: Store the segment table in memory
  - segment table base register (STBR), segment table length register (STLR)
    • saved and restored on each context switch
  - translation of address (s,d)
    • check that s is valid: s < STLR
    • Look up base address, limit: segment table entry at address (STBR + s)
    • check that offset d is valid: d < length
    • compute physical address
Segmentation: Pros and Cons

- **Pros**
  - protection in terms of ensuring that illegal address accesses are avoided, comes for free
    - the segment length check plays an important role here
  - sharing segments across programs is straightforward by loading identical segment table base register values
    - Caveat: How do instructions refer to addresses within segments?
      - Relative addressing works well with sharing
      - Absolute addressing does not: requires same segment number

- **Cons**
  - external fragmentation is potentially a big problem
  - contrast this with paging where only internal fragmentation is possible

Memory Mapping: Segmentation and Paging

- Overlay a segmentation scheme on a paging environment
  - several examples
    - originally proposed for GE 645 / Multics
    - Intel x86 uses segment registers to generate 32-bit logical addresses, which are translated to physical addresses by an optional multi-level paging scheme
  - alleviates the problem of external fragmentation

Memory Mapping: Summary

- **Partitioning**: Process is allocated a single contiguous region of memory
  - Translation and protection using size, limit registers
  - Suffers from external fragmentation

- **Paging**: Process pages are mapped into memory frames
  - Translation using per-process page table (TLBs cache translations)
    - Sharing possible by having multiple pages point to same frame
  - Protection because page-table mappings controlled by OS, extra bits ensure page being accessed in a valid fashion (e.g., read-only)
  - Internal fragmentation possible, but no external fragmentation

- **Segmentation**: Process is allocated multiple regions, one per segment
  - Translation and protection using size, limit registers
  - Sharing enabled by associating segment descriptors with same information
  - Suffers from external fragmentation, but this has smaller impact

Memory Mapping: Examples

- Multics (c. 1965)
  - 34-bit logical address
    - 18-bit segment number, 16-bit offset
    - [8-bit major segment, 10-bit minor segment], [6-bit page, 10-bit offset]
    - Both the segment table and segment itself are paged!

- **Segmentation structure**
  - Segment table is paged
  - major segment number indexes page table for segment table
  - minor segment number is offset within the page of the segment table
    - this gives the page table of the desired segment and the segment length

- **Paging structure**
  - one-level page table, 1KB pages

- **TLB**
  - 16 entries; key = 24-bit (seg# & page#); value = frame#
Memory Mapping: Examples (cont’d)

- OS/2 (on Intel 386+): Segmentation with paging

![Diagram of memory mapping]

OS/2 (on i386+) Memory Mapping (cont’d)

- Very flexible addressing scheme
  - pure paging
    - All segment registers set up with the same selector
    - Descriptor for this selector has base = 0, limit = MAXVAL
    - Offset becomes the address
  - pure segmentation
    - How can this be done?
  - options in between

Outline

- Announcements
  - Lab 4 due Monday, April 7th
  - Please sign up for demos on April 7th and 8th
- Memory Management
  - Paging (cont’d)
  - Segmentation
- Virtual Memory
  - Introduction
  - Demand paging

Virtual Memory

- Key ideas
  - Separation of logical and physical address spaces
  - Automatic memory mapping mechanisms which support
    - A large logical address space (bigger than physical memory)
    - On-demand movement of program components between the disk and memory (performed transparently by the OS using hardware support)
    - Demand paging + page replacement + frame allocation
  - Potential advantages
    - The programmer
      - Is not constrained by limitations of actual physical memory
      - Gets a clean abstraction of storage without having to worry about cumbersome attributes of the execution environment
        - Overlays, dynamic loading, disk transfers, etc.
    - The system
      - Benefits from a higher degree of multiprogramming
        - And hence utilization, throughput, …
Demand Paging

- Key mechanism for supporting virtual memory
  - Paging-based, but similar scheme can also be developed for segments
- The idea
  - Allocate (physical) frames only for the (logical) pages being used
  - Some parts of the storage reside in memory and the rest on disk
    - For now, ignore how we choose which pages reside where (next lecture)
- Strategy
  - Allocate frames to pages only when accessed
  - A lazy approach to page allocation
  - Deallocate frames when not used
- Implementation (must be completely transparent to the program)
  - Identifying an absent page
  - Invoking an OS action upon accesses to such pages
    - To bring in the page

Demand Paging: Identifying Absent Pages

- Goal: Determine when a page is not present in physical memory
- Extend the interpretation of valid/invalid bits in a page-table entry
  - valid: the page being accessed is in the logical address space and is present in a (physical) frame
  - invalid: the page being accessed is either not in the logical address space or is currently not in active (physical) memory
    - An additional check (of the protection bits) is required to resolve these choices
- The (hardware) memory mapping mechanism
  1. Detects accesses to pages marked invalid
    - Runs on each memory access: instruction fetch, loads, stores
  2. Causes a trap to the OS: a page fault
    - As part of the trap processing, the OS loads the accessed page
  3. Re-executes the instruction causing the trap
    - Amount of work involved depends on the architecture

Interrupting and Restarting

- Must make sure that it is possible to redo the side-effects of an instruction
  - Requires hardware support for precise exceptions
  - Note that page faults are only detected during instruction execution
    - An instruction can cause multiple page faults
- Some subtleties
  - Some architectures support primitive “block copying” instructions
    - Consider what happens if there is a page fault during the copy
    - Need to handle the situation where source and destination blocks overlap
  - What does it mean for the instruction to restart?
- See text book for other pathological cases that must be handled

Uses of Demand Paging

- Process creation
  - Load executable from disk on demand
  - UNIX fork semantics: child process gets a copy of parent address space
    - fork often followed by exec: explicit copying is wasteful
    - Demand-paging + page-protection bits enable copy-on-write
      - Child gets copy of parent’s page table, with every page tagged read-only
      - When a write is attempted to this page, trap to the OS
        » Allocate frame to hold (child’s copy of) the page, copy contents, permit write
  - Copy-on-write optimization (Q. 9.11): uses copy-on-write technique above
- Process execution
  - Frames occupied by unused data structures will eventually be reclaimed
    - Available for use by this and other processes
    - memcpy optimization: uses copy-on-write technique above
- Efficient I/O (Memory-mapped I/O)
  - Map files to virtual memory
  - Disk operations only initiated for accessed portions of the file
What Happens on a Page Fault?

On a page fault, the OS
1. Determines if the address is legal
   - Details are maintained in the PCB regarding address ranges
2. If illegal, “informs” the program
   - On Unix, a signal is sent to the process
3. Otherwise, allocates a frame
   - May involve “stealing” a frame from another page
4. Reads the requested page into the frame
   - Involves a disk operation
   - CPU can be context-switched to another process
5. Updates the page table
   - Frame information
6. Resumes the process

Cost of Demand Paging

- The cost of accessing memory
  - effective access time = \((1 - p) \cdot ma + p \cdot pf\)
  - where
    • \(ma\) is the memory access time when there is no page fault
    • \(pf\) is the page fault time
    • \(p\) is the probability of a page fault occurring
  - typical values
    • \(p\) is usually estimated empirically (and grossly) for the system
    • \(ma\) is 5-6 orders of magnitude lower than \(pf\) (order of tens of milliseconds)
  - disk access time
  - trapping the OS and saving user state
  - checking legality of page reference
  - context switch
  - when disk read is complete, interrupt existing user and save state
  - updating page table
  - restarting interrupted user process