V22.0202-001
Computer Systems Organization II (Honors)
(Introductory Operating Systems)

Lecture 14
Memory Management (contd.)
Virtual Memory

March 22, 2000

Outline

- Announcements
  - Graded write-up 2 will be returned on March 27th (my apologies)
    - I will hand out write-up 3 then and it will be due April 12th
  - Lab 4 due March 27th, TA has posted solutions for parts of Lab 3
  - Questions?
- Memory management (contd.)
  - Segmentation
  - Segmentation with paging
  - Example memory organizations
- Virtual memory
  - Background: Process working sets
  - Demand paging
  - Page replacement

[Silberschatz/Galvin: Sections 8.6 – 8.8, 9.1 – 9.4]

Memory Mapping: 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 is 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

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

Diagram:
- Logical address
- Linear address
- Segment Table
- Base-address
- Size
- Segment
- s#
disp
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
- 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: 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 (contd.)

OS/2 (on Intel 386+)
- Segmentation with paging
  - up to 16K segments (max 4 GB), 4KB pages

- Logical address space of process
  - divided into two partitions
    - first partition (up to 8K segments) are private (local descriptor table: LDT)
    - second partition (up to 8K segments) are shared (global descriptor table: GDT)
    - Each GDT/LDT entry is 8 bytes (24-bit segment base, 20-bit limit)
  - logical address: 16-bit segment selector, 32-bit offset
    - 16-bit selector: 13 (segment number) + 1 (GDT/LDT) + 2 (protection: rings)

- Machine has six 64-bit microprogram segment description registers
  - automatically loaded when a selector is loaded into a segment register

OS/2 Segmentation with Paging

- Translation of logical address into 32-bit physical address
  - load selector into segment register: microprogram register gets entry
  - check that offset is within limit: 20 bits interpreted as bytes or pages
  - form 32-bit linear address by adding 24-bit base to 32-bit offset
  - resolve 32-bit linear address using a 2-level page table scheme
    - 10-bit page directory, 10-bit page#, 12-bit offset

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

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)

- 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
  2. Causes a trap to the OS: a page fault
  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

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 = $(1 - p).ma + p.pa$
  
  - 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
Controlling Demand Paging Costs

Three degrees of freedom

- **Program structure**
  - Selection of data structures and programming structures
  ```
  var A: array [1..128] of array [1..128] of integer;
  for j := 1 to 128 for k := 1 to 128
    A[k][j] := 0;
  ```

- **Page replacement**
  - Given an allocation of frames to a process, how are these frames managed?
  - Algorithm must ensure that pages likely to be accessed are in memory

- **Frame allocation**
  - More frames allocated to a process → fewer page faults
  - How should the OS allocate frames to processes?

Page Replacement: Objectives

- In a fully-loaded system, all frames would be in use
- In general, page allocation involves
  - Selecting a page to "evict"
  - Writing it to disk (if it was modified)
  - Reading the new page from disk

- Objectives of page replacement/eviction policy
  - Remove a page with the least overall impact on system performance
    - (from the process’ perspective)
    - Minimize number of page faults
    - (from the system’s perspective)
    - Minimize disk activity

Page Replacement Algorithms: Components

- **Reference strings**: the sequence of page numbers being accessed
  - Example
    - A logical address sequence 0400, 0612, 0235, 0811, ...
    - Will yield the reference string 4, 6, 2, 8, ... (for 100-byte pages)

- **Hardware support**
  - Extra bits associated with the frames to store information about page use
    - Different from the bits stored in each page table entry
  - Commonly available: a page-referenced bit and a page-modified bit
  - Restriction: Must incur very low overhead to maintain
    - Potentially updated on every memory access

- **Algorithms**
  - FIFO algorithms
  - OPT (Clairvoyant) scheme
  - LRU algorithms and approximations

Page Replacement: FIFO

- Evict the page that brought in the earliest
  - **Pro**: Simple to implement
    - OS can maintain a FIFO queue and evict the one at the beginning
  - **Con**: Assumes that a page brought in a long time ago has low utility
    - Obviously not true in general (e.g., much-used library routines)

- How does FIFO perform?
  - Consider reference string (length 12)
    - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 (with 3 frames)
    - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 (with 4 frames)

**Belady’s anomaly**
Algorithms that don’t exhibit this behavior are known as stack algorithms
Page Replacement: What is the Best Algorithm?

- For read-only pages (discounting clean-page preference issues), it can be proven that the optimal algorithm (OPT) is
  - Replace the page whose next use is the farthest
    1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  (with 3 frames) \[\uparrow \uparrow \uparrow \uparrow \downarrow _3 \uparrow _4 \uparrow _1 \uparrow _2 \] (7)
  (with 4 frames) \[\uparrow \uparrow \uparrow \uparrow \downarrow _1 \uparrow _4 \downarrow _1 \] (6)

- Optimality stems from the fact that
  - The page replaced will cause a page fault far away
  - Any other page will cause a fault at least as quickly

- How do you prove that OPT does not suffer from Belady’s anomaly?

Next Lecture

- Virtual memory (contd.)
  - Page replacement algorithms: Can we match the behavior of OPT?
    - LRU (least recently used)
    - LRU approximations
    - Performance enhancements
  - Frame allocation algorithms
  - Why does virtual memory work?
    - Disk costs are very high, so even a small number of faults will hurt performance

- Reading
  - Silberschatz and Galvin, Sections 9.5 – 9.10