Outline

- Announcements
  - Lab 4 demos today and tomorrow
- Memory Management (cont’d)
  - Examples
- Virtual Memory
  - Introduction
  - Demand paging
  - Page replacement algorithms
    - FIFO
    - OPT
    - LRU

[Silberschatz/Galvin/Gagne: Sections 9.6, 10.1 – 10.4]

(Review) Memory Mapping

- **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# and page#); value = frame#
Memory Mapping: Examples (cont’d)

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

```
Local
  8K segments

Global
  8K segments
```

Logical Address:

```
16 bits
selector

32 bits
offset
```

Local/global

```
20-bit limit

24-bit base
```

8K segments

```
Logical Address:
Linear Address:
```

```
10 bits
directory

12 bits
page

12 bits
offset

Directory table

Page table

Frame
```

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

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, …

(1) 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

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

- **Process execution**
  - Frames occupied by unused data structures will eventually be reclaimed
    - Available for use by this and other processes
  - `memcpy` optimization (Q 10.11): 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 = \((1 – p).ma + p.pa\)
  - 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 \(\rightarrow\) 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 was 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)

  ![FIFO with 3 frames](image)

  (with 4 frames)

  ![FIFO with 4 frames](image)

  **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)

  ![OPT with 3 frames](image)

  (with 4 frames)

  ![OPT with 4 frames](image)

- 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?

Page Replacement: LRU

- Problem with OPT: Clairvoyance is generally not possible
  - But sometimes possible to analyze deterministic algorithms
  - In any case, a good baseline to compare other policies against

- LRU (least recently used) is a good approximation of OPT
  - Assumes that recent past behavior is indicative of near future behavior
    - A phenomenon called **locality** which is exploited repeatedly in virtual memory

- Main idea: Evict the page that has **not been used** for the longest time
  
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

  

  (with 3 frames)

  ![LRU with 3 frames](image)

  (with 4 frames)

  ![LRU with 4 frames](image)

Page Replacement: LRU (cont’d)

- LRU works reasonably well in simulations
  - “real” program traces exhibit locality
  - but, some pathological access patterns
    
    1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, …

  

  (with 3 frames)

  ![LRU cont’d with 3 frames](image)

- Main problem with LRU: How does one maintain an **active history** of page usage?
  - Counters
  - Stack
Page Replacement: Implementing LRU

- Counters
  - Attach to each frame, a counter that serves as a logical clock
    - Updated by the hardware on every reference
  - Page replacement: choose page in frame with smallest counter value
    - Counter is reset when a new page is loaded
  - Problems: Elaborate hardware, Search time
  - Largely of theoretical value

- Stack
  - Maintain a stack of page numbers
    - On each access, hardware moves the page# to the top of the stack
  - Page replacement: the LRU page is at the bottom of the stack
  - Typical implementation: microcoded doubly linked list
    - Used by one of the earlier CDC machines
  - Still too high a hardware cost

Page Replacement: LRU Approximations

- Page reference bit
  - Stored with the frame containing the page
  - Bit is set whenever the page is accessed
  - Periodically, the OS (or hardware) resets all reference bits
  - Page replacement: Choose an unreferenced page

- Additional reference bits
  - For each page \( p \), OS maintains an \( n \)-bit last-reference-time \( lrt[p] \)
  - Periodically, OS (or hardware)
    - Shifts right \( lrt[p] \), adds current reference bit as MSB, and resets reference bit
  - Note that the additional bits can be maintained in software
  - Page selected is the one with the lowest \( lrt \)
    
    \[
    lrt[p1] = 11000100 \text{ has been used more recently than } lrt[p2] = 01110111
    \]

Page Replacement: LRU Approximations (cont’d)

- Second-chance Algorithm (also known as Clock)
  - Only uses single-bit page reference information
  - Maintains a list of frames as a circular list
  - Maintains a pointer into the list
  - Replacement: search for a page with reference bit zero
    - If there is a page with reference bit 1
      - Set the bit to 0, and continue searching
    - Each page gets a second chance before being evicted

- Enhanced second-chance algorithm
  - Make decision using two bits: page reference and page modify
    - \( (0, 0) \): neither recently used nor modified: best candidate
    - \( (0, 1) \): not recently used but modified
    - \( (1, 0) \): recently used, but not modified
    - \( (1, 1) \): recently used and modified: worst candidate
  - Used in the Macintosh