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

Outline

- Announcements
  - Lab 4 due March 28th, please sign up for demos if you have not done so
  - Questions?

- Virtual memory
  - Background: Process working sets
  - Demand paging
  - Page replacement algorithms
    - FIFO
    - OPT
    - LRU

[Silberschatz/Galvin: Sections 9.1 – 9.4]
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 textbook 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) $\uparrow \uparrow \uparrow \downarrow _1 \downarrow _2 \downarrow _3 \downarrow _4$ (9)
  - (with 4 frames) $\uparrow \uparrow \uparrow \uparrow \downarrow _1 \downarrow _2 \downarrow _3 \downarrow _4$ (10)

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)

3
4
1
2

(7)

(with 4 frames)

4
1
2
3

(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?

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)

1
2
4
1

(10) versus FIFO (9) and OPT (7)

(with 4 frames)

3
4
5
1

(8) versus FIFO (10) and OPT (6)

Page Replacement: LRU (contd.)

- 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)

1
2
4
1

- 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
Next Lecture

- Virtual memory (contd.)
  - Page replacement algorithms
    - 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