Virtual Memory

Key ideas
- separation of logical and physical address spaces
- automatic memory mapping mechanisms which supports
  - 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, etc.
- the system
  - benefits from a higher degree of multiprogramming
    - and hence utilization, throughput, ...

Demand Paging

Key mechanism for supporting virtual memory

The idea
- allocate (physical) frames only for the (logical) pages being used
- some parts of the process storage reside in memory and the rest on disk

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
Identifying Absent Pages

- 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 required to resolve these two choices

- The (hardware) memory mapping mechanism
  1. detects accesses to pages marked invalid
  2. runs on each memory access: instruction fetch, operand fetch, result storing
  3. causes a trap to the OS: a *page fault*
     - as part of the trap processing, the OS loads the accessed page
  4. 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
   - in Unix, sent a signal
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
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
Controlling Demand Paging Costs

Three degrees of freedom

- Program structure
  - selection of data structures and programming structures affects locality

```plaintext
var A: array [1..128] of array [1..128] of integer;
for j := 1 to 128    for k := 1 to 128  for k := 1 to 128      for j := 1 to 128
  A[k][j] := 0;        A[k][j] := 0;
```

- Page replacement
  - given an allocation of frames to a process, how are these frames managed?

- Frame allocation
  - 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 "purge"
  - writing it to disk (if it was modified)
  - reading the new page from disk
- Objectives of page replacement / eviction policy
  - remove a page with least overall impact on system performance

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)

- HW support
  - extra bits associated with the frames to store information about page use
  - commonly available: a page-referenced bit, and a page-modified bit
  - restriction: must incur very little overhead to maintain

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

Page Replacement: FIFO

- Evict the page that was brought in the earliest
- **Pros**: Simple to implement
  - can maintain a FIFO queue & evict the one at the beginning
- **Cons**
  - 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 it perform?
  - consider reference string (length 12)
    1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
    (with 3 frames)
    ```plaintext
    \[ 1 \uparrow \uparrow \uparrow \quad 2 \quad 3 \quad 4 \quad 1 \quad 2 \quad 5 \]
    \[ 1 \quad 1 \quad 2 \quad 3 \quad 4 \]
    ```
    (with 4 frames)
    ```plaintext
    \[ 1 \uparrow \uparrow \uparrow \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 1 \]
    ```
  - Belady’s anamoly
    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)

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
(with 4 frames)

- Optimality stems from the fact that:
  - the page moved 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 anamoly?

Page Replacement: LRU

- Problem with OPT: Clairvoyance is not possible
  - but sometimes possible to analyze deterministic algorithms
  - however, 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

- Evict the page that has not been used for the largest time

- Performance: Works reasonably well in simulations
  - construct a reference trace where LRU performs worse than OPT

- Problem: How does one maintain an active “history” of page usage?
  - counters
  - stack

Page Replacement: Implementing LRU

- Counters
  - attach to each page, a counter that serves as a logical clock
    - updated by the hardware on every reference
  - page replacement: choose the page with the smallest counter value
  - 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

Page Replacement: LRU Approximations

- Page reference bit
  - periodically, OS (or hardware) resets all reference bits
  - on page replacement: select unreferenced page

- Additional reference bits
  - for each page, 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
    - page selected is that with the lowest \( lrt \)

- Second-chance cyclic (also known as Clock)
  - maintains the 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
Page Replacement: Performance Enhancements

- Maintain a *pool* of free frames
  - buffered (delayed) writes
    - frame allocation *precedes* deallocation
    - allocate immediately from pool, replace later
  - rapid frame and page reclaim
    - keep track of which page was in which frame
    - reclaim pages from free pool if referenced before re-use
      - can be used as an enhancement to FIFO schemes
      - used in VAX/VMS

- Background updates of writes to secondary store
  - whenever the disk update mechanism is free
    - write out a page whose modified bit is set and then reset the bit

- Delayed write (copy-on-write)
  - create a *lazy* copy (on the first write)
  - used to optimize Unix *fork*

Frame Allocation

- How do we allocate frames to processes?
- Naive single-user system
  - keep a list of free frames
  - allocate from this list
  - use eviction algorithm when list exhausted

- Problem: Multiprogrammed systems
  - how many frames for each process?
    - performance varies dramatically with the number of frames
      - e.g., matrix multiplication \((A := B \times C)\)
        - square matrices of size 64x64, 4-byte words
        - a page size of 4096 bytes (each matrix can fit in 4 pages)
      - with 12 page frames, get only 12 page faults
      - with 5, get >4096
      - with 1, get >250000

Frame Allocation: Two Critical Questions

- How many frames to assign to each process?
  - fixed
  - variable (from a global pool)
  - is there a minimum (critical) number of frames that must be allocated?

- How are they assigned?
  - when a new process needs more frames, do we
    - take away uniformly from a given process
    - or do we assign frames back and forth between processes?

Frame Allocation Algorithms: How Many?

- *Static* approach
  - allocates once and stays fixed during the process's life

- *Uniform* allocation
  - given \(m\) frames and \(n\) processes, allocate \(m/n\) per process
  - very simple but can lead to a lot of wasted frame usage since *even the size of each process' virtual space is not considered*

- *Proportional* allocation
  - let \(S\) be the sum of all the virtual memory "needs" across processes where \(s_i\) is the virtual memory associated with process \(i\)
    - Allocate \((s_i / S) \times m\) frames to process \(i\)
  - problems
    - does not distinguish between process priorities (could be patched with weights)
    - does not distinguish between process behaviours
Frame Allocation: Scope of Replacement

- How are additional requests for frame allocation satisfied?

  - Local replacement
    - new frames are allotted to pages from a fixed set associated with the process
    - number does not change with time

  - Global replacement
    - new frames can be selected from a variable pool that is shared by the whole system
    - the performance due to page faults of any one process is dependent on the behavior and demands of the others using this approach
      • can use ideas similar to “currencies” to insulate processes from each other

Frame Allocation: Constraints on Number of Frames

- Hardware: Determined by page fault induced instruction restarts
  - need frames to store all the needs of a single instruction
  - could be more than one page
    • CISC instruction may straddle page boundary
    • data may straddle page boundary
    • indirect addressing may straddle page boundary

- Software: Clearly there is a constraint
  - if a process gets too few frames, it spends all its time demand paging
  - this phenomenon is called thrashing
  - formally,
    • over any time window and summed over all processes, let T be the time spent by the process in computing and P be the time spent in page faults
    • a characterization of thrashing in a time window is when T < P
  - we can define it but can we do anything to reduce it?

Dealing with Thrashing

- The idea
  - exploit the fact that programs demonstrate temporally localized behavior in terms of their memory accesses
  - over each “time window”
    • monitor the behavior of active processes
    • estimate how many pages each process “needs”
    • adjust the frame allocation (and multiprogramming level) accordingly

- The working set of a process over time window W is the set of pages it accesses within W
  - use of the working set
    • choose a parameter W
    • over a time window of size W, estimate the size |wi| of the working set of each process i
    • do not activate more processes if the current sum of the |wi| together with the set |wj| of the new process j exceeds the number of frames

Computing Working Sets

- Maintaining the working set for each process and hence its cardinality (size) on each access leads to too much overhead

- Typically, approximations are used
  - the update is done only a few times in the window of choice
  - a common technique is similar to that used for LRU approximations
    • a frame is considered “freeable” if it is not referenced within the WS window

- Detailed discussion of working sets in Lecture 14 (March 11, 1999) [from Denning’s paper]
  - why do processes exhibit working sets
  - how does one choose the right window size
  - efficient methods of estimating working set sizes
Demand Paging: Other Issues

- I/O interlocking
  - need to ensure that I/O does not refer to pages that are swapped out
  - two common solutions
    - use kernel buffers to receive I/O responses
    - "pin-down" (also called "lock-down") the concerned pages

- Prepaging (warm start)
  - initial working set is brought in as one block
  - advantageous when the cost of bringing in a block is lower than that of generating page faults to bring in the subset of the working set that is used

- Choice of page size
  - large pages yield smaller tables, smaller I/O costs, fewer page faults
  - small pages yield less external fragmentation, less overall I/O
  - trend towards larger page sizes
    - limiting factor is reducing the number of page faults (since disks are slow)

Next Two Classes

- March 09, 1999: Mid-term
  - Silberschatz/Galvin Chapters 1–9
  - papers: [Laue78], [Bers90], [Wald94]
  - 4 questions:
    - 1 multiple-choice
    - 1 on IPC + process scheduling
    - 1 on process synchronization + deadlocks
    - 1 on memory management + virtual memory

- March 11, 1999: Virtual Memory (contd.)
  - process working sets [Dem80]
  - page tables for 64-bit address spaces [Tall95]