Silberschatz, et al.
Topics based on Chapter 10

Virtual Memory

Virtual memory

- **Virtual memory** permits the execution of processes that are not entirely in physical memory
- Programmer benefits
  - Programmer retains view of memory as a large, uniform array of storage
  - |logical storage| >= |physical storage|
- But requires care in implementation

Virtual Memory

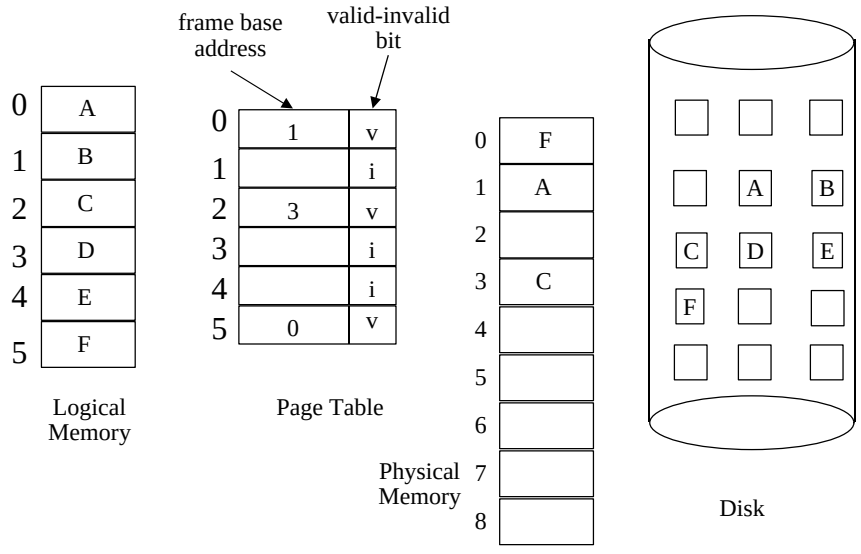
- Further justification for virtual memory
  - in many (most) cases, the entire program does not need to be in memory at the same time.
    - example: error code, sparse arrays, specialist commands
    - common estimate: program spends 90% of time in 10% of code
  - Often the program's demand on memory may change during runtime
  - Efficiency: more users can fit into memory, less i/o to swap active part of process, etc.

Demand Paging

- Demand paging: only bring in a page when it is demanded (an implementation of virtual memory).
- *Lazy swapper*: never swaps a page into memory unless that page will be needed
- Actually not a *swapper* (which manipulates entire processes). Called a ***pager*** instead.
- When swapping a process in, the pager has to guess which pages will be used and bring them in. If the guess is wrong, then missing pages have to be brought in, perhaps displacing existing pages (which?).

Demand Paging

- Demand paging requires hardware support
- valid/invalid bit attached to each page table entry (in addition to the frame base address that was already there)
- When “valid”, the page is in memory
- When “invalid”, it is not in memory
- **page fault**: attempt to access an address on an invalid page



Page Fault

- Page fault: a page is not in memory when a reference to it is made
- Paging hardware notices that “invalid” bit is set and causes a trap to the operating system
- Operating system first has to determine why it was interrupted--missing page (valid memory reference) or out-of-range reference (invalid memory reference)
- If invalid, then terminate process, otherwise have to bring in the new page.

Page Fault

- Bringing in a new page
  - suspend the process, saving user registers and process state
  - locate a free frame in memory (or create one)--mechanisms to be discussed
  - locate the missing page on backing store (its location on disk)
  - bring in the missing page to memory.

Page Fault

- Bringing in a new page (continued)
  - transfer requires the following steps
    - wait in queue for device until read request is serviced
    - wait for device seek and/or latency time
    - begin transfer to free frame
    - while waiting for the transfer to process, can let another process use the CPU, but need to save its state before continuing on

Page Fault

- Bringing in a new page (continued)
  - when transfer completes, correct page tables, marking page "valid"
  - reset process' PC in PCB so that when it starts the instruction will be re-executed
  - change process state from "waiting" to "ready"

Page Faults

- Sometimes page faults occur in the middle of instructions
  MOVE (SP)+,R2
- Just reexecuting the instruction may not be correct if some memory or registers change during the transfer (for example a block transfer in which the source and destination overlap)
- Possible solutions
  - determine if there is a page fault before any modification is done
  - hold the results temporarily until it is certain that there is no page fault
- Easiest to handle this kind of problem in the architecture design. Workarounds after the fact are more complicated. Indeed may not be possible to detect all such side effects without hardware assistance.

Page Faults

- In summary, there are three major tasks
  - Service the page-fault interrupt
  - Read in the page
  - Restart the process
- Times needed are significant
  - Service the page-fault interrupt: can be reduced with careful coding to several hundred instructions
  - Read in the page: significant amount of time (disk access speeds)
  - Restart the process: again can be accomplished in several hundred instructions

Performance of Demand Paging

Effective Access Time

Definition: average time taken by a memory access.
It involves with the following terms:

* ma: (average) time for a memory access without a page fault. ma ≈ 100 nanoseconds.

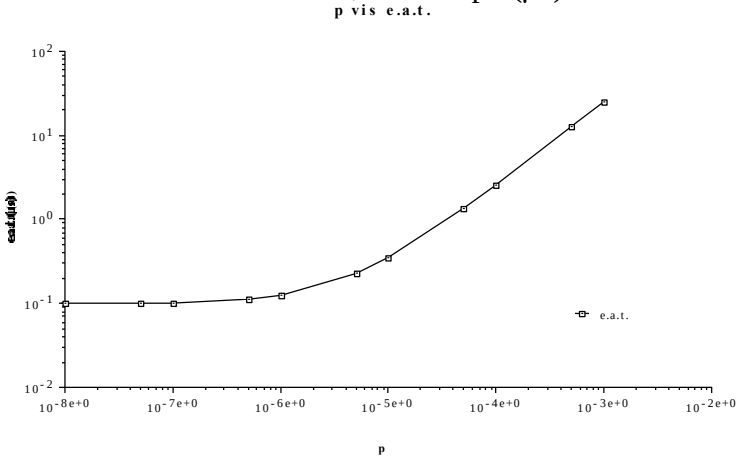
* page fault time: (average) time taken by a page fault:

  - O.S. service time when a page fault is detected
    ≈ 100 microseconds
  - Time by disk to read the page into memory
    ≈ 25 milliseconds
  - Time to restart the process
    ≈ 100 microseconds

  total of page fault time ≈ 25,000,000 nanoseconds.

* p: probability that a page fault occurs when a memory reference is made.

Effective Access Time = (1-p) * ma + p * page fault time
≈ 0.100 + 24,999.900 p (μs)



Effective Access Time

- e.a.t. becomes unbearable if there are lots of page faults
- Key notion for reducing the effective access time
  - Reduce the page fault rate
    - use a better page-replacement algorithm
    - allocate more pages to a process

Scheduling questions

- When to bring pages into memory (and into swap space) is a scheduling issue.
  - demand paging is one approach. prepaging and user-specified paging are additional ideas
- When to *replace* pages already in memory is another scheduling issue
  - for example: no frames on the free frame list. How do we decide what to remove from memory?
  - replacement algorithm is key in reducing average e.a.t.

Page Replacement

- The problem: decide which page to replace when there are none free
- Algorithms may need to augment page table with additional status bits
  - dirty (modified) bit
  - use bit
  - etc.
- More global question: frame allocation algorithm (how many frames to allocate to a given process)

Page Replacement

- Page replacement algorithms
  - metric (good) is lowest page-fault rate
  - **reference string**: string of page numbers representing references
  - reference strings can be generated artificially or can be captured from a running system
    - generally compress repeated elements in sequence to make the size more managable

      "rpppr" becomes "rpr"
    - Example: references (with 100 bytes per page)
      0100, 0432, 0101, 0612, 0102, 0103 0104, 0101,0611
      reference string 1, 4, 1, 6, 1, 6

Page Replacement Algorithms

FIFO algorithm

Replace the page which comes earliest.

Example

Page referenced	7	0	1	2	0	3	0	4	2	3
Memory status after the reference	7 - -	7 0 -	7 0 1	2 0 1	2 0 1	2 3 1	2 3 0	4 3 0	4 2 0	4 2 3
Page fault?	y	y	y	y	n	y	y	y	y	y

Another Example: Total number of pages for the process = 3

Page referenced	1	2	3	4	1	2	5	1	2	3	4	5
Memory status after the reference	1 - -	1 2 -	1 2 3	4 2 3	4 1 3	4 1 2	5 1 2	5 1 2	5 1 2	5 3 2	5 3 4	5 3 4
Page fault?	y	y	y	y	y	y	y	n	n	y	y	n

Page fault rate = 9/12

Another Example (cont): Total number of pages for the process = 4

Page referenced	1	2	3	4	1	2	5	1	2	3	4	5
Memory status after the reference	1	1	1	1	1	1	5	5	5	5	4	4
	-	2	2	2	2	2	2	1	1	1	1	5
	-	-	3	3	3	3	3	3	2	2	2	2
	-	-	-	4	4	4	4	4	4	3	3	3
Page fault?	y	y	y	y	n	n	y	y	y	y	y	y

Page fault rate = 10/12

While the number of pages is increased, the p.f.r. increases as well!

This is called Belady's anomaly!

Need a better page replacement algorithm.

Optimal Page Replacement Algorithm (OPT)

- Also known as MIN
- Replace the page that will not be used for the longest period of time
- Obvious disadvantage: have to predict the future

Example OPT vis FIFO

Page referenced	7	0	1	2	0	3	0	4	2	3	0 ....
Memory status after the reference	7	7	7	2	2	2	2	2	2	2	
	-	0	0	0	0	0	0	4	4	4	
	-	-	1	1	1	3	3	3	3	3	
Page fault?	y	y	y	y	n	y	n	y	n	n	

Memory status after the reference	7	7	7	2	2	2	2	4	4	4
	-	0	0	0	0	3	3	3	2	2
	-	-	1	1	1	1	0	0	0	3
Page fault?	y	y	y	y	n	y	y	y	y	y

OPT problems

- It is impossible to have knowledge about future references
- Try approximations that use the past to predict the future. For example
  - a page used recently is likely to be used in the near future
  - a page unused recently is unlikely to be used in the near future

Least Recently Used Algorithm (LRU)

- Replace the page that has not been used for the longest period of time

Example OPT vis LRU

Page referenced	7	0	1	2	0	3	0	4	2	3	0 ....
Memory status after the reference	7	7	7	2	2	2	2	2	2	2	
	-	0	0	0	0	0	0	4	4	4	
	-	-	1	1	1	3	3	3	3	3	
Page fault?	y	y	y	y	n	y	n	y	n	n	

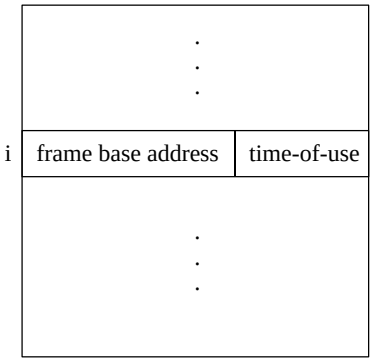
Memory status after the reference	7	7	7	2	2	2	2	4	4	4
	-	0	0	0	0	0	0	0	0	3
	-	-	1	1	1	3	3	3	2	2
Page fault?	y	y	y	y	n	y	n	y	y	y

Implementation of LRU

- How do we implement? Some alternatives
  - keep time-of-use register for each page, store system clock into that register on each memory reference. To discover least recently used frame, scan time registers and find the one with the oldest clock (this is expensive)
  - keep stack of page numbers. Reference to page causes entry to be moved to top of stack. Top is most recently used. Bottom is least recently used. Since stack entries are removed from the middle of the stack, a doubly-linked list implementation is best (6 pointer changes at worst--2 for internal neighbors, top of stack, previous top's pointer, entry's 2 pointers
- LRU does not suffer from Belady's anomaly

Implementation of LRU

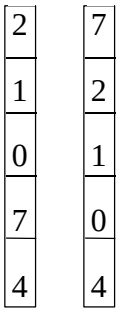
• Time-of-use register in every entry of page table.



• Stack of page numbers

Reference string

4 7 1 0 1 2 1 2 7 1...



LRU Discussion

- Implementation of LRU can be expensive, so in practice may approximate LRU
- Use of reference bit
  - every entry in page table has reference bit
  - bit set (to 1) when page accessed
  - periodically the OS "cleans" the reference bits (resets all to 0)
  - pages with bit reset (i.e., 0) have higher priority to be replaced
- Reference bits tell us what pages have been accessed but do not tell us the order of use

LRU Approximation

- Additional reference bits algorithm
  - keep record of reference bits at regular time intervals
  - associate 8 bit byte with each page table entry
  - at regular intervals (e.g., 100 miliseconds) OS shifts reference bit for each page into high-order bit of 8-bit byte, shifting other bits right by one. low-order bit discarded
    - 00000000 not used in 8 time units
    - 11111111 used in each of past 8 time units
    - 00111111 not used in last 2 time units but used in each of 6 previous ones
  - If view numbers as unsigned integers, the page(s) with the lowest number is the one least recently used (not guaranteed to be one page--can be ambiguous then)

LRU Approximation

- Second-chance algorithm (aka clock algorithm)
  - FIFO replacement but inspect reference bit before actually replacing
  - if reference bit == 0 then replace
  - if reference bit == 1 then clear reference bit and move on to inspect the next FIFO page
  - Implementation: circular queue; advance pointer, clearing reference bits until find one with reference bit == 0
  - Worst case: cycle through all entries (degenerates to FIFO in this case)
  - Slowly sweeping clock hand is good---means there is plenty of memory, not many page faults

LRU Approximations
Ad-hoc algorithm

- Keep pool of free frames
- When new frame needed, select one to be freed
- Obtain frame from pool to receive new page, transfer page in
- Initiate transfer of freed page
- When transfer finishes, move page to pool (overlaps page out time with subsequent computation)
- VAX/VMSalso keeps track of what is in the pool and if the page is needed before it is overwritten, recover it from the free pool. This brings back active pages before they get “wiped out”.
- Additional nuance: keep “dirty” bit to give preference to dirty pages. It is more expensive to throw out dirty pages since clean pages do not have to be written to disk.

Frame Allocation

- The problem: deciding how to allocate the fixed amount of free memory among the various processes that are active. In other words, how many frames can a process have?
- Obviously there is a maximum number of frames that can be allocated (e.g., the number available in the system) and a minimum (e.g., the number needed to execute an instruction)
  - Minimum number example: memory reference instructions have only one address. One frame needed for instruction and one for memory reference
  - Another example: one level of indirection--requires a third frame
  - Worst case--multiple levels of indirection
- Maximum is defined by the amount of available physical memory
- Minimum number is determined by computer architecture

Frame Allocation

- Simple algorithm: equal allocation
  - divide available space equally among processes
  - any leftover is used as a free-frame buffer pool
- Different processes need different amounts of memory
- Proportional allocation
  - $D_i = M * S_i / S$
  - D_i, an integer, the number of frames allocated to process P_i, M the size of physical memory, S_i the size of virtual memory for P_i, S is the sum of all S_i

Multiprogramming Level

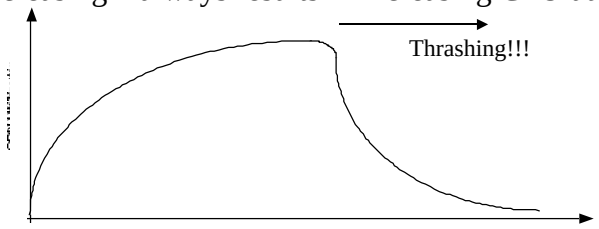
- How do we determine n, the number of processes in memory? (n is the multiprogramming level.)

Determining the Degree of Multi-programming

- Monitor the CPU utilization
- If CPU utilization is too low, increase n (add to ready queue).
- If CPU utilization is too high, reduce n (ready queue too long).

Problem?

Will increasing n always results in increasing CPU utilization?



Thrashing

- CPU utilization increases with increasing n to some point and rapidly drops to 0
- Why? Thrashing
- Consider individual processes as memory fills up
  - process starts faulting and taking pages away from other processes. Those other processes in turn must fault to bring their pages back. The scheduler sees decreasing CPU utilization and increases the degree of multiprogramming. CPU utilization continues to drop.
  - Effective access time (e.a.t.) increases and no useful work is getting done because processes are spending all their time paging

Thrashing

- A process is thrashing if it is spending more time paging than executing
- A system is thrashing if increasing the degree of multiprogramming reduces CPU utilization

Thrashing
Local Replacement Algorithm

- Can *limit* (not prevent) effects of thrashing with a **local** (or **priority**) replacement algorithm
- Local replacement: each process must select a new frame from only its set of allocated frames
  - system throughput better with global allocation because a process gives up unneeded pages
- A thrashing process cannot cause other processes to thrash. Effective access time does increase because of greater contention for paging device caused by thrashing process

Thrashing Prevention

- To prevent thrashing must
  - allocate sufficient frames to each process in memory. (How do you determine this? To be discussed.)
  - increase degree of multiprogramming only if current processes have had enough frames and CPU utilization is still low.

Thrashing
Locality Model

- Determine the "right" number of frames to allocate a process based on locality model of process execution
- Locality model: As process executes it moves from (potentially overlapping) locality to locality. A locality is the set of pages actively used together
- If we do not allocate enough pages to process to accomodate current locality, the process will thrash because it cannot keep all the pages it needs in memory.

Working-Set Model

- Assumes locality
- Working set of a process: the most recent Δ page references. Δ defines the *working-set window*
- The working-set approximates the process' locality
- Can reuse pages outside of the working set
- Note that reference strings are not minimized for WS

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...

$WS(t_1) = \{1,2,5,6,7\}$ $WS(t_2) = \{3,4\}$

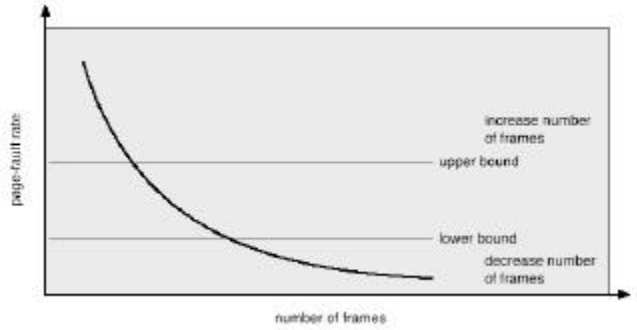
Working-set model

- Working-set model due to P. Denning
- Selection of Δ is tied to the accuracy of the working set. If Δ is too small it doesn't encompass the actual working set. If Δ is too large it overlaps several localities. Consider infinite Δ--this is the entire program!
- The total demand for frames is the sum of the individual processes' working sets
- If the total demand exceeds the number of available frames then thrashing will occur (since some processes need frames)

Working-set Model

- The difficulty in implementing the working-set model is keeping track of the working set window.
- Every page reference adds an element to the beginning of the window and drops one from the end
- One idea: approximate working set model with aging
  - For example, given reference bit, two in-memory copy bits, Δ of 10000 and timer interrupt every 5000 references, shift in and clear reference bit on every timer interrupt. Looking at the two copy bits and the reference bit lets us determine if the page was referenced in the previous 5000 to 15000 instructions. Consider those pages with at least one of the bits on to be in the working set.

Page-Fault Frequency Scheme



- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame

Other Considerations

- Considerations in addition to those of replacement algorithm and allocation policy
  - Prepaging
  - Page size selection
  - Page table structure
  - Effects of program structure and environment requirements

Prepaging

- High level of paging required when program starts, due to trying to get initial locality into memory
- Strategy: at same time, bring all pages into memory that will be needed
- Example: working set. Keep list of pages in working set and restore them
- Prepaging effort wasted if pages not used.

Page size selection

- Powers of 2, from 512 (2^9) to 16,384 (2^14) bytes
- Issues
  - Fragmentation: internal fragmentation; 50% of last page on average
  - Table size: large pages mean smaller page table (which is copied for each active process)
  - I/O overhead: latency and seek dwarf transfer time, hence large page size benefits
  - Locality: smaller page size favors locality
  - Page faults: larger size tends to reduce faults
- Trend: towards larger page sizes; no single best size for all circumstances

Paging--comment

- Higher-level programming languages may unintentionally cause major amounts of paging (or make it easy to accidentally specify programs that cause much paging). Consider row-major versus column-major access in 2D arrays, for example

Program Structure

- Array A[1024,1024] **of** integer
- Each row is stored in one page
- Program 1
```
for j := 1 to 1024 do
for i := 1 to 1024 do
A[ i,j]:=0;
```
- 1024 × 1024 page faults
- Program 2
```
for i := 1 to 1024 do
for j := 1 to 1024 do
A[ i,j]:=0;
```
- 1024 page faults