Demand Paging, Thrashing, Working Sets (Topic 7)

홍 성 수

서울대학교 공과대학 전기 공학부
Real-Time Operating Systems Laboratory

Demand Paging, Thrashing, Working Sets (1)

- Topic 6: Mapping hardware
  - Separated the programmer's view of memory from system's view
  - Mechanism: Mapping hardware
  - Each user sees a different memory organization
  - Allows OS to shuffle users around and simplifies memory sharing between users

Demand Paging, Thrashing, Working Sets (2)

- Topic 6 required the user process to be completely loaded into memory
  - Wasteful because of locality of reference

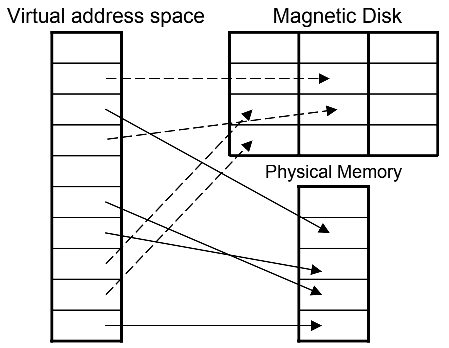
    A process only needs a small amount of its total memory at any one time

Demand Paging, Thrashing, Working Sets (Intro)

- Solution: virtual memory
  - A process can run with only some of its virtual address space loaded into physical memory

Demand Paging, Thrashing, Working Sets (Intro)

- Virtual address space:



Demand Paging, Thrashing, Working Sets (Intro)

- Virtual address translates to either:

  (a) Physical memory (1000원/megabyte): Small, fast

  (b) Disk (backing store) (50원/megabyte): Large, slow

  (c) Error — Not valid (Free)

Demand Paging, Thrashing, Working Sets (Intro)

- Idea: Produce the illusion of a disk as fast as main memory
  - Works because programs spend their time in only a small piece of the code
  - Knuth's estimate: 90% of the time in 10% of the code
- Key principle: Locality
  - Spatial locality
  - Temporal locality

Page Faults (1)

- If not all of process is loaded when it is running, what happens when it references a byte that is only in the backing store?
  - Hardware and software cooperate to make things work anyway

  (1) Extend the page tables with an extra bit “present (valid)”

  If present isn't set, then a reference to the page results in trap.

  This trap is given a special name, page fault.

Page Faults (2)

(2) Any page not in main memory right now has the “present” bit cleared in its page table entry

(3) When page fault occurs:

Operating system rings page into memory

Page table is updated, “present” bit is set

The process continues execution

Page Faults (3)

- Continuing process is very tricky, since page fault may have occurred in the middle of an instruction.
  - Don’t want user process to be aware that the page fault even happened
  - Can the instruction just be skipped?
    No: Wouldn’t be transparent to the process
  - Suppose the instruction is restarted from the beginning?
    What about instruction like:
    MOVE (SP)+, -(R2) or Block transfer
  - Requires hardware support to restart instructions

Page Faults (4)

- If you think about this when designing the instruction set, it isn’t too hard to make a machine virtualizable. It’s much harder to do after the fact. VAX is example of doing it right.

Demand Paging (1)

- Key Issues:
  - Page selection:
    - When to bring pages into memory?
  - Page replacement:
    - Which page(s) should be thrown out, and when?
  - Page replacement style:
    - Global vs local replacement

Demand Paging (2)

- Page selection policies:
  - (1) Demand paging:
    - Start up process with no pages loaded.
    - Load a page when a page fault for it occurs.
      - Wait until it absolutely MUST be in memory.
    - Almost all paging systems are like this.
  - (2) Prepaging:
    - Bring a page into memory before it is referenced.
    - When a page is referenced, bring in the next one, just in case.
    - Hard to do effectively without a prophet.
    - Sometimes works: sequential read-ahead

Demand Paging (3)

(3) Request paging:
- Let user say which pages are needed.
- What's wrong with this ?
  - Users don't always know best.
  - Users aren't always impartial.
  - (They overestimate needs.)
- Example: Overlay

Demand Paging (4)

- Page Replacement Algorithms:
  - (1) Random: Pick any page at random.
    - Works surprisingly well!
  - (2) FIFO: throw out the page that has been in memory the longest.
    - The idea is to be fair.
    - Give all pages equal residency.

Demand Paging (5)

(3) OPT: Throw out the page that won't be used for the longest time into the future.
- As always, the best algorithm arises, if we can predict the future.
- It isn't practical, but it is good for comparison.

(4) LRU: Throw out the page that hasn't been used in the longest time.
- Use the past to predict the future.
- With locality, LRU approximates OPT.

Demand Paging (6)

Example: Reference string A B C A B D A D B C B:

	FIFO			OPT			LRU		
A B C	A	B	C	A	B	C	A	B	C
A	A	B	C	A	B	C	A	B	C
B	A	B	C	A	B	C	A	B	C
D	D	B	C	A	B	D	A	B	D
A	D	A	C	A	B	D	A	B	D
D	D	A	C	A	B	D	A	B	D
B	D	A	B	A	B	D	A	B	D
C	C	A	B	C	B	D	C	B	D
B	C	A	B	C	B	D	C	B	D

Demand Paging (7): Stack algorithms

- Belay's anomaly: Page fault rate may increase as the number of allocated frames increases.
  Reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  with FIFO algorithms with frames 3 and 4, respectively.
- Stack algorithms never exhibit Belady's anomaly.
  A set of pages in memory for `n` frames in always
  a subset of pages in memory for `n+1` frames.
- LRU is a stack algorithm.
  The set pages in memory always include n most recently
  referenced pages.

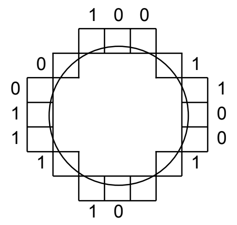
Demand Paging (8): LRU Approximations

1. Additional reference bits algorithm:
   - Each page has a reference bit and an 8-bit register.
   - At a regular interval (R-bit, register) is shift to right.
   - A page with the smallest register value is the LRU page.

Demand Paging (9): LRU Approximations

2. Clock algorithm:
   - Also called "second chance" algorithm.
   - Keep "use" bit for each page frame.
     Hardware sets the appropriate bit on every memory reference.
     The operating system clears the bits from time to time.
     OS uses the bit to figure out how often pages are being referenced.
     Also called the "reference' bit.
   - Algorithm: OS circulates through the physical frames cleaning use bits until one is found that is zero. Use that one.

Demand Paging (10): LRU Approximations



3. Enhanced clock algorithm: Some systems also use a "dirty" bit to give preference to dirty pages.
   - More expensive to throw out dirty pages:
     Clean ones need not be written to disk.
   - Problem: What if clean pages are frequently accessed?
     Code pages.

Demand Paging (11): Clock Algorithm

- What does it mean if the clock hand is sweeping very fast?
  - Not enough memory.
- BSD Unix uses the Clock Algorithm: Sun OS

  "`vmstat`" command gives info.

  "`sr`" — pages scanned by clock algorithm, per-second.

  `vmstat -s`:

  292853 pages examined by the clock daemon.
  6 revolutions of the clock hand.
  127878 pages freed by the clock daemon.

  `vmstat 5`:

  VM activity every 5 seconds.

Demand Paging (12): Replacement Style

- Three different styles for replacement

  (1) Global replacement

  All pages from all processes are lumped into a single replacement pool.

  Each process competes with all the other processes for page frames.

  (2) Per-process replacement

  Each process has a separate pool of pages.

  A page fault in one process can only replace one of the process's frames.

  This relieves interference from other processes.

  (3) Per job replacement

  Lump all processes for a given user into a single replacement pool.

Demand Paging (13): Replacement Style

- In per-process and per-job replacement, must have a mechanism for (slowly) changing the allocations to each pool.
  - Otherwise, can end up with very inefficient memory usage.
- Global replacement provides most flexibility, but least "pig protection"
- How do these compare to economic systems ?

Demand Paging (14): Frame Buffering

- Commonly keep a free frame pool
  - (1) When a page fault occurs, a frame is selected.
    - But the desired page is written into a page in the pool.
  - (2) Maintain a list of modified pages.
    - When the swap device is idle, they are written out.
    - Process 2 (pagedaemon) doses the job.
  - (3) Old pages are freed to the pool. But they may be reused when requested.
    - Used in VAX to offset the problem of FIFO.

Thrashing (1)

- Thrashing: consider what happens when memory gets overcommitted.
  - Suppose there are many users, and that between them their processes are making frequent references to 50 pages, but memory has 49 pages.
  - Each time one page is brought in, another page, whose contents will soon be referenced, is thrown out.

Thrashing (2)

- The system will spend all of its time reading and writing pages.
  - It will be working very hard but not getting anything done.
- Average memory access time equals disk access time.
- Illusion breaks: Memory access will look as slow as a disk rather than disks being as fast as memory.
  - Effective memory access time: $t_{eff\_acc} = pt_{mem\_acc} + (1-p)\,t_{fault}$
- Thrashing was a severe problem in early demand paging systems.

Thrashing (3)

- Thrashing occurs because the system doesn't know when it has taken on more work than it can handle. LRU mechanisms order pages in terms of last access, but don't give absolute numbers indicating pages that mustn't be thrown out.
  - What's the human analogy to thrashing ?
    - Too many courses ? – What's the solution ? Drop one?

Thrashing (4)

- What can be done about thrashing in computer systems?
  - If a single process is too large for memory, there is nothing the OS can do.
    - That process will simply thrash.
  - If the problem arises because of the sum of several processes:
    - Figure out how much memory each process needs.
    - Change scheduling priorities to run processes in groups whose memory needs can be satisfied.
    - Shed load.

Working Sets (1)

- Working Sets
  - A solution proposed by Peter Denning: An informal definition:
    - The collection of pages that a process is working with, and which must thus be resident if the process is to avoid thrashing.
  - The idea is to use the recent needs of a process to predict its future needs.
    - Choose t, the working set parameter
      - At any given time, all pages referenced by a process in its last t seconds of execution are considered to comprise its working set
    - A process will never be executed unless its working set is resident in main memory
    - Pages outside the working set may be discarded at any time

Working Sets (2)

- Problem with the working set
  - OS must constantly update working set information
    - What pages have been accessed in the last τ seconds?
  - One of the initial plans was to store some sort of a capacitor with each memory page. The capacitor would be charged on each reference, then would discharge slowly if the page wasn't referenced. t would be determined by the size of the capacitor.
  - This wasn't actually implemented.
    - One problem:
      - We want to separate working sets for each process. Need a capacitor that only discharges when the process is running. What if a page is shared ?

Working Sets (3)

- Actual solution: Take advantage of use bits.
  - OS maintains idle time value of each page:
    - Amount of CPU time received by process since last access to page.
  - Every once in a while, scan all pages of a process
    - For each use bit on, clear page's idle time.
    - For use bit off, add process CPU time (since last scan) to idle time.
    - Turn all use bits off during scan.
  - Scans happen on order of every few seconds.
- In Unix, t is on the order of a minute or more.

Working Sets (4)

- Other questions about working sets and memory management in general
  - What should t be ?
    What if it's too large ?
    What if it's too small ?
  - What algorithm should be used to determine which processes are in the balance set?
  - How do we compute working sets if pages are shared?
  - How much memory is needed in order to keep the CPU busy?
    Note that under working set methods the CPU may occasionally sit idle even though there are runnable processes.

Working Sets (5): Resident Set

- Approximation of working set in VAX VMS:
  - Resident set of a process is a set of its pages resident in physical memory
  - Each process has a resident set limit
    - Max no. of physical pages that can be assigned to that process
  - Resident set limit is dynamically adjusted based on page fault rate
    - No need for costly computation of working set information

Working Sets (6): Resident Set

- Thrashing avoidance with resident set
  - When a process is made inactive, all of its pages in resident set are swapped out to the swap space
  - When a process is made active, all pages in its resident set are loaded
    - Each process has a resident set list in its context

Working Sets (7): Balance Set

- Working sets are not enough by themselves to make sure memory doesn't get over committed. We must also introduce the idea of a balance set:
  - Balance set is a collection of processes whose working sets are resident in physical memory
  - If the sum of the working sets of all runnable processes is greater than the size of memory, then refuse to run some of the processes (for a while).

Working Sets (8): Balance Set

- Divide runnable processes up into two groups:
  Active and inactive. When a process is made active, its working set is loaded. When it is made inactive, its working set is allowed to migrate back to disk.
- The collection of active processes is the balance set.
- Some algorithm must be provided for moving processes into and out of the balance set. What happens if the balance set changes too frequently? (Thrashing)

Balance Set and Swapper

- In UNIX, swapper (process 0) chooses balance set.
  - The first process created by OS.
  - System process with no user context.
  - It executes a routine called `sched()`.
  - It is normally asleep and awakened once every sec.
  - It swaps out a process, if free mem is less than `t_gpgslo`.
  - It calls `as_swapout()`, which cycles through each segments.
  - To swap in a process, swapper swaps in its `u_area`.
    When the process eventually runs, it will fault in other pages as needed.

Thrashing and Working Sets

- The issue of thrashing may be less critical for workstations than for timeshared machines:
  - With just one user, he/she can kill jobs when response gets bad.
  - With many users, OS must arbitrate between them.
- Current trends - Larger physical memory.
  - Page replacement algorithm is less important.
    Hopefully, it will rarely get invoked.
  - Less hardware support for replacement policies.
    Software emulation of use and dirty bits.
  - Larger page sizes
    Better TLB coverage.
    Smaller page tables.
    Fewer pages to manage.

Thrashing and Working Sets

- Current trend – Larger virtual address spaces.
  - 64 bits with 4K page: Max page table size 64K TB.
  - Sparse address spaces.
  - Inverted page tables.
    Hash table VA $\rightarrow$ PA.
    Scale with the size of physical memory.
- Current trend – File I/O using the virtual memory system.
  - Memory mapped files.
    Uses VM system for file caching.
    Page fault handing for reads/writes.
  - Make page replacement interesting again.
  - More sequential bahavior.

Paging for Large Address Spaces

1. Hierarchical Paging
2. Hashed Page Tables
3. Inverted Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables.
- A simple technique is a two-level page table.

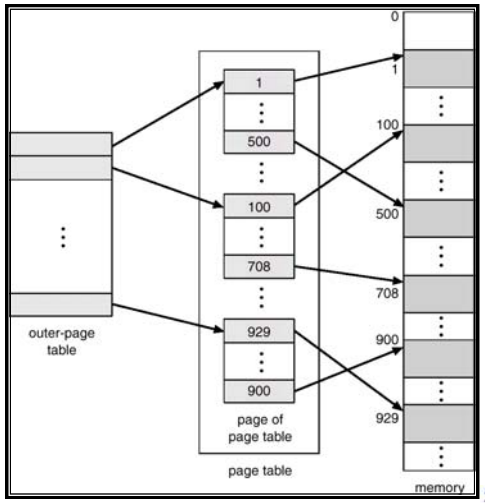
Two Level Paging

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number.
  - a 10-bit page offset.
- Thus, a logical address is as follows:

page number		page offset
p_i	p_2	d
10	10	12

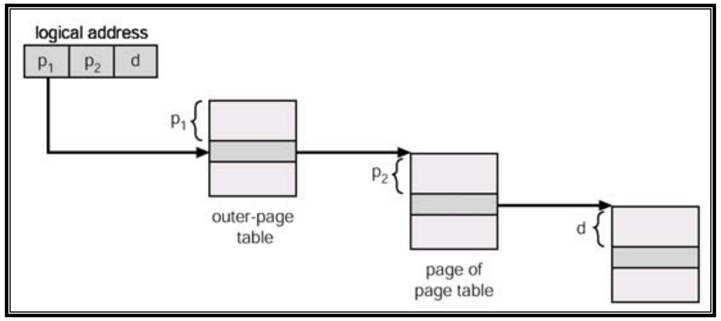
where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table.

Two Level Page Table



Address Translation Scheme

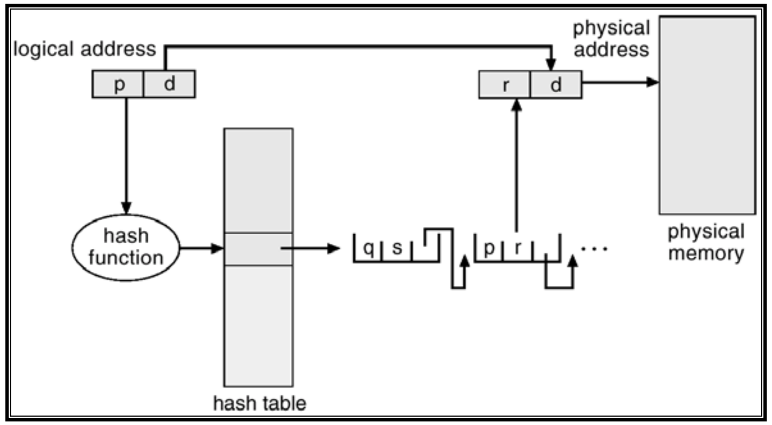
- Address translation scheme for a two-level 32-bit paging architecture



Hashed Page Tables

- Common in address spaces > 32 bits.
- The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

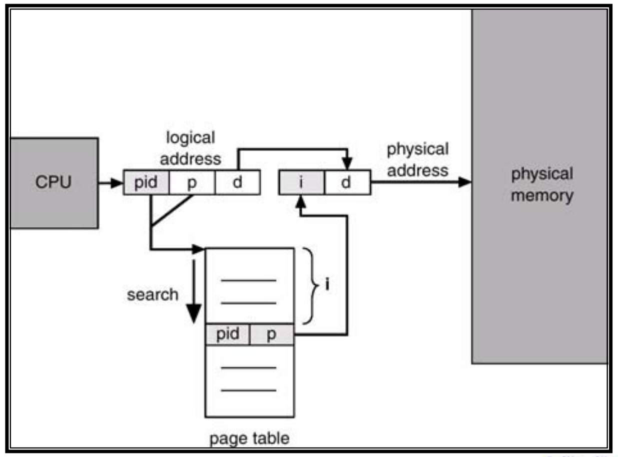
Hashed Page Table



Inverted Page Table

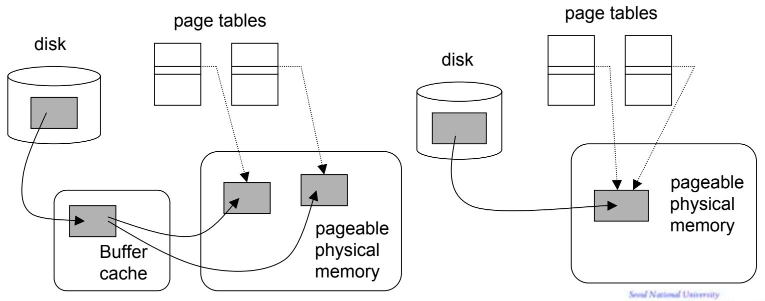
- One entry for each real page of memory.
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
- Use hash table to limit the search to one — or at most a few — page-table entries.
- Each process still keeps its page table.
  - Disk address of an invalid page

Inverted Page Table Architecture



Memory Mapped Files

- paddr = mmap(addr, len, prot, flags, fd, offset)
  - Map [offset,offset+len) onto [paddr,paddr+len).
  - fd is file descriptor returned by open().
  - Reduces the number of system calls and data copies for file access.



page tables
disk
Buffer cache
pageable physical memory
page tables
disk
pageable physical memory