The UNIVERSITY of VERMONT

Benjamin Fenster
CS-201 (Arslan)
11 December 2005

Notes – Memory Management

I. Binding of Instructions & Data to Memory
   a. Compile Time
      i. Address is known a priori (before execution)
      ii. Absolute addresses
      iii. This means the program has to be loaded at a particular address
   b. Load Time
      i. "Relocatable" code
      ii. Starting address is variable
   c. Execution Time
      i. Binding is delayed until runtime if the process can be moved from one memory segment to another.
      ii. Requires, obviously, some extra work.

II. Memory Management Unit (MMU)
   a. Maps logical addresses to physical addresses
   b. Parts of the process (or the entire process perhaps) can be swapped in and out of memory during execution.
   c. Thus many processes can share the same user space (meaning logical memory space is potentially much larger)
   d. Contiguous allocation
      i. Allocate whole chunks of memory contiguously.
      ii. When there's no singal chunk big enough (but there are enough little holes) shift what's in memory ("compaction") to make room.
   e. Best Fit
      i. Find a hole greater than or equal to the size of the incoming process
      ii. Among those holes, pick the one with the smallest size
      iii. Minimizes the number of tiny chunks of space between processes
         1. Internal Fragmentation: Free space within the chunk allocated to a process
         2. External Fragmentation: Total free space between partitions
         3. We measure efficiency by internal/external fragmentation
   f. Worst Fit
      i. From candidate holes, select the one with the largest size.
      ii. Leaves the most free space left over.
   g. First Fit: Select the first candidate hole found
   h. Disadvantage to contiguous allocation: the whole process has to be swapped in or out, whereas the whole thing may not be needed at once.

III. Paging
   a. Allocation doesn't need to be contiguous.
   b. Logical address has two parts: [p|d].
      i. p is an index to the page table.
      ii. From the page table we get the frame number f (page sizes are equivalent to frame sizes)
      iii. d is the offset
      iv. [f|d] together give a physical memory address.
   c. For a process, logical space is contiguous. It refers to an address in logical space, which is then translated to a physical address.
   d. The page table maps logical pages to physical frames.
   e. The OS has to keep track of free frames and allocate them when new processes are created or when an existing process needs more memory.
   f. This raises the degree of multiprogramming since the OS doesn't need to find free space, just picks some free frames and updates the page table.
   g. The page table is stored in associative memory – special hardware

i. Can check the desired page number to values in all registers in parallel (in one clock cycle can find the desired frame number)
ii. Normally the entire page table is in memory, and the page number is just an index in the array.
iii. Associative memory is used when the page table is cached.
iv. Called "Translation Lookaside Buffer" or TLB.
v. So to find a physical address, check in the TLB (hit gives frame number in one step). If no hit, access memory for the full page table.

h. Effective Memory Access Time (EAT)
i. If hit ratio = α (1 microsecond memory cycle)
ii. Associative memory lookup time = ε
iii. EAT = $\varepsilon + \alpha$ (1 μs) + (1 - α)(2 μs)
iv. Always try TLB, so include ε.
v. Then with probability α it will take only one microsecond because we'll just read directly from the right memory address. With probability (1 - α) we'll have to read the page table and then go read the data so it'll take twice as long.
vi. EAT = $\varepsilon + 2 - \alpha$
vii. Need a fast TLB, and want a high hit ratio to minimize the memory access time.

i. Valid/Invalid Bit: Tells whether a page is in memory.

j. Two-Level Paging
i. Typical page size is 4k, so for large processes the page table itself can be very large. Page out the page table itself!
ii. Now logical address is $[p_1|p_2|d]$
1. Use p_1 to get to p' (means, "get the 5th page table")
2. Use p_2, p' to get the index to the page table ("within that table, get the 10th frame")
3. Then finally get the frame number.
iii. So now only a subset of page tables need to be in memory.

IV. Types of Page Tables

a. Hashed
i. This is another solution for large page tables.
ii. Logical address p hashes into the frame number.
iii. This means there may be collisions, so multiple page-frame pairs will be stored in a chain.

b. Inverted
i. Have one table system-wide including process IDs.
ii. Each individual process could have as many (or more) pages as there are frames in memory..
iii. The fth entry in the table gives PID and page number for frame f.
iv. Given a logical address, we'd have to look through the entire table – a big disadvantage.
v. So it saves memory, but increases search time to linear.

V. Sharing Pages

a. Paging allows processes to share code and data.
b. Several editor processes may all use the same code with different data.
c. Each process has three logical pages of editor code, but It turns out "editor 1" maps to the same frame of memory for all processes (likewise for "editor 2" and "editor 3") but the data pages for each process map to distinct frames.
d. What's shared can't be modified without introducing a synchronization mechanism.
e. Note that each process has a different *context* even with some or all of the code shared.
f. That's not possible with an inverted page table.

VI. Segmentation

a. Paging: split memory into equal-sized chunks.
b. Segmentation: split it into groups
c. A program is a collection of segments

d. A segment is a logical unit (procedure, object, stack, method, main program)
e. Now a logical address is [s|d]
f. At what segment is it stored, and at what offset?
g. From the segment table, obtain the limit and base
    i. Test d < limit
    ii. If false, causes an addressing error (trap)
    iii. If yes, the physical address is base + offset
h. Sharing Segments
    i. Editor code may be at address 43062 in memory
    ii. Then segment 0 for several different processes mahy be the editor main program, so each process's segment table will map to limit 25286|43062
    iii. A Problem
        1. Each segment may be very large
        2. Perhaps we could apply paging to each segment.
        3. Add s to the base address where the segment table is stored; use that as an index to get the segment length and page table base.
        4. Logical address is now [s|p|d]
        5. In the page table, lookup page p (offset from the page table base) to get frame f
        6. Now finally you'll have [f|d] to read a word from memory.
        7. Thus we can now page out portions of segments. Phew!

VII. Page Faults
a. When a program requests a page not actually in memory it's a page fault
b. Steps: Demand Paging
    i. Get load M command
    ii. Search page table; assume the valid/invalid bit is "invalid"
    iii. Causes a trap to the OS (a software interrupt)
    iv. Find a free frame. If none exists, swap out a frame that's in use
    v. Swap the desired page from disk to memory
    vi. Update the page table, set the valid/invalid bit to "valid"
    vii. "Success" depends on the system's ability to capture locality of execution
c. Performance
    i. Effective Access Time (EAT)
    ii. Let p, $0 \leq p \leq 1$ be the page-fault rate.
    iii. EAT = (1 – p) (memory access time) + p (pf overhead + restart overhead + swapping in/out overhead)
    iv. Note that the instruction is restarted after the OS trap is finished.
    v. The second part is obviously much more significant.
d. Two Benefits to Demand Paging
    i. Copy on Write
        1. If a page is modified, create a copy of that page.
        2. After a fork, if the child modifies memory, needs its own copy (otherwise it can share the same page)
    ii. Memory-Mapped Files
        1. Have part of the file in memory
        2. References can be satisfied in memory
        3. Referencing data in the file is equivalent to a memory reference if the requested part of the file is in memory
e. Selecting a Victim
    i. When there's a page fault and no free frames, need to select a victim page to be swapped out.
    ii. If a page is unmodified, don't bother saving it to disk. Have a dirty bit indicating whether each page has been modified since it was last loaded.
    iii. Still need to select a victim.
    iv. Ideally want a page that won't be used or will be used furthest in the future.

v. Of course, we don't know the future so we need an algorithm to predict it
vi. A reference string indicates requested page numbers ("123412412345")
vii. FIFO
    1. First page brought in is the first page swapped out
    2. The assumption is that once a page is brought in and used it won't be needed anymore.
    3. For "123412412345" will have 9 page faults, 3 normal reads
    4. This isn't the best algorithm.
viii. Least Recently Used
    1. Select as a victim the page used farthest in the past
    2. Every time you access a page, update its timestamp. Select the page with the oldest timestamp as a victim.
    3. When the number of frames is much smaller than the number of pages, it's better to use a double linked list.
        a. After every reference, add to the end of the list.
        b. If the same page is already in the list, remove it.
        c. If a victim is needed, use what's at the beginning of the list
    4. Another Implementation
        a. When a page is brought in or referenced, set a reference bit = 1
        b. Periodically set all reference bits to 0.
        c. When you observe a zero reference bit it means the page hasn't been accessed since the last reset.
ix. Second Chance Page Replacement
    1. Have a circular list of pages that are in memory
    2. "Next Victim" pointer points to a page with reference bit = 1.
    3. When it's selected, advance to the next page and reset the reference bit.
    4. If all reference bits were 1s, you'd just make a full circle and come back to the original page (with reference bit 0 still)
    5. When you encounter a page with a zero bit, you can remove it.
    6. This gives each page two chances to be accessed, so you won't pick a page that's been used since you last went through the loop
x. Other Algorithms
    1. Least Frequently Used
    2. Most Frequently Used
    3. Keep a count of references and select a victim with the smallest (or largest, respectively) count.

f. Considerations with Multiple Processes
    i. How many frames for each process?
        1. Could allocate a fixed number for all processes
        2. Could use prioritization and allocate pages proportional to priority
    ii. Selecting a Victim
        1. Could select only within a process's own pages
        2. Could give the ability to select globally, so another process's pages may get victimized.

g. Thrashing
    i. The system gets bogged down swapping pages in and out.
    ii. Σ size of locality of P_i > total memory size
    iii. Despite multiprogramming, processes are still all waiting for I/O

h. Working Set
    i. The working set for a process is a window of a fixed number of instructions, denoted Δ (a fixed number of page references)
    ii. For process p_i, wss_i = working set of process p_i
    iii. Thrashing occurs when $\Sigma\ wss_i$ > memory size (m)
    iv. For systems using the working set model, Δ is a parameter
        1. If Δ is infinitely large, the entire process is one working set

2. If Δ is small, it's impossible to capture true locality.
3. It needs to fall somewhere in the middle

v. Example
1. Pages: 2 6 1 5 7 7 7 7 5 1 6 (t_1) 2 3 4 1 2 3 4 (t_2)
2. Δ = 10 (meaning the last ten references are in the working set)
3. ws(t_1) includes {1, 2, 5, 6, 7}
4. ws(t_2) includes {1, 2, 3, 4, 5, 6}

vi. Paging with the Working Set Model
1. When a page falls out of the working set, it becomes a candidate to be swapped out.
2. New pages entering the working set get swapped into their place

vii. Implementing Working Set
1. If Δ is 10,000, could have an interrupt every 5,000 units of time.
2. Have two reference bits for each page
3. On each interrupt, set one of the bits to zero
4. If a reference bit is not zero, that page is in the working set
5. When a page is brought into memory, set 11.

i. Target Page Fault Rate
i. If the rate of page faults for a process is too high, allocate more frames
ii. If the rate is low enough, decrease the number of frames allocated (you're wasting resources other processes might need)

j. Prepaging: Instead of waiting for the first few references, load some pages on process creation and perhaps avoid some page faults at the beginning.