Lectures 10 and 11: Memory Management (Feb 17 and 22, 2005) Yap

April 5, 2005

1 ADMIN

- READING GUIDE FOR Chap.4:

  Focus on Paging (§4.3) and Page Replacement Policy (§4.4). Skip §4.3.3, §4.3.4, §4.4.7.

- Today, we discuss virtual memory, paging mechanism, memory allocation policy, page replacement policy.

  We distinguish between "policy" and algorithms which tries to implement these policies. Tanenbaum do not make this distinction.

2 Review

- Q: What the two ways to return from a setjmp(), and how can we distinguish between them (if we so choose)?

  A: The original call to `setjmp(jmp_buf env)` returns 0. The other way to return from setjmp() is via a longjmp() which can specify the return value. To distinguish them, make sure that longjmp() return a non-zero value v: `longjmp(jmp_buf env, int v)`.

3 Intro

- Memory devices have two basic metrics: capacity (or size) and speed. E.g.,

  RAM is 500 MB and speed is 5 ns.

  Cache is 1 MB and speed is 2 ns.

  Disk is 80 GB and speed is 5 ms.

- Two memory devices exhibit a trade-off property if one has better speed but smaller capacity.

  E.g., the above 3 devices have mutual trade-offs.

- In a typical computer, we have a set of memory devices with exhibit such trade-offs.
- We can arrange these devices into a **memory hierarchy** based on decreasing speed.

  Typically: registers, cache, main memory (RAM), disk, tape.
- Managing these different units to optimize speed is the problem of memory management
- In the real world, memory devices we have two other metrics: price and physical size. The tradeoffs here are more complicated.

4 Basic Memory Management (Allocation Problem)

- The basic problem is how to allocate the main memory for processes.

  We had seen simple examples of this in the first 2 lectures (e.g., through the use of base register and limit register).
- As processes come and go, we need to solve two functions: free memory and allocate memory.
- What data structure do we need to enable these two functions?

  Memory is divided into blocks. Each block is used or free.

  Two linked lists: free list and used list.

  REMARK: These two lists can also be merged, or cross-linked.
- Free Memory Function:

  Return the memory to free list.

  Merge with adjacent free blocks if possible.
- Allocate Memory Function:

  Find a free block that is large enough.

  Break up the block into a used list and return the rest into the free list.
- But which free block to choose? This is the **Memory Allocation Problem**.
- Issues:

  Speed of implementation.

  Fragmentation of free space.
- Allocation policies:

1. first fit ("first" by some criterion).
2. best fit (or "smallest fit").
3. worst fit (to avoid breaking up into small useless chunks)
4. Quick fit (a separate pool of "typical size" blocks)
5. any fit (you can implement this by first fit)

REMARKS:
Best and Worst fit are slower than first fit because these are global criteria.

5 Virtual Memory and Paging

- Virtual vs. Physical Address Space
- Virtual memory is achieved by paging.

  Basic parameter here is **page size**.
- Divide virtual memory into **virtual pages** (of the same page size).

  Divide physical memory into **page frames**, of page size.

  Allocate virtual pages to page frames. Keep tract of this using a **page table**

  Need a method to translate from virtual address to physical address (usually by MMU).
- If you address a location not in any page frame, then
  1. Page fault (trap)
  2. Choose a page frame to replace (EVICTION POLICY, see below)
  3. If M-bit is 1, write out the page frame
  4. Read in the new page
- TABLE ENTRY:

  SIMPLIFIED: [present-bit, page frame number]

  MORE REALISTIC: [Cache-enable-bit, R-bit, M-bit, Protect Permission, Present-bit, page frame number]
- Example:

  Virtual memory is 64 KB. Main memory is 32 KB. Page size is 4 KB. Hence, 16 virtual pages, 8 page frames.

  – Virtual address: [virtual page #(4 bits), offset(12 bits)]

  – Physical address: [page frame #(3 bits), offset(12 bits)]

  – Page table of size 16 where each entry is a pair of the form

  (Present Bit, 3-bit frame number). If Present Bit is 1, then the frame number of this page is used. If Present Bit is 0, then ignore the second field. WHY IS THIS POSSIBLE?

- Two Issues:
  1. Page Table can be large! A typical address space is 32-bits, so the virtual memory size is 32 GB. With 4KB page size, this is 1 million pages. This table is kept for each process!
  2. Every memory reference requires an address translation.

  SOLUTIONS: §4.3.3 TLB (Translation Lookaside Buffer), and §4.3.4 Inverted Page Table (skip both topics).

- Multilevel page tables – see example in book...discuss this topic.

6 Page Eviction Policies/Algorithms

- Which current page should we evict in a page fault?

  – exploit locality of reference in code

  – translate this into various policies, such as LRU

- EVICTION TECHNIQUE FOR AN ideal program: we run it twice. The first time, we note all its page referencing pattern. Based on this, we can design the ideal eviction function!

  This assumes our program makes page references in a fully determined manner (if the references depend on the program input, then the input must also be fixed).

- Two Page Statistics: R-bit and M-bit.

  [To be continued]

  1. **Reference Bit**: R-bit is set when the page is referenced
  2. **Modified Bit**: M-bit is set when the page is modified
  3. NOTE: when the M-bit is set, the R-bit must be also be set.
     This suggest that there are only 3 possible values for these 2 bits.
  4. But we shall *periodically* unset the R-bit. This allows 4 possible values for these 2 bits.

- Let (R, M) be viewed as a binary number between 0 and 3.

  The **Not Recently Used** (NRU) Policy says that we should remove a page of the lowest category.

  E.g., if category 2 page should be removed before any category 3 page.

  REMARK: what are the possible transitions among categories? (Only 0 to 1 is not possible!)

- **FIFO** Policy: remove the page that was first to be paged in.

  This policy is not much used because it does not look at the referencing pattern.

- **Second Chance FIFO** Policy:

  1. 1. Maintain the R-bit.
  2. 2. To remove a page, we loop through the front page of the queue: if the front page has R-bit set, move it to end of queue and remove its R-bit. Otherwise, remove that page.
  3. REMARK: An efficient way to implement this policy is to maintain a "front pointer" to the queue, viewed as a circular list.
  4. REMARK: If the R-bit of the page is set again, then the page may have more than two chances!

- **Least Recently Used** (LRU) Policy.

  How different from NRU? We are more precise about "recent" – we want the "least" not just a level.

  Maintain a hardware counter (64 bits)

  Store a 64-bit value with each page.

  Each page reference will update the 64-bit value with the counter value.

  PROBLEMS: What if the counter resets?

- Modified implementation of LRU:

  Maintain a boolean matrix (the ith row and ith column corresponds to page i). Initialized to 0.

  When the ith page is referenced, set the ith row to 1, then the ith column to 0.

  To pick the LRU page, we just view the ith row as a binary number, and pick the row with smallest value for eviction.

  REMARK: Skip §4.4.7 (software LRU algorithms)

- **Working Set Paging**

  1. This is a form of paging in which we maintain for each process a "working set".
  2. We fix a parameter $k > 0$ and maintain for each process the set of pages that is referenced in the last k memory references.

     Let $w(k,t)$ be the set of pages most recently referenced before time t. CLEARLY, we have

     $$1 \leq w(k,t) \leq k$$

     and $w(k,t) \leq w(k',t)$ for all $k \leq k'$.

     BUT Tenanbaum [p.223] argues that $w(k,t)$ is not that sensitive to k when k lies in some reasonably large range.

     REMARK1: In figure 4-20 [p.223], time t is fixed!

3. Q: When a process is swapped, what do we do with its old pages?

   A: We swap it out, and do "prepaging" to restore the "working set" of restarted process.

   This is the FIRST use of working set.

4. The SECOND use of working set is in page replacement policy:

   When a page is to be evicted, pick one not in the working set.

- How to maintaing the "working set" for a process?

  BASIC WORKING SET REPLACEMENT algorithm:

  1. FIRST APPROXIMATION: use **virtual time** of last reference should be NOT more than τ time units earlier than current time. Thus, τ is a parameter used instead of "last k references".

     VIRTUAL TIME of a process is the time between its start and current real time, MINUS the time it was not running.

  2. We have the R- and M-bits, periodically clearing the R-bit as before.

  3. At page fault, the page table is scanned.

  4. For EACH page:

     A. If R-bit is set, the page is not candidate for removal. Instead, the current virtual time is written into the "TIME OF LAST USE" field of page table.

     B. If R-bit is unset, the page may be candidate for removal.

     B1. If the current time is MORE than τ units since TIME OF LAST USE, the page is removed from working set (and new page loaded).

     B2. If the age is less than τ, it is spared (for now) but the OLDEST TIME LAST USED is updated.

     C. If Step B did not find any candidate for eviction at the end of algorithm, we remove the one with the OLDEST LAST TIME USED. If ALL the pages have their R-bit set, we randomly pick one for removal.

- **WSClock Algorithm**:

  The above WS algorithm has to scan the page table, and is slow. Clock implementation of above.

- Belady's Anomaly:

  Consider the page request sequence

  $(0, 1, 2, 3; 0, 1, 4, 0; 1, 2, 3, 4)$

  where the FIFO page replacement algorithm is used. If the FIFO queue has size 3, we get 9 page faults. If the FIFO queue has size 4, we get 10 page faults.

  Stack Algorithms (e.g., LRU) will not have this anomaly.

- IMPLEMENTATION ISSUES:

  We will skip for now, but discuss one particular problem.

  In the page table, we do not store the disk address of a page. When we swap a page out of main memory, where do we know to put it? ANSWER: use a swap space in disk, the size of the core image. Then we only need to keep the offset of this swap space in the PROCESS table.