Nov 20, 2006
Lecture 13: Virtual Memory

September 5, 2007

1 ADMIN

We finish up our discussion of Memory Management and will start Virtual Memory next. Please start to read Chapter 10.

2 REVIEW

- Q: What are some solutions to huge page tables?

  A: (a) Hierarchical paging (see p.340 of text, §9.4.4.1)

  (b) Hashed page table (p.343, §9.4.4.2)

  (c) Inverted page table (p.344, §9.4.4.3)

3 Demand Paging

- What is the difference between paging and demand paging?

  The answer lies in a basic assumption:

  In standard paging, each process is either in main memory or it is not. We may introduce swapping, to move processes in and out of main memory.

  In demand paging, we assume that each process normally resides in secondary memory called **backing store** or **swap space**. We only bring in the necessary pages for it to execute.

- Swapper versus pager.

  In paging, we have the concept of swapping. But now, we have the concept of pager, where only certain pages are swapped in, not the entire process.

- What new issues arise?

  Like all caching techniques, we must now have a map (or table) to tell us which pages of a process is actually loaded in main memory.

  We also need some eviction policy.

  In addition, pages can now be invalidated. So there are additional "valid flags".

  The demang paging table is (as usual) per process.

- REMARK: we can also combine demand paging with demand segmentation. The latter introduces a slight complication in that segment sizes are different. This is not treated in our text.

- FUNCTIONS of the pager:

  – when a process is first swapped in, we may want to load some initial pages. One option is to load only the first page, or even no pages.

  – need a table to show which pages are in main memory and where are the pages located in secondary memory

  – when a process access a page not in main memory, it generates a **page-fault** (a trap).

  – the page fault trap is usually handled by hardware (see FIG 10.6 on p.370).

    – check page table to see if the reference is valid. If not, terminate process.
    – find a free frame
    – may have to schedule a disk write to evict current page in that frame
    – schedule a disk read to bring page into free frame
    – when disk read completed (interrupt), update the page table
    – resume from trap

  – IN MORE DETAIL:

    – Trap to OS
    – Save user registers and process state
    – Determine if interrupt was page fault
    – Check if page ref is legel
    – Issue a read from disk to free frame
    – Switch to some other process
    – Receive interrupt from disk read
    – Save registers and state of other process
    – Check if interrupt is from disk
    – Update page table with new information
    – Wait for process to be scheduled again
    – Restore user registers and state, and page table

  This "page fault time" (PFT) is estimated to be 8 ms.

  – Crucial requirement (p.371): how to restart after a page fault. E.G. ADD A,B,C

    – Fetch instruction (ADD)
    – Fetch A
    – Fetch B
    – Add A and B
    – Store in C

  We can fault in any one of these. Say we fault at the last step. We have to restart all over!

  The problem is worse in complicated instructions (e.g., copying an entire block of memory to a new location that might overlap with original location).

  Some solutions:

  (1) get all the needed pages before starting to execute the complicated instruction

  (2) Use temporary registers to store partial computation.

- PERFORMANCE:

  – p is probability of page fault

  – MAT is memory access time (10 to 200 ns)

  – PFT is page fault time (8-10 ms) (see above)

  – EAT is **effective access time**

  – FORMULA: $EAT = (1-p) \times MAT + p \times PFT$

  – e.g., if we page fault once every 1000 pages, $MAT = 200ns$ and $PFT = 10ms$ then in seconds:

  – Disk I/O for swap space is usually faster than for file system I/O because of larger block allocation.

- Copy-on-Write:

  When we fork(), the semantics calls for a copy of the parent's image. This may be wasted work if the child does an "exec()" right after!

  So continue to let the parent and child to share pages, until either one tries to write. At this point, a copy must be made to either child/parent's image space.

  So these pages must be tagged as "copy-on-write" in their page table.

  This is used in Windows XP, Linux, Solaris.

4 PAGE REPLACEMENT METHODS

- PAGE REPLACEMENT algorithms:

  Which frame to free up?

- **reference string**: a sequence of page numbers, as an abstraction of a sequence of address references in a computer.

  Can be generated by some algorithm (e.g., randomly) or taken from real computer runs.

  In the following, we use this reference string:

$$S_0 = (7012, 0304, 2303, 2120, 1701)$$

- FIFO Algorithm

  Example of using 3 frames on reference string S_0:

  Produces 15 page faults!

  E.g., the frames after each reference is as follows:

$$7(7)0(70)1(7^{*}01)2(20^{*}1)0(20^{*}1)3(231^{*})0\ldots$$

  Note the the first three references causes a page fault each time.

- Belady's Anomaly

  FIFO has the curious property: sometimes, by increasing the number of frames, you can increase the number of page faults!

  In the case of FIFO, this curious property is called Belady's Anomaly since he first observed this behavior. Consider the following reference string:

$$S_1 = (1234, 1251, 2345)$$

  If the number of frames is 3, then we get 9 page faults:

$$1(1)2(12)3(1^*23)4(42^*3)1(413^*)2(4^*12)5(51^*2)123(532^*)4(534)5$$

If the number of frames is 4, we get 10 page faults:

$$1(1)2(12)3(123)4(1^*234)125(52^*34)1(513^*4)2(5124^*)3(5^*123)4(41^*23)5(452^*3)$$

- : Show that when we increase from 1 frame to two frames, FIFO does not exhibit Belady's anomaly.

  Question: can Belady's anomaly occur (for FIFO) in going from 2 frames to 3 frames?

  Question: can Belady's anomaly occur (for FIFO) in going from n frames to $n+2$ frames?

- OPTIMAL PAGE REPLACEMENT (OPT)

  "Replace the page that will not be used furthest into the future"

  Let us apply this algorithm to S_0 using 3 frames:

  this produces 9 page faults, a great improvement over the 15 page faults caused by FIFO.

  Problem: we cannot implement this algorithm as we cannot look into the future!

- LRU PAGE REPLACEMENT

  This algorithm says: the OLDEST Referenced Page should be replaced. We must associate a "last time of reference" with each page, and update this value each time we reference a page.

  Apply this algorithm to S_0, we obtain 12 page faults.

  ...do the simulation...

  This is not as good as OPT, but better than LIFO.

  Monotone property: we say that a page replacement scheme is monotone if increasing the number of frames does not increase the number of page faults for any reference string.

  THEOREM: LRU is monotone.

  Proof: Initially, we use n frames. Using a fixed reference string, let P_i be the set of pages in the cache after the ith reference. Suppose we increase the number of frames to $n+1$. and now let Q_i be the set of pages in the cache after the ith reference.

  Note that P_i contains the n pages that has the smallest time stamps. Similarly Q_i contains the $n+1$ pages that has the smallest time stamps. This implies that

  $$P_i \subseteq Q_i.$$

  Thus, Q_i will never lead to a page fault when P_i does not page fault. QED

- IMPLEMENTATIONS OF LRU Policy

  We need hardware support to implement the LRU algorithm. But since hardware is expensive, we often achieve only an approximation to the true LRU Policy.

  (1) Instead of using "real time", we can use a counter. Each page reference will increase this counter, and we store the value of this counter with the page, as its time-stamp. To use this information for LRU, we still need to search through the pages to find the one with the smallest time-stamp for replacement.

  (2) Instead of using time stamp, we just put the pages into an ordered list, with the oldest referenced page at the front. Each time we reference a page, it is moved to the front. THIS IS THE MOVE-TO-FRONT Rule.

  (3) Move-to-front hardware is not easy to implement. A circular queue is easier to implement. This circular queue can easily be used for a FIFO queue.

  The SECOND CHANCE ALGORITHM is based on exploiting this mechanism.

The frames are put in a FIFO queue, and each frame has a **reference bit**. Initially, when the frame is loaded, the bit is set to 1. Each time we reference a page, the bit is set to 1. When we need to evict a page, we consider the front of the queue: if its bit is 0, we replace it. If itss bit is 1, we set it to 0 and put it at the end of the queue again.

- On **Policy** Versus **Mechanism** Versus **Implementation**

  – Sometimes we have a policy that can only be approximated. We can provide mechanisms to help its implementation. The final concrete solution is the implementation (or algorithm).

  – Of course, this is very similar to how we govern and order human society as well.

  – We had seen this already in job scheduling: SJF is optimal but not really achievable. So we try to approximate this. We see a similar phenomenon here.

  – E.g.,

  - Policy is LRU
  - Mechanism(s) is the REFERENCE BIT and the CIRCULAR QUEUE.
  - Implemenation is Second Chance Replacement Algorithm.

5 Header Files

- In all programming languages, we need some method for dividing a large piece of code into functional units, called **modules**. Assume each module is stored in its own file. However, these modules still share considerable amount of common information, and there are dependencies.

- The "make" program allows one to automatically maintain dependency information, and if a dependent file is modified, the "dependee file" must also be recompiled. However, if the interface between them are not modified, we really do not need to recompile the dependee file. That is where the concept of "header file" comes in.

- A **header file** is normally used to specify these common information or "interfaces".

  In C language and its derived languages (C++, etc), the header files are normally given the ".h" extension. We call these **dot-h** files. However, Java has a different solution to this problem and it does not use header files.

- Example. Consider our STM software. You will be writing various projects modifying its OS. Currently, you need to Suppose we split stm.c into stm.h and stm.c. We can compile stm.c into a stm.o file. Any OS that is written for STM machine can just link to stm.o. As long as the "hardware" does not change, we do not have to recompile stm.c.

- Example. All the C libaries are organized in this way. For instance, to use the functions in the "standard I/O" library, you just have to include "stdio.h" in your program. You can then link to these precompiled routines such as printf and scanf.

- Now, in a long sequence of includes, you could actually include the .h files many times. This could lead to the error of multiple definition. To avoid this, we insert an **include guard** into each .h file. For the file stm.h, we might write:

```
#ifndef H_STM
#define H_STM

... stm header information

#endif
```

Here, `H_STM` is a unique variable associated with the file. Note the convention of using CAPS and also *H*-underscore.

- THE DOT-H SOLUTION: For each file (like stm.c), we create a file called "stm.h". The file stm.c must include stm.h, and so must any other module that depends on stm.c. The include statement is this:

```
#include <stdio>            // standard library includes first
#include "stm.h"            // user-defined includes follow
                            // NOTE: the <...> versus "..." construct
                            //   which distinguishes between standard
                            //   and user-defined libraries.

... stm.c implementation here ...
```

  The advantage of this is that we have separated the INTERFACE and IMPLEMENTATION into dot-h and dot-c files (resp).

- WHAT GOES INTO A DOT-H FILE?

  function headers (stubs), global and static declarations, type definitions, defined constants.