Lecture X
Virtual Memory

§1. Introduction

- So far, we have introduced "standard paging". In standard paging, each process is either in main memory or it is not. We may refine it slightly, by introducing swapping. Swapping processes out of main memory puts them in an non-running state. But conceptually, all running processes must be in MM.

  In demand paging, we assume that each process to reside in disk or secondary memory called **backing store**. We only bring in the necessary pages for it to execute. Since pages are brought in on demand, we call it **demand paging**.

- We prefer not to use the term "swap space" for backing store because we like to reserve the notion of "swapping" as a refinement of of standard paging.

- Swapper versus pager.

  In paging, we have the concept of swapping. But now, we have the concept of pager, where only certain pages are "paged in", as opposed to entire process being "swapped in".

- Basically it is an issue of caching! Just as there is a cache between CPU and MM in order to improve standard paging, we now need a cache between MM and Disk in order to speedup demand paging.

  Thus demand paging and standard paging are both caching issues at different points of the memory hierarchy.

- 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 legal
  – 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**

  –

  $$\boxed{\text{FORMULA: } EAT = (1-p) \times MAT + p \times PFT}$$

  – e.g., Suppose we page fault once every 1000 pages, memory access time is $MAT = 200ns$, and page fault time is $PFT = 10ms$. Thus $p = 0.001$ and effective access time is

  –

  $$\begin{aligned} EAT &= 0.999MAT + 0.001PFT \\ &= 200 \times 10^{-9} + 0.001 \times 10^{-5} \\ &= 200 \times 10^{-9} + 10^{-8} \\ &= (200 + 10) \times 10^{-9}. \end{aligned}$$

  Thus $EAT = 10.2$ microseconds.

  – 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.

§2. 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 (see Silberschatz):

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

- 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! (Such algorithms are said to be clairvoyant)

- Let us prove that OPT is optimal in the following very strong sense. For any page replacement algorithm ALG, any integer $n > 0$ and any reference string S, let $ALG_n(S)$ denote the number of page faults when ALG is run on S using an initially empty cache with n frames. Then we claim that $OPT_n(S) \le ALG_n(S)$.

  PROOF: Suppose $S = (a_1, a_2, \ldots, a_m)$ where a_i is the ith page reference. The **trace** of $ALG_n(S)$ is a sequence

  $$\tau = (r_1, r_2, \ldots, r_m)$$

  where r_i is the action of ALG_n in response to the ith page reference. Now, r_i can either be $*$ (no action) or $b \to c$ (replace page b by page c). In case of $b \to c$, naturally b must be previously loaded into the cache, and we may assume that $b \neq c$. We allow the special case where $b = *$ meaning that c is loaded into an empty frame. Let $\#\tau$ denote the number of page replacement actions in τ.

  Consider the action in response to the ith page reference a_i. No action is only allowed if there is a page hit. But one can imagine ALG perform a page replacement b/c even in case of a page hit. But in this case $b \neq a_i$ and of course c should be $*$ or a In case of a page miss, we must have a page replacement b/c in which $c = a_i$.

  Let $\tau^* = (r_1^*, \ldots, r_m^*)$ be the trace of OPT_n on S. For any sequence of actions, $\tau = (r_1, \ldots, r_m)$, we say τ is **valid** for S if it satisfies the above rules. It suffices to show that for any valid trace τ, the following inequality holds:

  $$\#\tau^* \le \#\tau$$

  For simplicity, we first assume that τ takes no action in case of a page hit. Suppose

  $$\#\tau^* > \#\tau. \qquad (1)$$

for some S and n. We will derive a contradiction as follows: among the valid traces τ that satisfies (1), choose one for which the common prefix of τ and τ^* is the longest. We will construct a new valid trace τ' such that $\#\tau' \leq \#\tau$ and such that the common prefix of τ' and τ^* is greater. This would be a contradiction.

CONTD:

- 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.

  WHY IS THIS CONSIDERED AN APPROXIMATION TO OPT? Because if recent usage pattern is a predictor of the future, then recently used pages will probably be used in the near future. Conversely, very old references may suggest unlikely usage in the future...

  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 its 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.

§3. LRU Implementations

- We consider enhancements of the Reference Bit and the Second Chance Algorithm.
- Aging Of Bits. Another idea is to have automatic aging of reference bits. Every 100 ms or so, a timer interrupt will cause the OS to refresh (reset) the reference bits of each page.
- Another idea is to have a **reference byte** (i.e., 8 reference bits). Of course more bits is conceivable.

  When combined with the aging idea, we simply shift the reference byte to the right (hence losing the lowest order bit).

E.g., a reference byte of 0000,0000 shows the page has not been referenced in the last 8 periods. Thus the reference byte is a historical sample of page references.

So we can use the numerical value of this byte as the time stamp (the smallest numerical value is the one to be replaced).

- We can store a PAIR of bits, the usual reference bit and a new **modify bit**. The pair (r, m) has four possible values:

  $(0, 0)$: not recently used or modified. Best for replacement.

  $(0, 1)$: not recently used but modified. Need to write out the page if we replace it.

  $(1, 0)$: recently used but not modified.

  $(1, 1)$: recently used and modified. Least likely candidate for replacement.

  We view (r, m) as an integer between 0 and 3, representing the "class" of the page.

  We search for a page in the lowest class for replacement.

§4. Frame Allocation Among Processes

- So far, in demand paging, we considered each process in isolation.
- In a multiprocessing environment, assuming that all the frames are allocated to the running process, it means that we must swap out ALL the pages in a frame when we do a context switch. This is very expensive.
- But first, let us review the basic SCHEDULING MODEL (§4.2., p.109) that we assume. We assume that processes are partitioned into two pools:

  – **active processes** which are partially loaded in main memory. These processes may be further subdivided into queues (Fig.4.5, p.109): a **ready queue**, and feeding into this ready queue are several queues (I/O queue, fork queue, etc). The fork queue might be waiting for a child to terminate.

  – **swapped processes** which are spooled on the disk.

  There are two schedulers:

  – **long-term scheduler** or job scheduler: chooses a process from the swapped process to make it active. This scheduler controls the **degree of multiprogramming**.

  – **short-term scheduler** or CPU scheduler: chooses a process from active processes to make it the current running process.

  The short-term scheduler runs relatively frequently (say, every 100 ms). as compared to the long-term scheduler (say, every 10 second).

  The longterm scheduler tries to get a good mix of I/O-bound and CPU-bound processes in the active list. In some systems, the longterm scheduler is missing or minimal.

- We may want to allocate a certain number of frames per process.

  A simple choice is equal allocation of frames per process.

  Another is proportional allocation – larger processes is allocated more frames.

- This leads to the distinction between LOCAL versus GLOBAL page replacement:

  Local is when we choose replacement page from the current allocated frames.

  Global is when we choose a replacement page from the entire pool of frames.

- **Thrashing** is the phenomenon of a process spending more time doing paging than executing its code.

  Thrashing can occur if we do global replacement without regard to actual performance (p.394 and also Fig 10.16.)

  By using only local replacement, we can limit thrashing to each process, but it also affects the performance of other processes.

- To avoid thrashing, we want to have a way to estimate the frames needed by each process.

  The fundamental idea to be exploited is the **locality assumption**. It says that each process executes in a locality (which is a small set of pages that are used together). This locality moves over the course of the process's execution.

  If we can estimate this locality, we can try to provide the necessary pages for the process. Then the process will not page fault until it goes to the next locality.

§5. Working Set Model

- The **Working Set Model** is a method of estimating locality.

  We use a parameter Δ to define a **working set window**. The most recent Δ page references defines the set of pages called the **working set**.

- Suppose $S = (r_1, r_2, \ldots)$ is the reference string. Let S_t denote the prefix of S of length t ($t = 1, 2, 3, \ldots$). E.g., $S_1 = (r_1)$, $S_2 = (r_1, r_2)$, etc. Let $WS(t)$ denote the last Δ page references in S_t.

- E.g., let $S = (2615, 7777, 5162, 3412, 3444, \ldots)$.

  If $\Delta = 5$ then $WS(8) = \{5, 7\}$ and $WS(12) = \{1, 2, 5, 6, 7\}$.

- Once the Δ is fixed, let WWS_i denote the **working set size** of the ith process. Then the demand is $D = \sum_i WWS_i$.

  If D exceeds the number of pages, we might get thrashing, and the degree of multiprogrogramming ought to be reduced.

  We could also allow the Δ be a function of the process: Δ_i for the ith process.

§6. OS Examples of Demand Paging

CONSIDER demand paging in Windows XP.

- When a process is created, it is given a minimum and maximum for its working set.

  Typically, minimum is 50, maximum is 345.

- The Virtual Memory manager maintains a list of free frames.

  When a page fault occurs for a given process, and its maximum allocation is not reached, and there is a free frame, we allocate a new frame.

  Otherwise we use a local replacement algorithm.

- If the free memory falls below a threshold, it uses a **automatic working-set trimming** to restore this threshold: for each process, see if we can deallocate some pages so that it reaches its worksing-set minimum.

Exercises

Exercise 6.1: What is the basic difference in assumptions between (standard) paging and demand paging? ♢

Exercise 6.2: Let S be a reference string, and S^r denote its reverse.
(i) Show that OPT gives the same number of page faults on S as on S^r.
(ii) Show that LRU gives the same number of page faults on S as on S^r.

♢

Exercise 6.3: What is the difference between Policy, Mechanism and Implementation? Illustrate all three concepts with one example from page replacement algorithms. ♢

End Exercises