VIRTUAL MEMORY

WHY VIRTUAL MEMORY?

- We've previously required the entire logical space of the process to be in memory before the process could run. We will now look at alternatives to this.

- Most code/data isn't needed at any instant, or even within a finite time - we can bring it in only as needed.

VIRTUES

- Gives a higher level of multiprogramming

- The program size isn't constrained (thus the term 'virtual memory'). Virtual memory allows very large logical address spaces.

- Swap sizes smaller.

DEFINITIONS

Virtual memory The separation of user logical memory from physical memory. Thus can have large virtual memory on a small physical memory **<<< SEE FIGURE 9.1 >>>**

Demand paging When a page is touched, bring it from secondary to main memory.

Overlays Laying of code data on the same logical addresses - this is the reuse of logical memory. Useful when the program is in phases or when logical address space is small.

Dynamic loading A routine is loaded only when it's called.

DEMAND PAGING:

- When a page is referenced, either as code execution or data access, and that page isn’t in memory, then get the page from disk and re-execute the statement.

- **<<< FIGURE 9.2 >>>** Shows migration between memory and disk.

- One instruction may require several pages. For example, a block move of data.

- May page fault part way through an operation - may have to undo what was done. Example: an instruction crosses a page boundary.

- Time to service page faults demands that they happen only infrequently.

- **<<< FIGURE 9.3 >>>** Note here that the page table requires a "resident" bit showing that page is/isn't in memory. (Book uses "valid" bit to indicate residency. An "invalid" page is that way because a legal page isn't resident or because the address is illegal.

STEPS IN HANDLING A PAGE FAULT

- The process has touched a page not currently in memory.

- Check an internal table for the target process to determine if the reference was valid (do this in hardware.)

- If it was valid, but page isn't resident, then try to get it from secondary storage.

- Find a free frame; a page of physical memory not currently in use. (May need to free up a page.)

- Schedule a disk operation to read the desired page into the newly allocated frame.

- When memory is filled, modify the page table to show the page is now resident.

- Restart the instruction that failed

- Do these steps using **<<< FIGURE 9.4 >>>**

REQUIREMENTS (HARDWARE AND SOFTWARE) INCLUDE:

- Page table mechanism
- Secondary storage (disk or network mechanism.)
- Software support for fault handlers and page tables.
- Architectural rules concerning restarting of instructions. (For instance, block moves across faulted pages.)

PERFORMANCE OF DEMAND PAGING

- We are interested in the effective access time: a combination of "normal" and "paged" accesses.
- It's important to keep fraction of faults to a minimum. If fault ratio is "p", then

$$\text{effective\_access\_time} = (1 - p) * \text{memory\_access\_time} + p * \text{page\_fault\_time}.$$

- Calculate the time to do a fault as shown on **<<< PAGE 310 >>>**

  fault time = 10 milliseconds (why)

  normal access = 100 nanoseconds (why)

- How do these fit in the formula?

PAGE REPLACEMENT:

- When we overallocate memory, we need to push out something already in memory. Overallocation may occur when programs need to fault in more pages than there are physical frames to handle.

- Approach: If no physical frame is free, find one not currently being touched and free it. Steps to follow are:
  <<< FIGURE 9.6 >>>

  1. Find requested page on disk.
  2. Find a free frame.
     a. If there's a free frame, use it
     b. Otherwise, select a victim page.
     c. Write the victim page to disk.
  3. Read the new page into freed frame. Change page and frame tables.
  4. Restart user process.

- Hardware requirements include "dirty" or modified bit.

PAGE REPLACEMENT ALGORITHMS:

- When memory is overallocated, we can either swap out some process, or overwrite some pages. Which pages should we replace?? <--- here the goal is to minimize the number of faults.

- Here is an example reference string we will use to evaluate fault mechanisms:

  7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

FIFO

- Conceptually easy to implement; either use a time-stamp on pages, or organize on a queue. (The queue is by far the easier of the two methods.)

- Look at pattern in **<<< FIGURE 9.8 >>>**

OPTIMAL REPLACEMENT

- This is the replacement policy that results in the lowest page fault rate.

- Algorithm: Replace that page which will not be next used for the longest period of time.

- Impossible to achieve in practice; requires crystal ball. **<<< FIGURE 9.10 >>>**

LEAST RECENTLY USED (LRU)

- Replace that page which has not been used for the longest period of time. **<<<FIGURE 9.11 >>>**

- Results of this method considered favorable. The difficulty comes in making it work.

- Implementation possibilities:

  Time stamp on pages - records when the page is last touched.

  Page stack - pull out touched page and put on top

- Both methods need hardware assist since the update must be done on every instruction. So in practice this is rarely done.

LRU APPROXIMATION

- Uses a reference bit set by hardware when the page is touched. Then when a fault occurs, pick a page that hasn't been referenced.

- Additional reference bits can be used to give some time granularity. Then pick the page with the oldest timestamp.

- Second chance replacement: pick a page based on FIFO. If its reference bit is set, give it another chance. Envision this as a clock hand going around a circular queue. The faster pages are replaced, the faster the hand goes.
- **<<< FIGURE 9.13 >>>**

- Maintain a modified bit, and preferentially replace unmodified pages.

ADD HOC (OR ADD-ON) ALGORITHMS

- These methods are frequently used over and above the standard methods given above.
-
- Maintain pools of free frames; write out loser at leisure

- Occasional writes of dirties - make clean and then we can use them quickly when needed.

- Write out and free a page but remember a page id in case the page is needed again - even though a page is in the free pool, it can be recaptured by a process (a soft page fault.)

ALLOCATION OF FRAMES:

- What happens when several processes contend for memory? What algorithm determines which process gets memory - is page management a global or local decision?

- A good rule is to ensure that a process has at least a minimum number of pages. This minimum ensures it can go about its business without constantly thrashing.

ALLOCATION ALGORITHMS

- Local replacement -- the process needing a new page can only steal from itself. (Doesn't take advantage of entire picture.)

- Global replacement - sees the whole picture, but a memory hog steals from everyone else

- Equal vs. proportional allocation - pros and cons.

- Can divide memory equally, or can give more to a needier process. Should high priority processes get more memory?

THRASHING:

- Suppose there are too few physical pages (less than the logical pages being actively used). This reduces CPU utilization, and may cause increase in multiprogramming needs defined by locality.

- A program will thrash if all pages of its locality aren't present in the working set.

- Two programs thrash if they fight each other too violently for memory.
  <<< FIGURE 9.14 >>>

- Locality of reference: Programs access memory near where they last accessed it.
  <<< FIGURE 9.15 >>>

WORKING SET MODEL

- The pages used by a process within a window of time are called its working set.
  <<< FIGURE 9.16 >>>

- Changes continuously - hard to maintain an accurate number. How can the system use this number to give optimum memory to the process?

PAGE FAULT FREQUENCY

- This is a good indicator of thrashing. If the process is faulting heavily, allocate it more frames. If faulting very little, take away some frames.

OTHER CONSIDERATIONS:

PREPAGING

- Bring lots of pages into memory at one time, either when the program is initializing, or when a fault occurs.

- Uses the principle that a program often uses the page right after the one previously accessed.

PAGE SIZE

- If too big, there’s considerable fragmentation and unused portions of the page.

- If too small, table maintenance is high and so is I/O.

- Has ramifications in code optimization.

PROGRAM STRUCTURE

Paging **should** be transparent to the user, BUT.   This routine gives **many** page faults:

```
long    A[MAX][MAX];

( for j = 0;   j < MAX;    j++ )
     ( for i = 0;   i  <  MAX;     i++ )
          A[ i ][ j ] = 0;
```

This routine gives **few** page faults:

```
long    A[MAX][MAX];

( for i = 0;   i  <  MAX;     i++ )
     ( for j = 0;   j < MAX;    j++ )
               A[ i ][ j ] = 0;
```