VIRTUAL MEMORY

Main memory management strategies have a common goal of keeping many processes in memory simultaneously to allow multiprogramming. However, they tend to require that an entire process be in memory before it can execute. This requirement that instructions must be loaded in physical memory to be executed seems both necessary and reasonable, but has the disadvantage of limiting the size of a program to the size of physical memory. Even in those cases where entire program is needed, it may not all be needed at the same time.

Virtual memory comes in to provide the ability to execute a program that is only partially in memory, thus conferring the following benefits:

- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large virtual address space, simplifying the programming task.
- Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap user programs into memory, so each user program would run faster.

Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory. Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory. This technique frees programmers from the concerns of memory-storage limitations. Virtual memory also allows processes to share files easily and to implement shared memory. In addition, it provides an efficient mechanism for process creation. Virtual memory is not easy to implement, however, and may substantially decrease performance if it is used carelessly.

Virtual memory is commonly implemented by demand paging.

Demand Paging

Consider how an executable program might be loaded from disk into memory. One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially need the entire program in memory. Loading the entire program into memory results in loading the executable code for all options, regardless of whether or not an option is ultimately needed by the user. An alternative strategy is to load pages only as they are needed. This technique is known as **demand paging** and is commonly used in virtual memory systems. With demand-paged virtual memory, pages are loaded only when they are demanded during program execution. Pages that are never accessed are thus never loaded into physical memory.

Usually, when a process is to be swapped in, the pager guesses which pages will be used before the process is swapped in instead of swapping in a whole process, thus, the pager brings only those necessary pages into memory. It therefore, avoids reading into memory pages that will not be used in anyway, decreasing the swap time and the amount of physical memory needed.

Hardware support is required to distinguish between those pages that are in memory and those pages that are on disk using the valid-invalid bit scheme. Where valid and invalid pages can be checked, checking the bit and marking a page will have no effect if the process never attempts access pages. Again, execution proceeds normally if pages to be accesses are inside memory. However, if a process tries to access a page that is not brought into memory, i.e. a page marked invalid causes a **page fault.** The paging hardware, in translating the address through the page table, notices that invalid bit is set, causing a trap to the operating system. This trap is the result of the OS failure to bring the desired page into memory. This trap is handled as follows;

- We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
- If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
- We find a free frame (by taking one from the free-frame list, for example).
- We schedule a disk operation to read the desired page into the newly allocated frame.
- When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
- We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

Advantages of Demand Paging

- Large virtual memory.
- More efficient use of memory.
- Unconstrained multiprogramming. There is no limit on degree of multiprogramming.

Disadvantages

- Number of tables and amount of processor over head for handling page interrupts are greater than in the case of the simple paged management techniques.

Page Replacement

When a page fault occurs, the OS determines where the desired page is residing on the disk and page it in if there are free frames. However, if there are no free frames, the OS instead of terminating the process, it finds out a frame that is not currently being used and frees it. The frame is freed by writing its contents to swap space and changing the page table to indicate that the page is no longer in memory. Now the freed frame is used to hold the page which the process faulted and page-fault procedure is carried out to perform necessary modifications resulting in page replacement.

When page replacement is required, the OS must select the frames that are to be replaced. Therefore, appropriate algorithms must be designed to solve this problem.

Page replacement algorithms

1. FIFO Page Replacement

The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

2. Optimal Algorithm

An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal page-replacement algorithm exists, and has been called OPT or MIN. It is simply replace the page that will not be used for the longest period of time.

3. LRU Page Replacement Algorithm

The key distinction between the FIFO and OPT algorithms is that the FIFO algorithm uses the time when a page was brought into memory, whereas the OPT algorithm uses the time when a page is to be ***used.*** If we use the recent past as an approximation of the near future, then we can replace the page that ***has not been used*** for the longest period of time. This approach is the **least recently used (LRU) algorithm**. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.

4. LRU-Approximation Page Replacement

Few computer systems provide sufficient hardware support for true LRU page replacement. In fact, some systems provide no hardware support, and other page-replacement algorithms (such as a FIFO algorithm) must be used. Many systems provide some help, however, in the form of a **reference bit**. The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table. Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware. After some time, we can determine which pages have been used and which have not been used by examining the reference bits, although we do not know the ***order*** of use. This information is the basis for many page-replacement algorithms that approximate LRU replacement.

Thrashing

If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately. This high paging activity is called **thrashing**. A process is thrashing if it is spending more time paging than executing.

External and Internal Fragmentations

As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when enough of the memory space exists to satisfy a request, but it is not contiguous; storage is fragmented into a large number of small holes. Internal fragmentation on the other hand occurs when there is free memory that is internal to partition, but is not being used.