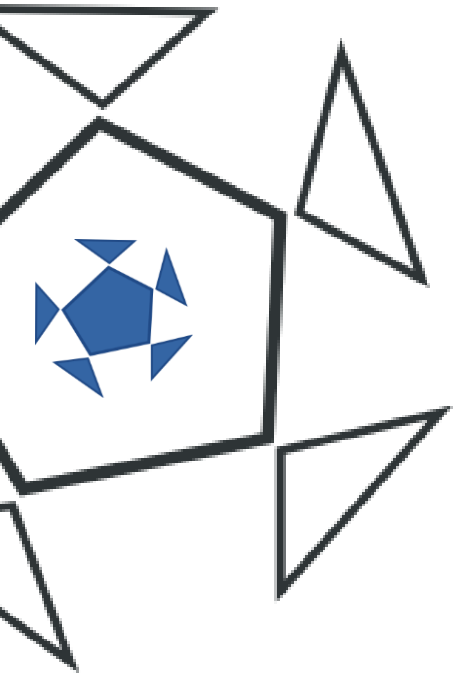
Raza's Simplified

Chapter 5



VIRTUAL MEMORY

In this Chapter

- Introduction
- Demand Paging
- Demand Segmentation

Introduction

Give me an answer! Let say the system has 2^{12} bytes of memory. Can you calculate the total size of processes that can run on this available memory?

You are right! 2^{12} B. Answer one more thing: can we run more processes, without increasing the memory? The answer is '*Yes'*.

Now you will ask: How? Well, we can do it using virtual memory. If you have 2^{12} bytes of physical memory available, then still, you can run 2^{16} bytes or more of the processes. So we can say, virtual memory removes the limits of physical memory.

The area on disk acting as an extension of RAM, storing swapped out pages is called virtual memory.

Demand Paging

Virtual memory is applied using both demand paging and demand segmentation.

Processes run from RAM and the job scheduler creates new processes. The processes are divided into pages of equal size. Then physical memory is divided into page frames of equal size. The page and page frame sizes are also same and are in power of 2.

New processes keep on coming. When there is not enough memory available in the system, the pages are swapped to disk. There they are stored on contiguous locations. When the process needs to execute a swapped out page, then a reference to the page is created. Now the page is copied back to the memory in a free page frame. Then the page is used as in simple paging.

The page table now contains at least two types of information in addition to other information:

1. Page frame
2. In/Out bit

When a virtual address is generated, the OS reads the page number and finds the corresponding page table entry. From this entry, the page in/out bit is checked. *In/Out bit is used to find if the page is in memory or it has been swapped out on to the disk.*

If the page is in memory then a physical address will be generated by appending the page frame number to the offset.

But if the page is not in memory than physical address cannot be generated. Rather, a page fault will occur.

Page Fault:

When a referenced page is not in memory, the situation is called page fault. It is a trap. The in/out bit gives information if the page is in memory or not. The process cannot continue from this point until the page is brought back in memory. *To handle this page fault, a page fault handling routine is executed.*

Working of Page Fault Routine

First an empty/free page frame is searched. If it is found, then the page referenced by the process is copied to that page frame.

If an empty frame is not available, then the OS selects a page frame for swapping it out to get a free page frame. This selection is done using page replacement algorithms. The selected page is copied to the hard disk and its in/out bit is set to out in the page table. Since a free page frame is available, so the page needed is copied into memory on the newly freed page frame.

Either way, the page is in memory so the in/out bit is rest to in. Also the page frame field is adjusted and contains information about the current location of the page. Then the instruction which generated the page fault is executed again, and this time, it does not generate a page fault. Rather it gets the task done.

Page faults decrease the system performance because they need their own processing by the CPU and involve use of disk which is quite a slow device.

Thrashing

When there is excessive page swapping, then the system performance eventually lowers. This situation is not favored and it is called thrashing.

Using the Dirty Bit

The page table may contain a dirty bit field. Its use is especially helpful when a previously replaced page is again selected for replacement. It informs the OS if the versions of the page in memory and disk are the same or different. If the versions are same (dirty bit=0), then we need only to reset the page table in/out bit to out and free

the page frame. There is no need to copy the page to the disk since it is already there in unchanged form.

But if the dirty bit is set to yes/1, then it means the page has been changed since its last replacement. The change occurs due to some write operations or due to some change in variables or arrays in the page. In any case, the page in memory is different from the page in disk. So we must actually write the page to the disk if it is to be replaced.

When a page is changed, the memory management hardware sets the dirty bit to 1.

A previously replaced page again selected for replacement and has been modified since (whose dirty bit is set) needs to be written on disk.

Locality of Reference

When references to the pages which are in memory are generated, then the system performance increases and it is the same as systems having no virtual memory.

But when swapped out pages are referenced, the performance goes down due to the excessively occurring page faults. If the memory references are predictable, then OS can load those pages before they are referenced. So the page faults can be minimized.

But if the references are random, then advantages of virtual memory are not materialized and system performance will degrade.

Fortunately, most references are predictable and using this prediction, system performance remains quite acceptable. The references appear to stay within limits and we call it *locality of reference*. This gives us a very good average access time for swapped out pages. Here we get a *reference string, which is the ordered list of all pages referenced by a process.*

Forms of Locality

1. Temporal locality
2. Spatial locality

Temporal Locality

It means a page referenced in near past will be used again. Like a variable used in an expression can occur twice or more in that expression. Or a loop can run a block of code many times.

Spatial Locality

It means that if page is referenced then its upcoming neighbor page would be referenced soon. For example, using an array in a loop, when an element in an array is being iterated, OS can predict that the next iteration will use next element of the array.

Experimental work highlights that references group in localities, for example a loop executes a specific group of instructions or pages repeatedly. If the functions and subroutines are called, then the pages in locality will increase. When the process comes out of a running locality, for example the loop ends, then the process begins to use another locality.

Page Locking

The OS must not allow some pages to be swapped out because they are critical in their nature and swapping them would destroy the smooth working of the system. This is a necessary requirement. For example:

1. The pages containing OS must never be swapped out.
2. Suppose that the pages containing the swapper are swapped out. Since they are now on disk, so they cannot swap themselves in back to memory because only the code in memory can be run by the CPU. Thus, swapper must never be swapped out.
3. This is also needed in I/O system design. Suppose a process is taking input and saving it directly to a page's memory location. Then that page cannot be swapped out. If it is swapped out, then the incoming data will find no place to reside.

The page locking is used to achieve this requirement. The pages are locked in memory so if an algorithm selects them for removal, the operation is denied.

To implement this, a page table field called ***lock bit*** is provided.

If it is set then it means that the page is locked and a page replacement operation must be denied.

Alternatively, OS can be designed without page table having the lock bit field and still not allowing some critical pages to be swapped out.

1. The OS can be designed so that no part of OS would ever be replaced.

2. The I/O system can be designed so the input data being received in a memory location of a page can be received and buffered in the device's I/O buffer, from there it can be transferred to the process later.

Such a system will always be able to swap user processes when needed. And OS will never be swapped out.

Page Size

Many factors are considered for selecting a page size:

1. The page size is always a power of 2. It is usually from 512 bytes to 16 kilo bytes. (512 bytes to 16,384 bytes)
2. To reduce internal fragmentation, the page size should be small.
3. To reduce page table size, the page size should be large. It also decreases the pages needed by a process. Hence the table fragmentation is low, as low memory is needed to store a smaller page table. When the page table is loaded into registers, the time needed is also reduced as small page tables load faster.
4. The overall overhead is reduced when pages are of large size. When a page is to be loaded from disk, processing of the page fault needs its own processing time. Then the disk has its own three types of intervals: seek time, rotational latency and transfer latency. In most cases, the transfer time is negligible. Seek time and rotational latency together make 90% of the time needed. To read a 2K contiguous data block takes time equal to reading one 1K block. So it is better to keep the page size larger as more data can be transferred in a single operation of disk.
5. Small pages offer better resolution. They can manage a locality of reference of a process in a better way. Using a small page reduces the amount of un-needed information which would be transferred during page swapping. Also, small sized page consume less memory. So less unused data stays in memory. The secured memory can be used for other purposes.

Page Replacement Algorithms

Page replacement algorithms select a page to be replaced. The swapper uses page replacement algorithms to make room in RAM. This is done when RAM runs short.

Optimal Page Replacement Algorithm

An algorithm which avoids page fault for the maximum possible executed instructions is called optimal page replacement algorithm. It gives ideal performance. It is only a conceptual thing. It needs prediction about future use of a swapped out page. So it remains only a benchmark and newly developed algorithms are judged against it.

FIFO

The page that has spent longest time in memory is selected to be swapped out.

It is straight forward. The page table must have a field to record load time or swap in time of a page. When a page is loaded in memory this filed is updated with the current time. When swapper executes, it compares the swap in time of every page and the oldest page is replaced.

Merits

i. Easy to implement

ii. Costs less

Demerits

FIFO can replace even a highly used page. It does not pay any attention to the usage of page. It just compares the arrival time in memory and swaps the page out. This is not efficient as a heavily used page will be referenced soon, resulting in page faults. Therefore, it is an inefficient algorithm.

Least Recently Used (LRU)

This algorithm keeps track of time when a page was last referenced. Also, the frequency of page reference is stored in a counter. The page table stores a counter for every page. When a page is referenced, the value in counter is incremented. When a replacement is to be made, the algorithm will select the page which has been least referenced. This is done by comparing all the counter values of the pages and selecting the lowest one. The page with lowest counter value is replaced.

We can use a square matrix to implement LRU. If the number of page frames is n then a matrix of n×n order is developed. Initially it contains 0s all over. The first row and first column of the matrix belong to page 1. Second row and second column belong to page 2 and so on. When a reference to a page m arises, first the complete m^{th} row is set to 1, then complete m^{th} column is set to 0. Now, if the swapper selects a page for replacement, it will be the page whose row contains minimum binary value in the matrix.

Not recently Used (NRU)

It is a variation of LRU. When this algorithm is implemented, the page table will contain a field for dirty bit and one for reference bit. The reference bit is set to 0 initially. When the pages are referenced, reference bit is set to 1 by the memory management hardware. An interval is set and upon expiry of this interval the OS sets the reference bit to 0. So, simply put, if the reference bit is 0, then the page has not been recently used. If the reference bit is 1, then it was recently used.

Now we have groups of pages on the basis of reference bit and dirty bit. First, the OS will search for a page whose reference bit is 0, then it will search for the pages whose dirty bit is 0. Swapping out the pages which are dirty takes more time and overhead. So the OS will try to find a page with both the reference and dirty bits having 0s. When multiple pages are available with 0s in both bits, then the OS can select anyone of them.

Aging

It is a variation of LRU. When this algorithm is enforced, a reference byte is added as a filed in page table. This byte can have 6 or 8 bits. When a page is referenced, the left most bit in the byte (called most significant bit or MSB) is set to 1 by the hardware. An interval is set and after that interval all bits are moved one step to the right. As a result the old right most bit (called least significant bit or LSB) is updated with 7^{th}, the 7^{th} is updated with 6^{th}, and so on. Now the MSB is given 0. This interval is kept smaller than the interval in NRU. An OS routine performs the rotation in the bits as a result of timer interrupts. When swapper executes the page with lowest binary value in the reference byte is selected.

If many pages have same lowest value in the reference byte, then OS can select anyone of them.

Second Chance

This algorithm has properties of both FIFO and NRU. It will select the oldest page from memory and then it will check its reference bit. If the reference bit is 0, then this page will be replaced. But if it is 1, then the currently selected page will be ignored. But its reference bit will be set to 0 and its arrival time will be set to current time. Now the algorithm will make another selection but this time the previously ignored page will not be considered. Now the previously second oldest page will be the oldest. And it will be checked for reference bit. This process will continue until a page is selected for removal.

Algorithm Performance

To measure algorithm performance, we need different sets of memory references. We call it reference string. The criterion is simple:

We need an algorithm, the reference string, the available free page frames and the number of page faults thus occurred. An algorithm with small page faults will be considered better than the algorithm with high page faults, if reference string and page frames are kept constant.

We can also judge the performance of an algorithm using different reference string and different page frame numbers. We can expect that if we increase the number of available free page frames, the page faults will reduce or at least they will be the same as previous. This expectation is quite logical but sometimes a special case comes in view where this scheme is not followed. It is called ***Belady's anomaly.***

A situation when increasing the page frames increases the page faults is called Belady's Anomaly.

Allocation Policies

Absolute Minimum Number of Page Frames

This is the minimum number of page frames which must be allocated to a process to run. No process can have lesser page frames. This number is set by the system hardware.

Suppose each process is allocated a page frame. If a page fault occurs, then the requested page will be brought to memory and that instruction will be rerun. But if the instruction needs two different pages, then an infinite cycle of page

faults will result. In many cases a single instruction may need multiple pages to load. This is especially true in systems using indirect addressing.

Minimum Number of Page Frames

To enhance performance, the OS also sets a minimum number of page frames. This is always larger than the absolute minimum number of page frames. With the exception of Belady's Anomaly, the page faults decrease, if we increase the page frames.

The number of processes running at any time must be adjusted to achieve this performance goal. This limitation on the number of processes in memory is done by the job scheduler (long term scheduler). The job scheduler can adjust the number of processes so the page faults caused due to the shared page frames would reduce, and thrashing would not result.

OS decides the number of page frames to allocate to a process. OS can use following strategies:

1. **Equal allocation**: every process gets an equal number of page frames.
2. **Proportional Allocation**: every process gets page frames in proportion to its size. Large processes get more page frames than smaller ones.

In both cases, a process gets a fixed number of page frames

Replacement Strategies

Local Scope: The page is replaced from a page faulting process. The overall number of page frames allocated to the process does not change.

Variable Allocation, Global Scope: The page frame can be freed from any process running in memory. The swapper searches all the processes running in memory to find a suitable page frame for replacement when a page fault occurs. Since a page frame can be taken away from any process, therefore it will result in variable page allocation to the processes. The page frames allocated will change dynamically as page faults in the system occur.

If removing a page frame from a process reduces its allocated page frames lower than minimum number of page frames, then this page frame is ignored

and a new selection is made on another process, or the complete process is swapped out.

Variable Allocation, local scope: this strategy offers a compromise. The number of allocated page frames change as the process requirements change. But whenever a process generates a page fault, the replacement is made from current process. No other process is affected. In most cases, the working set of a process is used to find how many page frames should be allocated to a process.

Working Set

It is the set of pages in some particular point in time referenced in some preceding time interval.

Notation:

W (t, Δ)

W is the working set, t is the time when working set is measured and Δ is the interval.

It is important to note here that time is not the clock time. It is usually measured in terms of executing instructions. If we say here one unit time, then it actually means one executed instruction.

Selecting a suitable value for Δ is also tricky. It should be so moderate to reflect the true locality of a process. If Δ is too small, then all necessary pages will not be included. If Δ is too large, then unnecessary pages will be included, mostly from the previous unwanted locality.

The information about the pages in the working set serves its own purpose. For the OS, real important thing is the size of the working set. The size of the working set can be used to decide the number of page frames needed to be allocated to a process. The process gets page frames according to its needs derived from its working set. As a result the number of processes running parallel in memory is also adjusted.

Prepaging

Prepaging means to divide a full process into pages at the time it is swapped in to memory. The OS can do it only when it has the information about the process's working set.

Advantage is that the initial references to the working set will not generate page faults. So the OS will not have to manage the extra overhead. But, it happens that

some pages are not needed and they are also uselessly copied in memory, wasting useful space. It degrades system performance.

Prepaging is also used while handling a generated page fault. While a page is swapped in, a nearby page in virtual memory may also be loaded for the sake of spatial locality. It is especially beneficial while working with sequentially running programs. If a page is being accessed in such a program, the next page will be accessed after the current page. It will save time, as it is short and easier to load two consecutive pages together than loading individual pages. It is because of the fact that disk`s seek time and rotational time remain nearly equal when one page is accessed or two consecutive pages are accessed in a single request.

The only disadvantage is that many pages will be loaded but they may not be used and referenced. This method is not much useful and it is not found to be effective in practical systems due to wastage of memory.

Demand Segmentation

As we use paging to manage virtual memory, similarly, the segmentation can also be used, and we call it demand segmentation. But segments have variable sizes. So, it is far more complex and difficult to use segmentation to implement virtual memory. First, the size of a segment is considered when swapping is needed as a page fault occurs.

The working set is highly changed because of varying sizes of segments.

Demand Segmentation with Demand Paging

A better way is to use segmentation with paging to implement virtual memory. We call it demand segmentation with demand paging. The segments are considered swapped in if the page table is in memory. So, it is actually like working with simple segmentation with simple paging. And virtual memory is implemented by demand paging the segments.