Virtual Memory

1 Introduction

Virtual memory is a technique that allows the execution of processes that are not completely in memory. Programs can be larger than physical memory. It abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory.

The size of virtual storage is limited by the addressing scheme of the computer system and amount of secondary memory is available not by the actual number of the main storage locations. The ability to execute a program that is only partially in memory has many benefits.

- A program would no longer be constrained by the amount of physical memory that is available.
- Because each user program could take less physical memory, more programs could be run at the same time.
- Less I/O would be needed to load or swap user programs into memory, so each user program would run faster.

The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory - typically, in contiguous memory, though, physical memory may be handled in page frames by MMU.

Virtual address spaces that include holes are known as **sparse address spaces**. Using a sparse address space is beneficial because the holes can be filled only when the stack or heap segments grow or if we wish to dynamically link libraries.

Virtual memory allows files and memory to be shared by two or more processes through page sharing.

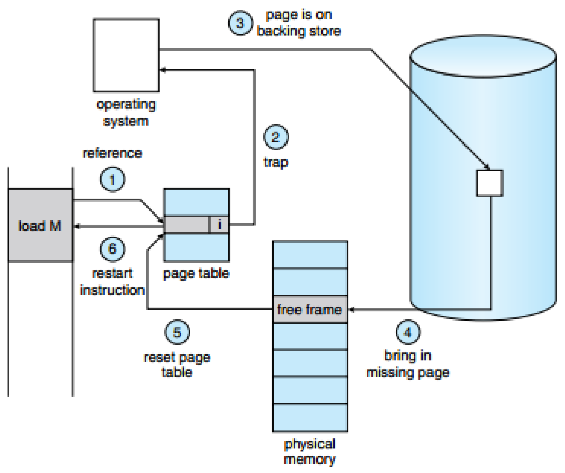
- System libraries can be shared by several processes through mapping of the shared object into a virtual address space.
- Virtual memory allows one process to create a region of memory that it can share with another process. Processes sharing this region consider it part of their virtual address space.
- Virtual memory can allow pages to be shared during process creation with the fork() system call, thus speeding up process creation.

2 Demand Paging

A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory. When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a **lazy swapper** that never swaps a page into memory unless that page will be needed.

A swapper manipulates entire processes, whereas a pager is concerned with the individual pages of a process. We thus use pager, rather than swapper, in connection with demand paging.

The **valid-invalid bit scheme** distinguishes a page in memory from disk. When a bit is set to "valid", it is both legal (part of the logical address space of process) and in memory. Access to a page marked invalid causes a **page fault**.



Pure demand paging is to never bring a page into memory until it is required. We start with no pages and fault every time we need a new page. Programs tend to have **locality of reference**. Therefore, we dont need to access several new pages with every execution practically.

The hardware to support demand paging is same as that for paging and swapping - **Page table** and **Secondary memory**. The ability to restart an instruction after page fault is important.

The major difficulty arises when one instruction may modify several different locations with an overlap between source and destination. It is not sufficient to restart in this case as the contents have been modified.

- In one solution, a microcode attempts to access both ends of both blocks. If a page fault is going to occur, it will happen at this step, before anything is modified.
- The other solution uses temporary registers to hold the values of overwritten locations. If there is a page fault, all the old values are written back into memory before the trap occurs.

To keep the slowdown due to paging reasonable, fewer than one memory access out of 399,990 to page-fault can be allowed. Disk I/O to swap space is generally faster than that to the file system because swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used The system can gain better paging throughput by:

- copying entire file image into swap space at startup and performing demand paging from the it.
- demanding pages from file system initially but writing it to swap space as they are replaced.

3 Copy-on-Write

Process creation using the *fork()* system call can bypass demand paging by using a technique similar to page sharing. *fork()* worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent, only to be replaced by the *exec()* call. **Copy-on-write** allows parent and child to share same pages until the child requests to update a particular page.

Many OS provide a **pool of free pages** for requests when copy-on-write pages need to be managed and the stack or heap for a process must expand, using a technique known as **zero-fill-on-demand**. The pages have been zeroed-out before being allocated, thus erasing the previous contents.

With *vfork()* (**virtual memory fork**), the parent process is suspended, and the child process uses the address space of the parent. It does not use copy-on-write and altered pages are visible to the parent. It is used when the child process calls exec() immediately. It is an extremely efficient method of process creation and is sometimes used to implement UNIX command-line shell interfaces.

4 Page Replacement

Allocation of frames and **Page replacement** is basic to demand paging. It completes the separation between logical memory and physical memory. With no demand paging, all the pages of a process still must be in physical memory but may not need to be in the main memory in entirety.

If no frames are free, 2 page transfers are required. The **dirty bit** is set if the page was modified in memory requiring it to be saved in swap space otherwise it is discarded.

We evaluate an algorithm by running it on a particular string of memory references, called **reference string** and computing the number of page faults.

FIFO Page Replacement

It associates with each page a monotonically increasing timestamp, typically the time when it is brought into memory. When a page must be replaced, the page with smallest timestamp is chosen.

Given a page replacement algorithm and a reference string, the page-fault rate may increase as the number of allocated frames increases. This unexpected result is known as **Belady's Anomaly**.

Optimal Page Replacement

Replace the page that will not be used for the longest period of time. It is difficult to implement as it requires future knowledge of the reference string.

If we let R be the reverse of a reference string S, then the page-fault rate for the OPT algorithm on S is the same as on SR.

For a **stack algorithm**, it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with n + 1 frames. They dont exhibit Belady's anomaly.

LRU Page Replacement

If we use the recent past as an approximation of the near future and replace the page that has not been used for the longest period of time. This is a stack algorithm. Two implementations are feasible:

- **Counters**. Associate a counter or logical clock with each pate-table entry. The clock is incremented for every memory reference.
- **Stack**. Store a stack of page numbers and whenever a page is referenced, it is removed from the stack and put on the top. It is best to implement it by using a doubly linked list with a head pointer and a tail pointer as we may need to remove middle element. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement.

LRU-Approximation Page Replacement

Counting-Based Page Replacement

We can keep a counter of the number of references that have been made to each page.

- **Least-frequently-used** requires page with smallest count to be replaced. A problem arises when a page is active initially but is not used again and is not replaced. On solution is to shift the counts right by 1 bit at regular intervals.
- **Most-frequently-used**is based on the argument that the page with smallest count has just been brought in and has yet to be used.

Page-Buffering Algorithms

- When a page-fault occurs, the victim frame is not written out immediately. It is written when the paging device is idle. This increases the probability that a page will be clean when it is selected for replacement.
- The used frame sent back to the pool of free frames is not emptied and is called back by a pointer when that frame is required.

5 Allocation of Frames

Under pure demand paging, we only allocate frames using a page-fault when it is requested by the process. We can allocate some of the frames from the free-frame list, reducing the overhead of restarting the instruction on a page-fault.

There must enough frames to hold all the pages requested by a single instruction.In an architecture that allows multiple levels of indirection, an instruction can reference an indirect address which could do the same, eventually requiring every page in the virtual memory be touched and present in physical memory. To overcome this, a limit of the number of levels of indirection is placed.

The minimum number of frames is defined by the computer architecture, whereas the maximum number is defined by the amount of physical memory available.

Allocation Algorithms

The remaining frames in the free-frame buffer pool can be equally shared among all processes (**equal allocation**) or shared based on the total amount of memory required (**proportional allocation**).

With the increase in multi-programming level, the number of frames per process reduces. In order to give a high-priority process more frames to enhance execution speed, the proportional split can be done based on the priority of the process.

Global Vs Local Allocation

Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process. **Local replacement** requires that each process select from only its own set of allocated frames.

With a global replacement policy, frames from a low-priority process can be allocated to a high-priority process. One problem with global policy is that a process cannot control its own page-fault rate and it depends on the paging behavior of another process. The same process can take different execution time due to external circumstances.

Local replacement might hinder a process by not making available to it other, less used pages of memory. Thus, global policy results in greater system throughput and is therefore the more common method.

Non-Uniform Memory Access

Systems in which memory access times vary significantly are known collectively as **non-uniform memory access** (**NUMA**) systems. They are slower than systems in which memory and CPUs are located on the same motherboard.

The goal is to have memory frames allocated "as close as possible" to the CPU on which the process is running. The scheduler tracks the last CPU on which each process ran to improve cache hits. The memory-management system tries to allocate frames for the process close to the CPU on which it is being scheduled to decrease memory access times.

It is more complicated once threads are added. A process with many running threads may end up with those threads scheduled on many different system boards. Solaris allocates memory by creating an **lgroup** entity in the kernel which gathers together close CPUs and memory. There is a hierarchy based on the amount of latency between the groups. Solaris tries to schedule and allocate memory for all threads within an lgroup otherwise uses the next lgroup.

6 Thrashing

If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, it must be suspended as it can not reach the threshold now and the allocated frames are freed. This provision introduces a **swap-in, swap-out** level of intermediate CPU scheduling.

If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault and replace a page that will be needed right-away resulting in high paging activity called **thrashing**.

Cause of Thrashing

If CPU utilization is too low, the OS increases the degree of multi-programming by introducing a new process to the system. A global page-replacement algorithm replaces pages without regard to the process to which they belong. If a process enters a new phase in its execution and needs more frames, it starts faulting and taking frames away from other processes.

These processes need those pages, however, and so they also fault, taking frames from other processes. These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties and CPU utilization decreases.

The scheduler increases the degree of multi-programming as a result. The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multi-programming even more.

Solution Approaches

We can limit the effects of thrashing by using a local replacement algorithm (or priority replacement algorithm). With local replacement, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well.

The average service time for a page fault will increase because of the longer average queue for the paging device. Thus, the effective access time will increase even for a process that is not thrashing.

Working-set model

A **locality** is a set of pages that are actively used together. We allocate enough frames to a process to accommodate its current locality.

The set of pages in the most recent d page references is the **working set**. If a page is in active use, it will be in the working set. The basic principle states that if we allocate enough frames to a process to accommodate its current locality, it will only fault whenever it moves to some new locality.

The OS monitors the working set of each process and allocates to that working set enough frames to provide it with its working-set size. A process can be initiated or suspended based on the difference in the number of available free frames and the sum of working-set sizes.

Page-Fault Frequency

When the page fault rate is too high, we provide more frames to the process. If it is too low, we take away frames from the process. If the page-fault rate increases and no frames are available, a process may be suspended.