**Virtual Memory**

Virtual memory is a technique that allows the execution of process that may not be completely in memory. The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory is the separation of user logical memory from physical memory this separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.

**Following are the situations, when entire program is not required to load fully.**

1. User written error handling routines are used only when an error occurs in the data or computation.

2. Certain options and features of a program may be used rarely.

3. Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.

**The ability to execute a program that is only partially in memory would counter many benefits.**

1. Less number of I/O would be needed to load or swap each user program into memory.

2. A program would no longer be constrained by the amount of physical memory that is available.

3. Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

**Paging**

* Paging is a memory management technique that allows non-contiguous memory allocations to processes and avoids the problem of external fragmentation. Most modern operating systems use paging. With paging, the physical memory is divided into frames, and the virtual address space of a process is divided into logical pages. Memory is allocated at the granularity of pages. When a process requires a new page, the OS finds a free frame to map this page into and inserts a mapping between the page number and frame number in the page table of the process.
* Suppose the size of the logical address space is 2 m bytes, and the size of the physical memory is 2 n bytes. That is, logical addresses are m bits long, and physical memory addresses are n bits long. Suppose the size of each logical page and physical frame is 2 k bytes. Then, out of the m bits in the virtual address, the most significant m − k bits indicate the page number, and the least significant k bits indicate the offset within the page. A page table of a process contains 2 m−k page table entries (PTEs), one for each logical page of the process. Each PTE stores, among other things, the physical frame number which this page corresponds to. Now, since there are a total of 2 n−k physical frames, it would take n − k bits to identify a physical frame. Thus, each of the PTEs must be at least n−k bits in size, putting the total size of the page table of a process at at least 2 m−k × (n − k) bits. Note that PTEs contain a bit more information than just the physical frame number, e.g., permissions to access the page, so PTEs are slightly longer than n − k bits.
* For example, consider a system with 32-bit virtual addresses (i.e., the virtual address space is of size 2 32 bytes = 4GB). Suppose the size of the physical memory is 64 GB, and pages/frames are of size 4KB each. The total number of pages, and hence the number of PTEs in a page table, is 2 20 for each process. The number of physical frames in the system is 2 24. Therefore, each PTE must be at least 24 bits, or 3 bytes long. Assume that a PTE is 4 bytes long. Therefore, the size of the page table of every process is 2 20 ∗ 4 bytes = 4MB.
* To translate a virtual address to a physical address, the MMU uses the most significant p = m − k bits of the virtual address to offset into the page table and replace the most significant p bits with the f = n − k bits of the physical frame number. Assuming pages and frames are the same size, the lower k bits that indicate the offset in the logical page will still map to the offset in the physical frame.
* Note that a process need not always use up its entire virtual address space, and some logical pages can remain unassigned. In such cases, a bit in the PTE can indicate whether the page is valid or not, or the size of the page table can itself be trimmed to hold only valid pages. Accesses to unmapped/invalid logical addresses will cause the MMU to raise a trap to the OS. The OS can sometimes leave certain addresses unmapped on purpose, e.g., at the end of the user stack, to detect processes writing beyond page boundaries. The valid bit in the PTE, and the read/write permissions bits are powerful tools for the OS to enforce memory access control at the granularity of a page.
* A subtle point to note is that when a process is executing on the CPU, memory access requests from the CPU directly go to the memory via the MMU, and the OS is not involved in inspecting 6 or intercepting every request. If the OS wishes to be informed of access to any memory page, the only way to enforce it would be to set appropriate bits in the PTEs which will cause a memory request to trap and reach the kernel. For example, consider how an OS would implement copyon-write during fork. If the OS has only one memory copy for the child and parent processes, how does it know when one of them modifies this common image, so that it can make a copy? The OS can set all PTEs in the common memory image to read only, so that when either the parent or the child tries to write to the memory, a trap is raised by the MMU. This trap halts the execution of the process, moves it to kernel mode, and transfers control to the kernel. The kernel can then make a copy the memory image, update the page table entries to point to the new image, remove the read only constraint, and restart the halted process.
* How are pages sized? In general, smaller page sizes lead to lesser internal fragmentation (i.e., space being unused in the last page of the process). However, smaller pages also imply greater overhead in storing page tables. Typical pages sizes in modern operating systems are 4-8KB, though systems also support large pages of a few MB in size. Most modern operating systems support multiple page sizes based on the application requirements.
* With smaller virtual address spaces, the page table mappings could fit into a small number of CPU registers. But current page tables cannot be accommodated in registers. Therefore, page tables are stored in memory and only a pointer to the starting (physical) address of a page table is stored in CPU registers (and changed at the time of a context switch). Thus, accessing a byte from memory first requires mapping that byte’s virtual address to a physical address, requiring one or more accesses of the page table in memory, before the actual memory access can even begin. This delay of multiple memory accesses imposes an unreasonable overhead. Therefore, most modern systems have a piece of hardware called a **translation look-aside buffer (TLB).**

**TLB cache**

A TLB is a small, fast cache that stores the most recently used mappings from virtual to physical addresses. TLB is an associative memory that maps keys (virtual addresses) to values (physical addresses). Most architectures implement the TLB in hardware, and the TLB is transparent to the OS. That is, the OS only manages the page tables and does not control the TLB, and caching in the TLB is done automatically by the hardware. However, some architectures do provide a software TLB with more complicated features.

• If a virtual address is found in the TLB (a TLB hit), a memory access can be directly made. On the other hand, if a TLB miss occurs, then one or more memory accesses must first be made to the page tables before the actual memory access can be made. A healthy TLB hit ratio is thus essential for good performance. Further, the TLB reach (or the amount of address space it covers) should also be large.

• Let Tc denote the time to check if an address is mapped in the TLB, and let Tm denote the time to access main memory. Further, suppose a TLB miss requires n additional memory accesses to page tables before the actual memory access. Then, the expected time for a memory access in a system with a TLB hit ratio of f is given by f × (Tc + Tm) + (1 − f) × (Tc + (n + 1)Tm).

• LRU (least recently used) policy is often used to evict entries from the TLB when it is running out of space. However, some entries (e.g., those corresponding to kernel code) are permanently stored in the TLB. Further, most TLB entries become invalidated after a context switch, because the mappings from virtual to physical addresses are different for different processes. To avoid flushing all TLB entries on a context switch, some TLB designs store a tag of the process ID in the TLB entry, and access only those TLB entries whose tag matches that of the current running process.

**Page table design**

* How are page tables stored in memory? Since the CPU uses just one register to store the starting address of the page table, one way is to store the page table contiguously beginning from the start address. However, it is hard to store page tables as large as 4MB contiguously in memory. Therefore, page tables are themselves split up and stored in pages. Suppose a PTE is of size 2 e bytes and pages are of size 2 k bytes. Then each page can hold 2 k−e PTEs. Now, the 2 m−k page table entries in the inner page table will be spread out over 2m−k 2 k−e = 2m+e−2k pages. And an outer page table stores the frame numbers of the pages of the inner page tables. That is, given the virtual address, the least significant k bits provide an offset in a page, the middle k−e bits will be used to offset into the inner page table to obtain the frame number, and the most significant m + e − 2k bits will be used to index into the outer page table to find the location (frame number) of the inner page table itself. For example, for 32-bit virtual addresses, 4KB pages, and 4-byte page table entries, the most significant 10 bits are used to index into the outer page table, and next 10 bits are used to index into the inner page table, to find the frame number. For this example, the outer page table fits snugly in a page. However, if the outer page table size gets larger, it further may need to be split, and several levels of hierarchy may be required to store the page table. This concept is called hierarchical paging, and quickly gets inefficient for large virtual address spaces (e.g., 64 bit architectures).
* An alternative design for large address spaces is a hashed page table. In this design, the virtual address (key) is hashed to locate its corresponding value entry. For multiple virtual addresses that hash to the same location, a linked list or some such mechanism is used to store all the mappings.
* Another alternative design for page tables in systems with large virtual address spaces is to have a table with entries for every physical frame, and not for every logical page. This is called an **inverted page table**. Every entry of the inverted page table maps a physical frame number to the process identifier, and the logical page number in the address space of that process. To translate a virtual address to a physical address, one would have to search through all the entries in the inverted page table until a matching logical page number is found. So, while inverted page tables save on size, lookup times are longer. Also, inverted page tables make it hard to share pages between processes.
* Some operating systems use a combination of segmentation and paging. The logical address space is divided into a small number of segments, and the logical address is the combination of the segment number and an offset within that segment. The CPU can only specify the offset part of the logical address, and based on which segment of the memory image it is executing in, the complete logical address can be generated. This logical address is then mapped to the physical address via paging. For example, Linux uses segmentation with paging on architectures that have support for segmentation (e.g., Intel x86). Linux uses a small number of segments: one each of user code, user data, kernel code, kernel data, stack segment, and so on. A segment table stores the starting (logical) addresses of each of these segments in the virtual address space of a process, and implements protection in terms of who can access that segment. For example, access to the kernel segments is allowed only when the CPU is in kernel mode.

**Demand Paging**

When are physical frames allocated to logical pages? In a simplistic case, all logical pages can be assigned physical frames right at the beginning when a process is created and brought into memory. However, this is wasteful for a number of reasons. For example, a process may access only parts of its memory during a run. Therefore, modern operating systems use a concept called demand paging. With demand paging, a logical page is assigned a physical frame only when the data in the page needs to be accessed. At other times, the data in the page will reside on secondary storage like a hard disk. A bit in the page table indicates whether the page is in memory or not. With demand paging, a process is allocated only as much physical memory as it needs, thereby allowing us to accommodate many more processes in memory.

• Suppose the CPU tries to access a certain logical address and the page table shows that the page containing that address has not been mapped into memory. Then a page fault is said to occur. A page fault traps the OS. The process moves into kernel mode. The OS then handles the page fault as follows: a free physical frame is identified, the required page is requested to be read from the disk, and the process that generated the page fault is context switched out. The CPU then starts running another process. When the data for the page has been read from disk into the free frame, an interrupt occurs, and is handled by whichever process currently has the CPU in its kernel mode. The interrupt service routine in the OS updates the page table of the process that saw the page fault, and marks it as ready to run again. When the CPU scheduler runs the process again, the process starts running at the instruction that generated the page fault, and continues execution normally.

Note that an instruction can be interrupted midway due to a page fault, and will be restarted from the beginning after the page fault has been serviced. Care should be taken in saving the context of a process to ensure that any temporary state that needs to be preserved is indeed preserved as part of the saved context. The instruction set of the underlying architecture must be carefully examined for such issues when adding paging to the OS.

Let Tm be the time to access main memory and Tf be the time to service a page fault (which is typically large, given that a disk access is required). If p is the page fault rate, then the effective memory access time is (1 − p) × Tm + p × Tf . Systems with high page fault rate see a high memory access time, and hence worse performance.

Note that most operating systems do not allow the kernel code or data to be paged out, for reasons of performance and convenience. For example, the operating system could never recover from a page fault, if the code to handle the page fault itself is paged out, and causes a page fault when accessed.

Some operating systems implement pre-paging, where a set of pages that are likely to be accessed in the future are brought into memory before a page fault occurs. The success of pre-paging depends on how well the page access pattern can be predicted, so as to not bring pages that will never be used into memory.

Note that an inverted page table design will be inefficient when demand paging is used, because one also needs to lookup pages that are not in memory, and an inverted page table does not have entries for such pages. Therefore, operating systems that use inverted page tables must also keep a regular page table around to identify and service page faults.

Demand paging allows for an over-commitment of physical memory, where the aggregate virtual addresses of all processes can be (much) larger than the underlying physical address space. Over-commitment is generally acceptable because processes do not tend to use all their memory all the time. In fact, most memory accesses have a locality of reference, where a process runs in one part of its virtual address space for a while (causing memory accesses in a small number of pages), before moving on to a different part of its address space. Therefore, over-commitment, when done carefully, can lead to improved performance and an ability to support a large number of processes. Operating systems that over-commit memory to processes are said to provide virtual (not physical) memory to processes. Note that virtual addressing does not necessarily imply virtual memory, though most modern operating systems implement both these ideas.

To implement virtual memory, an OS requires two mechanisms. It needs a frame allocation algorithm to decide how many physical frames to allocate to each process. Note that every process needs a certain minimum number of frames to function, and an OS must strive to provide this minimum number of frames. Further, when all physical frames have been allocated, and a process needs a new frame for a page, the OS must take a frame away from one logical page (that is hopefully not being actively used) and use it to satisfy the new request. A page replacement algorithm decides which logical page must be replaced in such cases

A process memory image is initially read from the file system. As the process runs, some pages may be modified and may diverge from the content in the file systems. Such pages that are not backed by a copy in the file system are called dirty pages. When a dirty page is replaced, its contents must be flushed to disk (either to a file or to swap space). Therefore, servicing a page fault may incur two disk operations, one to swap out an unused page to free up a physical frame, and another to read in the content of the new page. The page table maintains a dirty bit for every page, along with a valid bit.

demand paging (as opposed to anticipatory paging) is a method of virtual memory management. In a system that uses demand paging, the operating system copies a disk page into physical memory only if an attempt is made to access it and that page is not already in memory (i.e., if a page fault occurs). It follows that a process begins execution with none of its pages in physical memory, and many page faults will occur until most of a process's working set of pages are located in physical memory. This is an example of a lazy loading technique.

Consider a process executing on an operating system that uses demand paging. The average time for a memory access in the system is M units if the corresponding memory page is available in memory, and D units if the memory access causes a page fault. It has been experimental measured that the average time taken for a memory access in the process is X units. Give an expression for the page fault rate experienced by the process?

Given, average time for a memory access = M units if page hits, and average time for a memory access = D units if page fault occurred.

And total/experimental average time taken for a memory access = X units.

Let page fault rate is p. Therefore,

Average memory access time = ( 1 – page fault rate) \* memory access time when no page fault + Page fault rate \* Memory access time when page fault

→ X = (1 – p)\*M + p\*D = M – M\*p + p\*D

→ X = M + p(D – M)

**Page Replacement Algorithms**

1. FIFO (First In First Out) is the simplest page replacement algorithm. With FIFO, logical pages assigned to processes are placed in a FIFO queue. New pages are added at the tail. When a new page must be mapped, the page at the head of the queue is evicted. This way, the page brought into the memory earliest is swapped out. However, this oldest page maybe a piece of code that is heavily used, and may cause a page fault very soon, so FIFO does not always make good choices, and may have a higher than optimal page fault rate. The FIFO policy also suffers from Belady’s anomaly, i.e., the page fault rate may not be monotonically decreasing with the total available memory, which would have been the expectation with a sane page replacement algorithm. (Because FIFO doesn’t care for popularity of pages, it may so happen that some physical memory sizes lead you to replace a heavily used page, while some don’t, resulting in the anomaly.) The FIFO is the simplest page replacement algorithm, the idea behind this is **“Replace a page that page is the oldest page of all the pages of the main memory” or “Replace the page that has been in memory longest”.**
2. What is the optimal page replacement algorithm? One must ideally replace a page that will not be used for the longest time in the future. However, since one cannot look into the future, this optimal algorithm cannot be realized in practice. The optimal page replacement has the lowest page fault rate of all algorithms. The criteria of this algorithm is “**Replace a page that will not be used for the longest period of time”.**
3. The LRU (least recently used) policy replaces the page that has not be used for the longest time in the past, and is somewhat of an approximation to the optimal policy. It doesn’t suffer from Belady’s anomaly (the ordering of least recently used pages won’t change based on how much memory you have), and is one of the more popular policies. To implement LRU, one must maintain the time of access of each page, which is an expensive operation if done in software by the kernel. Another way to implement LRU is to store the pages in a stack, move a page to the top of the stack when accessed, and evict from the bottom of the stack. However, this solution also incurs a lot of overhead (changing stack pointers) for every memory access.

LRU is best implemented with some support from the hardware. Most memory hardware can set a reference bit when a page is accessed. The OS clears all reference bits initially, and periodically checks the reference bits. If the bit is set, the OS can know that a page has been accessed since the bit was last cleared, though it wouldn’t know when it has been accessed during that period. An LRU-like algorithm can be built using reference bits as follows: the OS samples and clears the reference bit of every page every epoch (say, every few milliseconds), and maintains a history of the reference bit values in the last few epochs in the page table. Using this history of references, the OS can figure out which pages were accessed in the recent epochs and which weren’t.

The Criteria of this algorithm is **“Replace a page that has not been used for the longest period of time”.** The strategy is “**Page replacement algorithm looking backward in time, rather than forward”.**

Finally, several variants of the simple FIFO policy can be created using the hardware support of reference bits. In the second chance algorithm, when the OS needs a free page, it runs through the queue of pages, much like in FIFO. However, if the reference bit is set, the OS skips that page, and clears the bit (for the future). The OS walks through the list of pages, clearing set bits, until a page with no reference bit set is found. The OS keeps this pointer to the list and resumes its search for free pages from the following page for subsequent runs of the algorithm. In addition to the reference bit, the second chance algorithm can also look at the dirty bit of the page. Pages that have not been recently accessed or written to (both reference and dirty bits are not set) are ideal candidates for replacements, and pages that have both bits set will not be replaced as far as possible.

1. Least Frequently Used (LFU) algorithm “**Selects a page for replacement, if the page has not been used often in the past”** or **“Replace page that page has smallest count”.** For this algorithm each page maintains a counter, which counter value shows the least count, replace the page. The reason for this selection is that an actively used page should have a large reference count. So don’t replace the activity used page.
2. Most Frequently Used Algorithm (MFU), the criteria of this algorithm is replace a page that has the maximum frequency count of all pages. The Implementation of this algorithm is fairly expensive.

**Text & Reference books:**

1. Operating Systems: Three Easy Pieces, Remzi H. Arpaci-Dusseau and Andrea C. Arpaci- Dusseau, Arpaci-Dusseau Books, May, (2014).