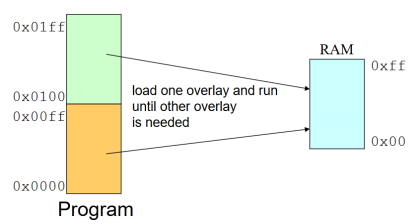
Operating Systems

Virtual Memory

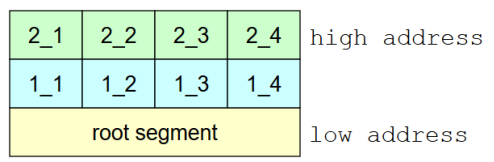
Large Programs

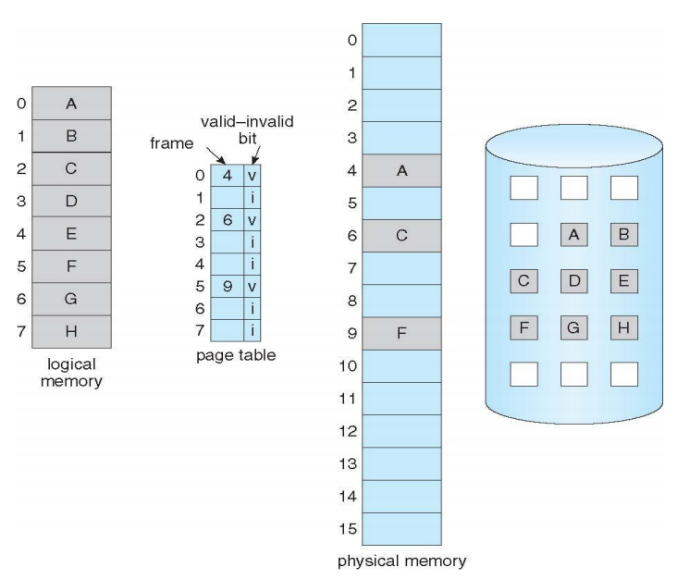
How do we deal with a program that is bigger than the available physical memory? Could we correctly execute a program even if it isn’t all in memory at all times?

Overlays

We can only load part of the program at any time. The programmer breaks the address space of the program into pieces that fit into memory (still constrained by physical memory size). These pieces are called overlays and are loaded and unloaded by the program.

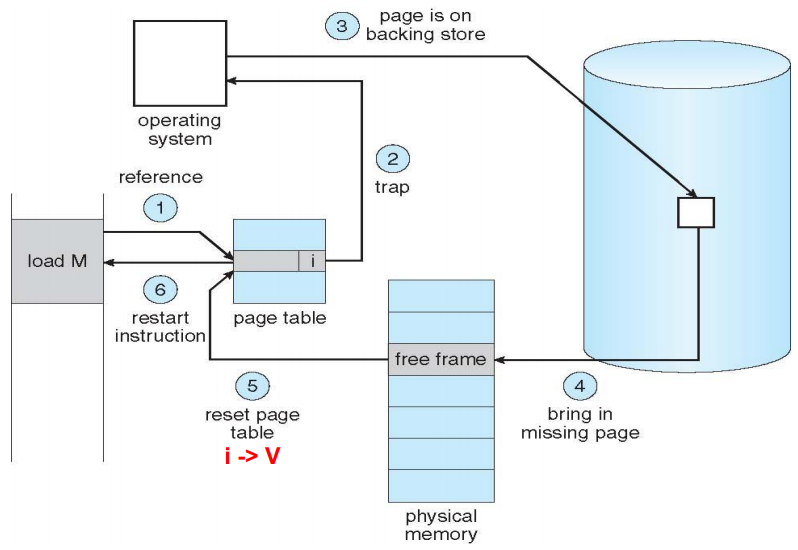
The overlay manager (part of the program, not the OS) handles loading an overlay when it is not in RAM and unloaded overlays previously in RAM.

The overlays mechanism invovles one root segment that always stays in RAM that includes the overlay manager and two or moer memory partitions. Each partition is associated with any number of overlay segments but only one overlay segment can be in a partition at a time.

Virtual Memory

Virtual memory fully decouples the address space from physical memory allowing a larger logical address space than physical memory. Paged virtual memory is mased on hardware with OS support and is transparent to the programmer requiring no extra work.

Not all the pages of an address space need to be in memory, the full address space is on the disk in page sized blocks and the main memory is used as a cache. We need to transfer pages to free page frames in memory in order to use them, if there aren’t any free page frames, we need to find a page to evict and replace.

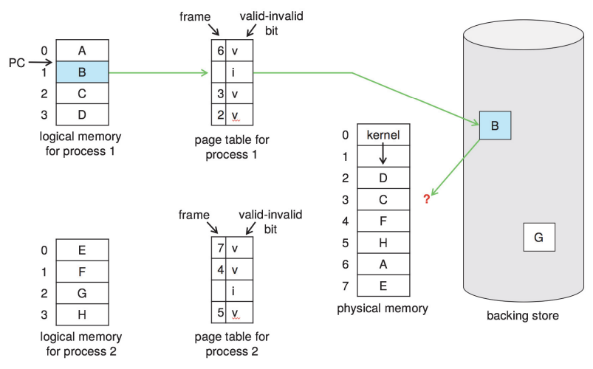
Page Fault

If a program attempts to access a page that is not in memory (the relative page table entry is invalid before the access) the hardware will trigger a page fault (exception). The operating system then checks internal data structures to ensure the reference was valid, if not it aborts, if it is then the OS finds a free frame and swaps the desired page into that frame via a scheduled disk operation. The OS then sets the internal data structures to indicate the page is now in memory (setting the valid bit) and restarts the instruction that caused the exception.

It’s important to note that only invalid references involve the OS, valid pages are accessed directly without OS involvement.

Demand Paging

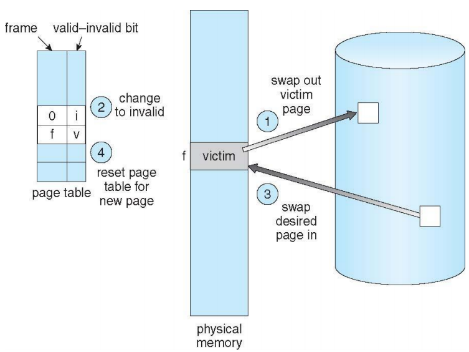
Demand paging is a scheme where pages are only brought into memory when a process attempts to access them (on the programs demand). This means that processes start with no pages and fault until they have the set of pages they need but that only the code/data actually needed by a process needs to be loaded (which will of course change over time). Very few systems try to anticipate future needs.

Pages may be clustered though, with the OS keeping track of pages that should come and go together brining them all in when on is reference.

Demand paging can be expensive and depends heavily on storage latency.

Page Allocation and Replacement

When you read in a page where does it go? If there are free frames, great then grab one (page allocation) but if there are no free frames we must evict one (page replacement).

The OS tries to keep a pool of free pages around to avoid the cost of eviction, but high degrees of multiprogramming can cause over-allocation where all memory is in use and eviction is necessary.

Page Replacement Algorithms

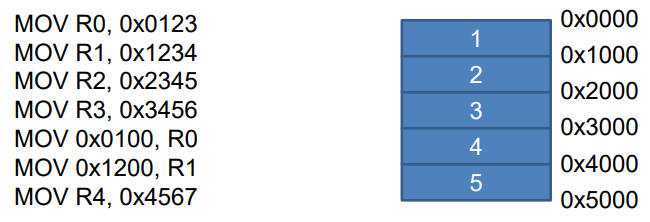
How do we choose which page to page to evict? We want to reduce the page-fault rate (and thus page-fault overhead) by selecting the best possible victim page. The best victim page is the one that will never be touched again (won’t be needed in the near future). Belady’s theorem states that evicting the page that won’t be used for the longest period of time will minimize the page fault rate.

We also want to evict unmodified pages first as there will be no need to write them back to the disk.

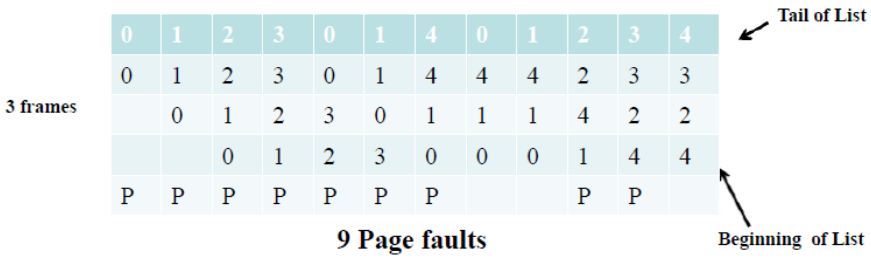
We will examine page replacement algorithms with the assumptions a process pages against itself and using a fixed number of page frames.

Example Memory Reference String

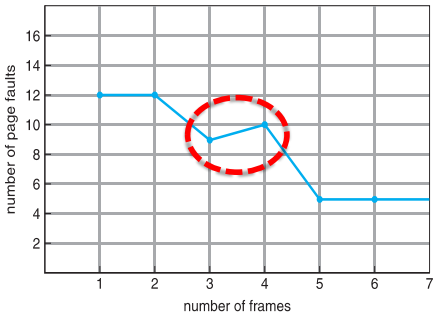
This is an ordered list of pages the program will reference.



First-In-First-Out (FIFO) Algorithm

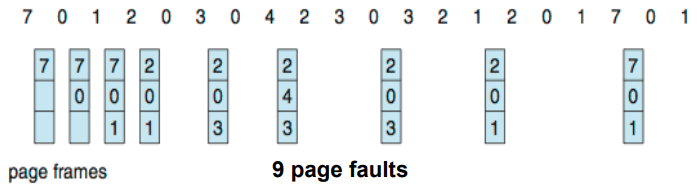
We replace the oldest page in memory. This is very easy to implement, we maintain a linked list of all the pages in the order they come into memory.

Our example uses 3 page frames, a 5 pages and the reference string: 0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4

Belady’s Anomaly

We expect the number of page faults to decrease as the number of frames available increases, but hit isn’t actually always the case for all algorithms, this is known ad Belady’s anomaly. The example to the right is using FIFO.

Optimal Algorithm

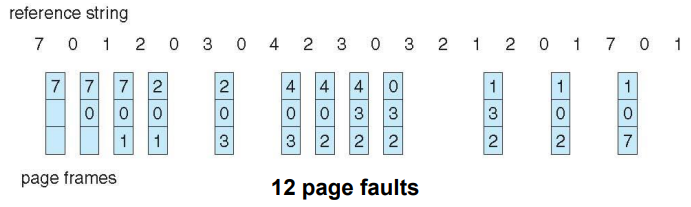
Replace the page that will not be used for the longest period of time. This will have the lowest page-fault rate and will never suffer from Belady’s anomaly.

In the example we have 3 page frames, 8 pages and the reference string:

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

But how will we know what page will not be used, we don’t know the future? In practice this is only used for measuring how well other algorithms perform.

Least Recently Used (LRU) Algorithm

Replace the page that has not been used the longest. Instead of trying to predict the future, we use past knowledge and assume the near future will be similar. This will never suffer from Belady’s anomaly either.

This is the same example as the optimal algorithm (just different algorithm.

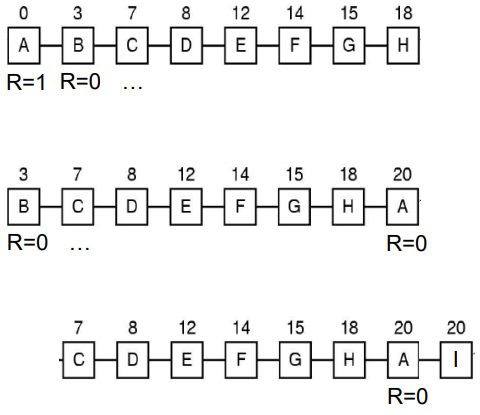
How do we implement this though, we could associate the time of last use with each page? This would require substantial hardware assistance.

Approximating LRU

Use the page table entry bits maintained by hardware indicate it the page has been referenced and iff the page has been modified. We can then keep a history/counter for each page in software.

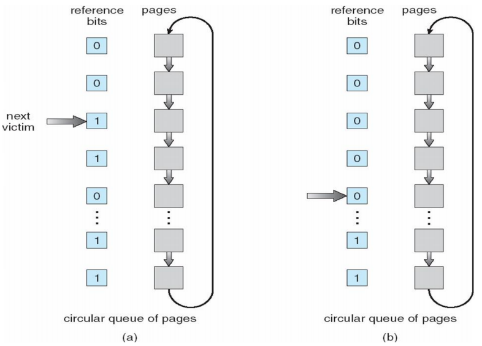
History-based page replacement algorithms record the reference bits at regular intervals and keep the history bits in a table in memory. These include aging, second-change (clock) and enhanced second-chance algorithms.

Counting-based page replacement algorithms keep a counter of the number of references that have been made and can choose least frequently used (LFU) or most frequently used (MFU) policies (MFU works on the assumption of initialisation code being used a lot at the beginning but then never again).

Second Chance

This is a variant of FIFO that adds the concept of usage (references). We examine the pages in FIFO order starting from the beginning of the list, at each page we consider the “reference bit”, if it’s 0 it hasn’t been referenced and we remove the page adding the new page to the end of the FIFO, if it’s 1 then we set it to 0 and place it at the end of the FIFO list (second chance) and start again.

If all of the pages have been used then on the second pass this simply reverts to pure FIFO.

Second Chance Clock

The second chance algorithm described will end up pushing and popping a lot of elements creating overhead. The second chance clock algorithm avoids this by storing all the pages in a circular list and keeping a next victim pointer, incrementing that instead.

Frames Among Processes

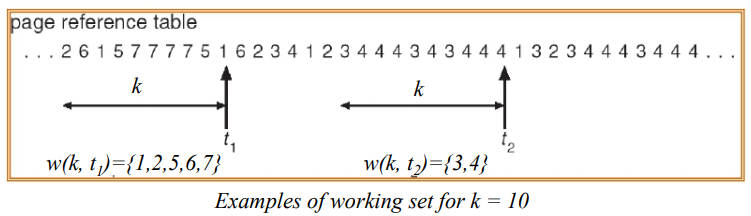
There are many ways to allocate frames between processes, equally proportionally, each have their benefits.

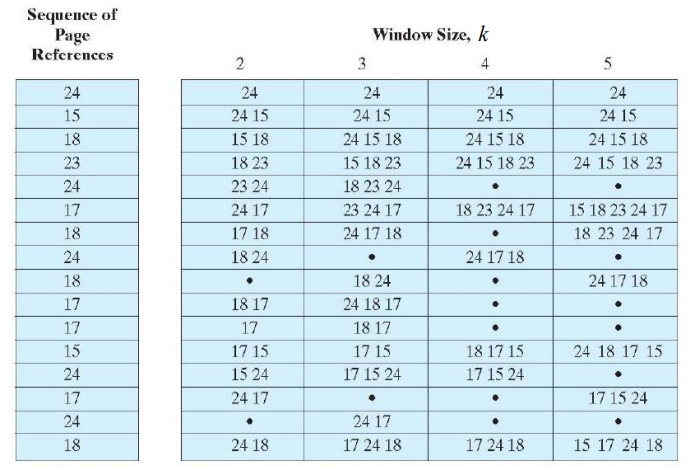
There are also Two main ways to go about frame replacement:

* Local: each process is given a limit of pages it can use, the process can then only page against itself (evict its own pages) leaving the other processes be. This has poor utilisation of (all) free page frames and has long access times.
* Global: the victim page is chosen from among all the page frames regardless of owner, the processes’ page frame allocation can vary dynamically but his also introduces a risk of global thrashing.

The Working Set Model

How many pages does a program really need?

The workings set of a process is used to model the dynamic locality of its memory usage. The working set is the set of pages the process currently ‘needs’ defined as:

WS(k,t) = {pages referenced in the time interval (t, t-k)} where t is the time, k is the working set window (measured in page refs) and a page is in WS iff it was referenced in the last k references.

The working set varies over the life of the program (as does the working set size). This relies heavily on k:

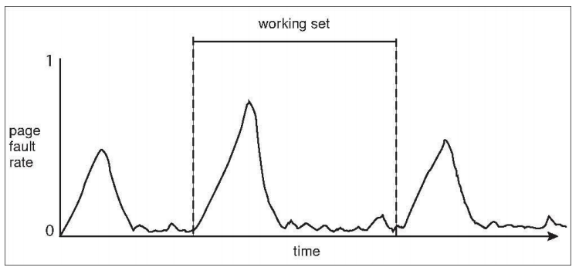
Working Set Size

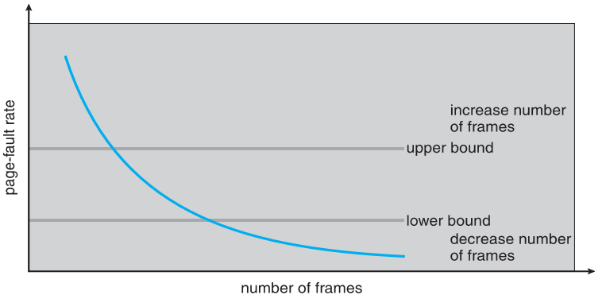
The working set size |WS(k,t)| changes with the program locality. During periods of poor locality more pages are referenced and the working set size is larger.   
The working set must all be in memory to avoid heavy faulting and thrashing.

(Hypothetical) Working Set Allocation Algorithm

Estimate |WS(k,0)| for a process and allow the process to start only if the OS can provide that many frames. Then use a local replacement algorithm making sure that the working set occupies the process’s frames. Tracking each process’s workings et size and re-allocating page frames among processes dynamically.

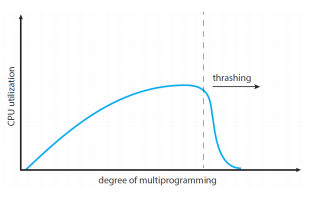
How do we keep track of WSs? We could use reference bits with a fixed-interval timer interrupt.

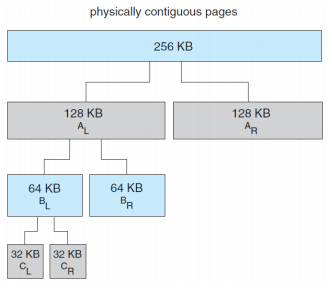
Page-Fault Frequency Allocation

The relationship between the working set and the page-fault rate of a process is such that the working set will change over time and the page-fault rate will peak and then valley as this happens. We can use the page-fault rate/frequency to steer allocations.

We establish an acceptable page-fault frequency (PFF) rate and use a local replacement algorithm, if the page fault rate is too low the process will lose a frame, if it’s too high it will gain a frame.

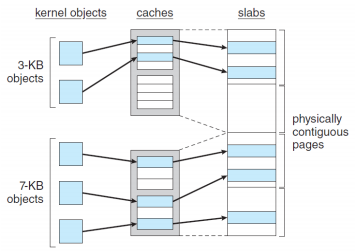
Thrashing

Trashing occurs when the system spends most of its time servicing page faults and little time doing actual useful work. It could be that there is enough memory but a poor replacement algorithm incompatible with program behaviour. Or it could be that the memory is over-committed and the OS sees the CPU being poorly utilised (do to everything waiting for pages) and adds more processes making the problem worse creating even more processes requesting memory.

Kernel Memory Allocation: Buddy System

The kernel allocates memory for applications, in order to allocate this memory they kernel can use different algorithms, one of which is the buddy system.

The buddy system allocates the closest (above) power of 2 physcially contiguous pages. If the request (rouned to the nearest power of 2) is smaller than any of the currently available buddies, then we repeated break on down until we have one (or in reality two) of the correct size. Addtionally two equal size free continguous buddies may be coalesced into one.

Kernel Memory Allocation: Slab Allocation

Another system is known as slab allocation. A slab is made up of one or more phycsially contiguous pages and a chache consits of one or more slabs. There is a chache for each unique kernel data strcutre eche populated objects (instantiations fo the kernel data structure). If there are free slabes the allocation is immediate if not then we need to search for memroy space.

There are several variations of these in linux SLOB AND SLUB (more performant).

CPU Cache Vs Virtual Memory ‘As A Cache’

CPU cahce is a hardware component that is completely transparent to the programmer. It holds dta coming from memory and the CPU cache fetches data from the memory completely transparently.

Virtual memory ‘as a cache’ is a combination of hardware and the OS, it is transparent to the application but not to the OS. Virtual memoroy ‘as a cache’ hodles data coming from storage and the OS moves memory from storage to memory. 