Virtual Memory

Introduction

So far, we separated the programmer's view of memory from that of the operating system using a mapping mechanism. This allows the OS to move user programs around and simplifies sharing of memory between them.

However, we also assumed that a user program had to be loaded completely into the memory before it could run.

Problem: Waste of memory, because a program only needs a small amount of memory at any given time.

Solution: **Virtual memory**; a program can run with only some of its virtual address space in main memory.

Principles of operation

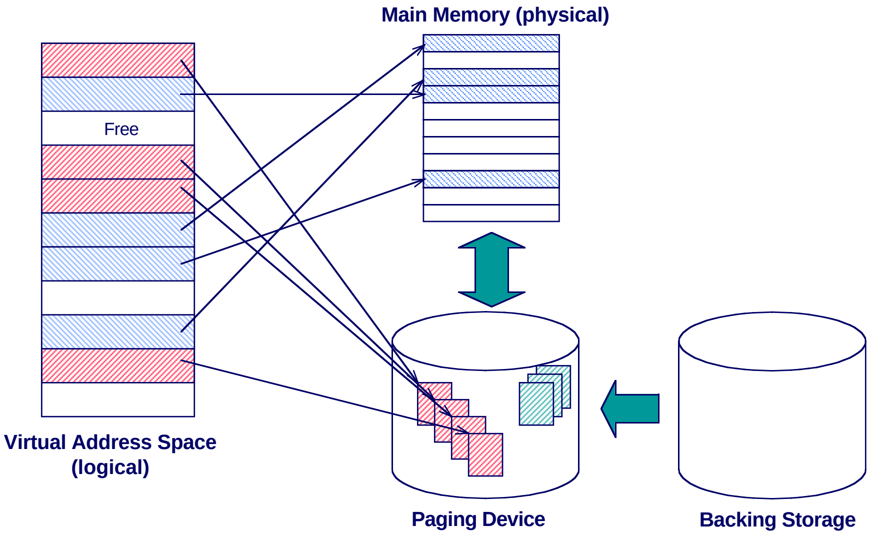
The basic idea with virtual memory is to create an illusion of memory that is as large as a disk (in gigabytes) and as fast as memory (in nanoseconds).

The key principle is *locality of reference,* which recognizes that a significant percentage of memory accesses in a running program are made to a subset of its pages. Or simply put, a running program only needs access to a portion of its virtual address space at a given time.

With virtual memory, a **logical (virtual) address** translates to:

- *Main memory* (small but fast), or
- *Paging device* (large but slow), or
- *None* (not allocated, not used, free.)

A *virtual* view



Virtual memory

Virtual memory (sub-)system can be implemented as an extension of paged or segmented memory management or sometimes as a combination of both.

In this scheme, the operating system has the ability to execute a program which is only *partially loaded* in memory.

Note*: the idea was originally explored earlier in “overlays”. However now, with virtual memory, the fragmentation and its management is done by the operating system.*

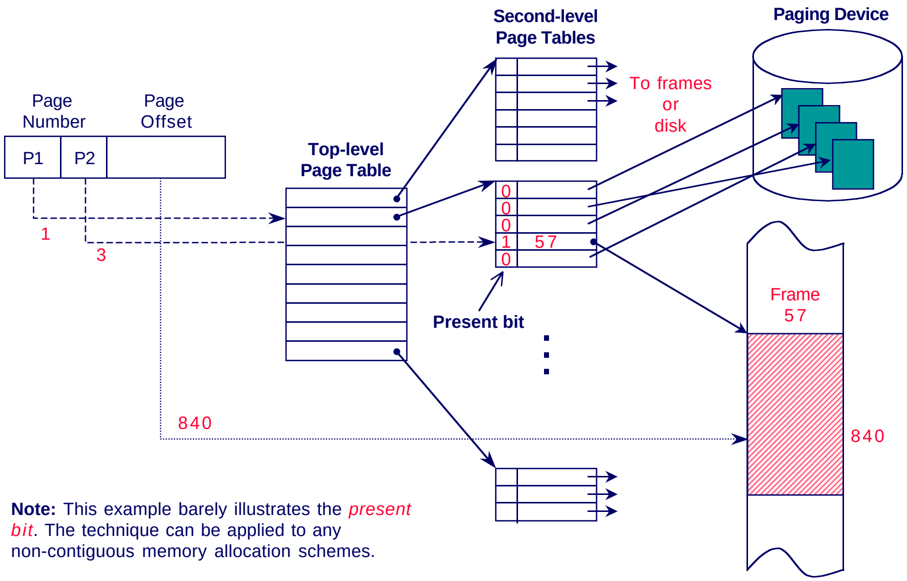
Missing pages

What happens when an executing program references an address that is *not* in main memory? Here, both hardware (H/W) and software (S/W) cooperate and solve the problem:

- The page table is extended with an extra bit, *present.* Initially, all the present bits are cleared (H/W and S/W).
- While doing the address translation, the MMU checks to see if this bit is set. Access to a page whose present bit is *not* set causes a special hardware trap, called *page fault (H/W)*.
- When a page fault occurs the operating system brings the page into memory, sets the corresponding present bit, and restarts the execution of the instruction (S/W).

Most likely, the page carrying the address will be on the paging device, but possibly does not exist at all!

Multi-level paging—*revisited*



Note: This example barely illustrates the *present bit*. The technique can be applied to any non-contiguous memory allocation schemes.

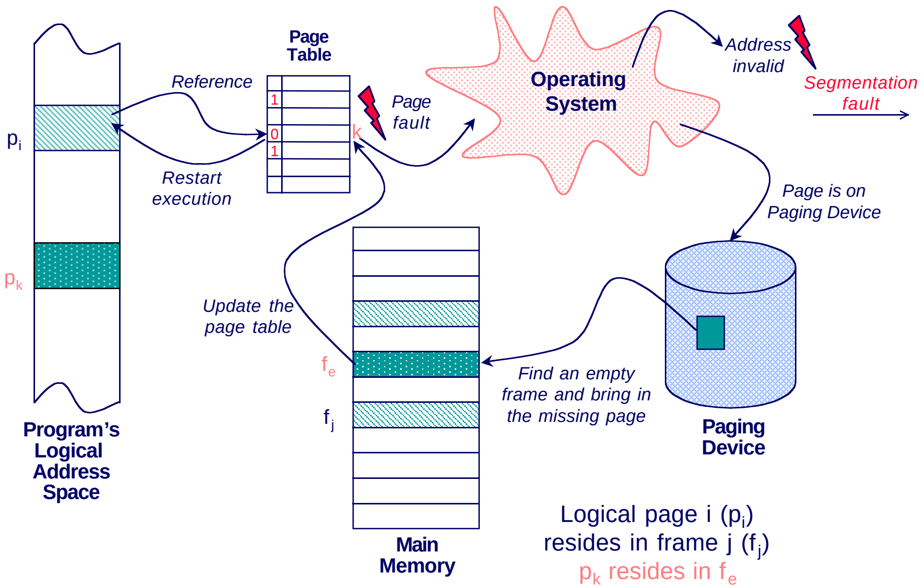
Page fault handling—by words

When a page fault occurs, the system:

- marks the current process as blocked (waiting for a page),
- finds an empty frame or make a frame empty in main memory,
- determines the location of the requested page on paging device,
- performs an I/O operation to fetch the page to main memory,
- triggers a “page fetched’’ event (e.g., special form of I/O completion interrupt) to wake up the process.

Since the fourth (and occasionally the second) step involves I/O operations, it makes sense to perform this operation with a special system process (e.g., in UNIX, **`pager`** process.)

Page fault handling—by picture



Logical page i (p_i) resides in frame j (f_j)
p_k resides in f_e

Additional hardware support

Despite the similarities between paging or segmentation and virtual memory, there is a small but an important problem that requires additional hardware support. Consider the following M68030 instruction:

```
DBEQ    D0, Next  ; Decrement and Branch if Equal
```

Which can be micro-coded as:

```
Fetch (instruction); decrement D0;
if D0 is zero, set PC to “Next” else increment PC.
```

What if the instruction itself and the address Next are on two different pages and the latter page was not in memory? *Page fault...* From where and how to restart the instruction? (*Note:* D0 is already decremented.)

Possible support

The moral of the previous example is that if we want to have complex instructions and virtual memory, we need to have additional support from the hardware, such as:

- Partial effects of the faulted instruction are *undone* and the instruction is *restarted* after servicing the fault (VAX-11/780)
- The instruction *resumes* from the point of the fault (IBM-370)
- Before executing an instruction make sure that all referenced addresses are available in the memory (for some instructions, CPU generates all page faults!)

In practice, some or all of the above approaches are combined.

Basic policies

The hardware only provides the basic capabilities for virtual memory. The operating system, on the other hand, must make several decisions:

- *Allocation*—how much real memory to allocate to each (*ready)* program?
- *Fetching*—when to bring the pages into main memory?
- *Placement*—where in the memory the fetched page should be loaded?
- *Replacement*—what page should be removed from main memory?

Allocation policy

In general, the allocation policy deals with conflicting requirements:

- The fewer the frames allocated for a program, the higher the page fault rate.
- The fewer the frames allocated for a program, the more programs can reside in memory; thus, decreasing the need of swapping.
- Allocating additional frames to a program beyond a certain number results in little or only moderate gain in performance.

The number of allocated pages (also known as *resident set size*) can be *fixed* or can be *variable* during the execution of a program.

Fetch policy

- **Demand paging**
  - Start a program with no pages loaded; wait until it references a page; then load the page (this is the most common approach used in paging systems).
- **Request paging**
  - Similar to overlays, let the user identify which pages are needed (not practical, leads to over estimation and also user may not know what to ask for.)
- **Pre-paging**
  - Start with one or a few pages pre-loaded. As pages are referenced, bring in other (not yet referenced) pages too.

Opposite to fetching, the *cleaning policy* deals with determining when a modified (dirty) page should be written back to the paging device.

Placement policy

This policy usually follows the rules about paging and segmentation discussed earlier.

Given the matching sizes of a page and a frame, placement with paging is straightforward.

Segmentation requires more careful placement, especially when *not* combined with paging. Placement in pure segmentation is an important issue and *must* consider “free” memory management policies.

With the recent developments in *non-uniform memory access (NUMA)* distributed memory multiprocessor systems, placement becomes a major concern.

Replacement policy

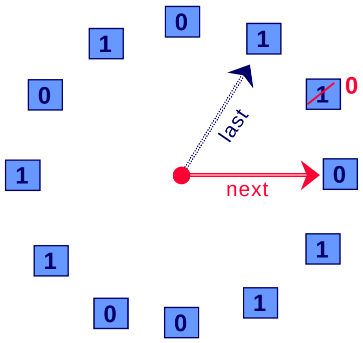
The most studied area of the memory management is the replacement policy or victim selection to satisfy a page fault:

- FIFO—the frames are treated as a circular list; the oldest (longest resident) page is replaced.
- LRU—the frame whose contents have not been used for the longest time is replaced.
- OPT—the page that will not be referenced again for the longest time is replaced (*prediction of the future; purely theoretical, but useful for comparison.*)
- Random—a frame is selected at random.

More on replacement policy

- *Replacement scope*:
  - Global—select a victim among all processes.
  - Local—select a page from the faulted process.
- *Frame locking*—frames that belong to resident kernel, or are used for critical purposes, may be locked for improved performance.
- *Page buffering*—victim frames are grouped into two categories: those that hold unmodified (clean) pages and modified (dirty) pages (VAX/VMS uses this approach.)

🕐 algorithm



This is also known as *2nd chance* algorithm.

All the frames, along with a *used* bit, are kept in a circular queue. A pointer indicates which page was just replaced. When a frame is needed, the pointer is advanced to the first frame with a zero used bit. As the pointer advances, it clears the used bits. Once a victim is found, the page is replaced and the frame is marked as used (i.e., its used bit is set to one.)

The hardware, on the other hand, sets the used bit each time an address in the page is referenced.

Used ==> accessed (read or written)

Clock algorithm—some details

Some systems also use a “dirty bit” (memory has been modified) to give preference to dirty pages.

Why? It is more expensive to victimize a dirty page.

Problem: code pages are clean, but…

If the clock hand is moving

- fast ➞ not enough memory (thrashing is possible!)
- slow ➞ not many page faults (system is lightly loaded)

BSD UNIX (e.g., SunOS up to release 4.1.4) uses clock algorithm. **`vmstat`** command gives some details about virtual memory.

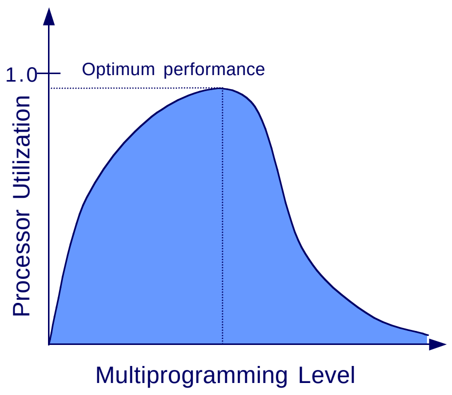
Thrashing

The number of processes that are in the memory determines the *multiprogramming (MP) level*. The effectiveness of virtual memory management is closely related to the MP level.

When there are just a few processes in memory, the possibility of processes being blocked and thus swapped out is higher.

When there are far too many processes (i.e., memory is over committed), the resident set of each process is smaller. This leads to higher page fault frequency, causing the system to exhibit a behavior known as *thrashing*. In other words, the system is spending its time moving pages in and out of memory and hardly doing anything useful.

Thrashing continued



The only way to eliminate thrashing is to reduce the multiprogramming level by suspending one or more process(es). Victim process(es) can be the:

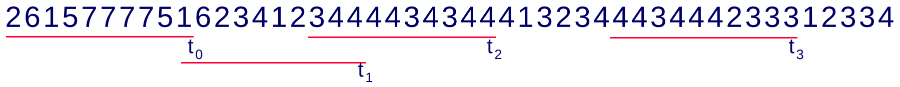
- lowest priority process
- faulting process
- newest process
- process with the smallest resident set
- process with the largest resident set

Student analogy to thrashing: Too many courses!
Solution? *Drop one or two…Well, it is too late now!*

Working sets continued

The *working set* of a program is the set of pages that are accessed by the last D memory references at a given time *t* and denoted by *W(t*,D).

Example (D=10):


26157777516234123444434344413234443444233312334

t_0 t_1 t_2 t_3

W(t_0, D) = {1,2,5,6,7} W(t_1, D) = {1,2,3,4,6} W(t_2, D) = {3,4} W(t_3, D) = {2,3,4}

Denning's *Working Set Principle* states that:

- A program should run iff its working set is in memory.
- A page may not be victimized if it is a member of the current working set of any runable (not blocked) program.

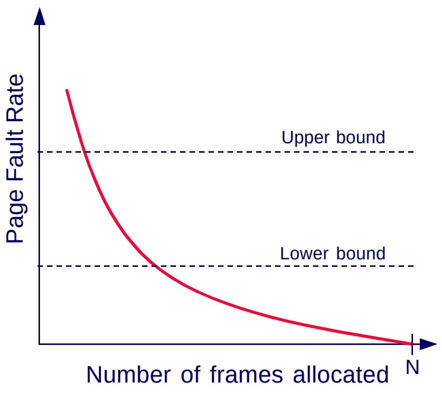
Working sets continued

One problem with the working set approach is that the information about each working set (one per process) must constantly be gathered (i.e., what pages have been accesses in the last D seconds?)

A solution (along with the clock algorithm):

- Maintain idle time value (amount of CPU time received by the process since last access to the page)
- Every once in a while (e.g., every few seconds), scan all pages of a process. For each used bit on, clear page's idle time; otherwise, add process' CPU time since the last scan to idle time. Turn off all the used bits.
- The collection of pages with the lowest idle time is the working set.

Page-fault frequency



T_i = average time between page faults for process P_i

Dealing with the details of working sets usually incurs high overhead. Instead, an algorithm known as *page-fault frequency*, can be used to monitor thrashing. The page fault rate is defined as

P = 1 / T

where, T is the *critical inter-page fault time*. When the process runs below the lower bound, a frame is taken away from it (i.e., its resident set size is reduced). Similarly, an additional frame is assigned to a process which runs above its upper bound.

Current trends

- Larger physical memory
  - page replacement is less important
  - better hardware support for replacement policies
  - larger page sizes
    - better TLB coverage
    - smaller page tables, fewer pages to manage
- Larger address spaces
  - sparse address spaces
  - single (combined) address space (part for the OS part for the user processes)
- File systems using virtual memory
  - memory mapped files
  - file caching with VM

A case study—paging in BSD UNIX

Page fault handling is separated from page replacement.

Page fault also occurs when there is plenty of memory available. Instead of loading a program for execution, the system simply builds its logical address space with all pages marked as invalid.

Program's image (code and data) is fetched into memory in response to page faults.

Dynamic data and stack are allocated as the program uses them.

Consequently, in many cases, the system does not have to look for a victim page when there is a page fault.

Paging in BSD UNIX continued

The BSD paging system was first implemented on the VAX with no used bit.

Page faults are serviced by a special system function, **`pagein`**, which executes in the context of the faulting process, with kernel privileges.

`pagein` attempts to get a free frame from the free-frame list maintained by the kernel. It the list is empty, the process blocks waiting for free memory.

The "free memory" event is raised by a system process, called the *page daemon*, (**`pager`**) when it reclaims some frames.

Paging in BSD UNIX continued

The page daemon remains dormant until the number of free frames drops a certain threshold value (e.g., 5% of the total number of frames.)

When the page daemon is awakened, it executes a variation of the clock algorithm.

When the page daemon cannot catch-up with memory demands (thrashing!), the system starts swapping some processes out.