Allocating Primary Memory To Processes

The important task of allocating memory to processes, and efficiently ensuring that processes have their instructions and data in main memory, is termed *memory management*.

An important goal of the operating system is to keep as many processes executable as possible, and it will only achieve this when processes have available the memory they require.

For simplicity, we'll initially assume that a process's *image* occupies a contiguous region of main memory.

Requirements of Memory Management

Surveys of various memory management schemes, all show that they address five major requirements; following the order of Stallings, we see:

- **Relocation:**
  In a multiprogramming system, the execution of a single process is often unrelated to others. When a program is first created, it is difficult (if not impossible) to know where its process will be placed when initially loaded into memory.

  Similarly, when a process is *swapped-out* (`Suspended`), it is unrealistic to expect that the process will be *swapped-in* at exactly the same memory location.

  Memory management has the role of determining where either instructions or data are located, i.e. how a process's memory references (requests) translate into actual physical memory addresses.

Requirements of Memory Management, *continued*

- **Protection:**
  Each process must be protected from either accidental or deliberate "interference" from other processes. Although compilers for high-level programming languages offer some support (e.g. constrained control flow, static array bound references), most data references are dynamic (array access and pointer following).

  Memory references made by a process must be checked (at run-time) to ensure that they lie within the bounds of memory allocated by the operating system.

  Checks are performed by hardware at run-time, and invalid references generate an *access violation* interrupt, trap or exception, for the operating system software to handle.

  "...the memory protection must be performed by the processor (hardware) rather than the operating system (software). This is because the operating system cannot anticipate all the memory references that a program will make. Even if this were possible, it would be prohibitively time-consuming to screen each program in advance for possible memory violations."

  [Stallings, p304]

Under LINUX we see invalid memory references for data reported via *segmentation violations* (signal number 11), and invalid memory references for instructions (or invalid instructions!) reported via *bus errors* (signal 10).

Requirements of Memory Management, *continued*

- **Sharing:**
  Even with protection, there is an equal need to allow processes to *share* memory.

  Multiple processes running the same program can share the (read+execute only) instructions, and co-operating processes may wish to share and communicate via memory containing data structures.

- **Logical Organization:**
  Although processes in memory often occupy linear sequences of addresses, programs are seldom written this way.

  Structured programming, and more recently object-oriented, techniques enforce programming using *modules*, which are developed and compiled independently.

  Ideally, all references from one module to another are resolved at run-time, maintaining their independence (termed *late binding*).

- **Physical Organization:**
  The relationship between primary memory (RAM) and secondary memory (disk) is clear, but not one programmers wish to manage. Old techniques termed *overlays* permitted reuse of a process's memory, but they were unnecessarily complex.

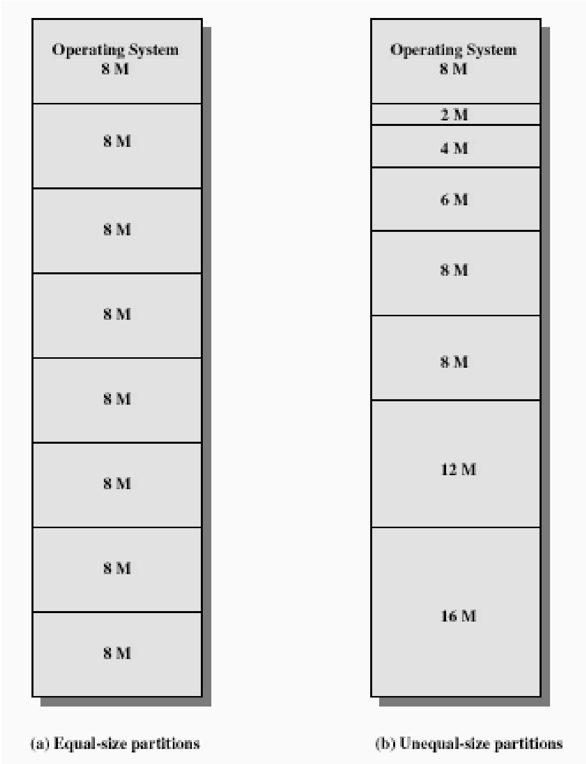
  Moreover, in a multiprogrammed system, the programmer cannot predict the size nor location of a process's memory. The task of moving information between main and secondary memory is clearly the task of the operating system.

Initial Memory Allocation Using Partitioning

In modern operating systems offering memory management, the operating system itself occupies a (fixed?) portion of main memory, and the remainder is available for multiple user/application processes.

The simplest technique is to consider main memory being in fixed-sized *partitions*, with two clear choices –

equal sized partitions or *unequal sized* partitions.



Any new process whose size is less than or equal to a partition's size, may be loaded into that partition.

Memory Allocation Using Partitioning, *continued*

Equal size partitions introduce two problems:

1) a process's requirements may exceed any partition size, or

2) a small process will still occupy a full, large, partition. Such wastage of memory is termed *internal memory fragmentation*.

The initial choice of partition, the *placement algorithm*, is trivial with equal-sized partitions.

Unequal sizes partitions offer obvious advantages, but complicate the placement algorithm.

Either:

1) a process is placed in the smallest partition, to minimize internal fragmentation, or

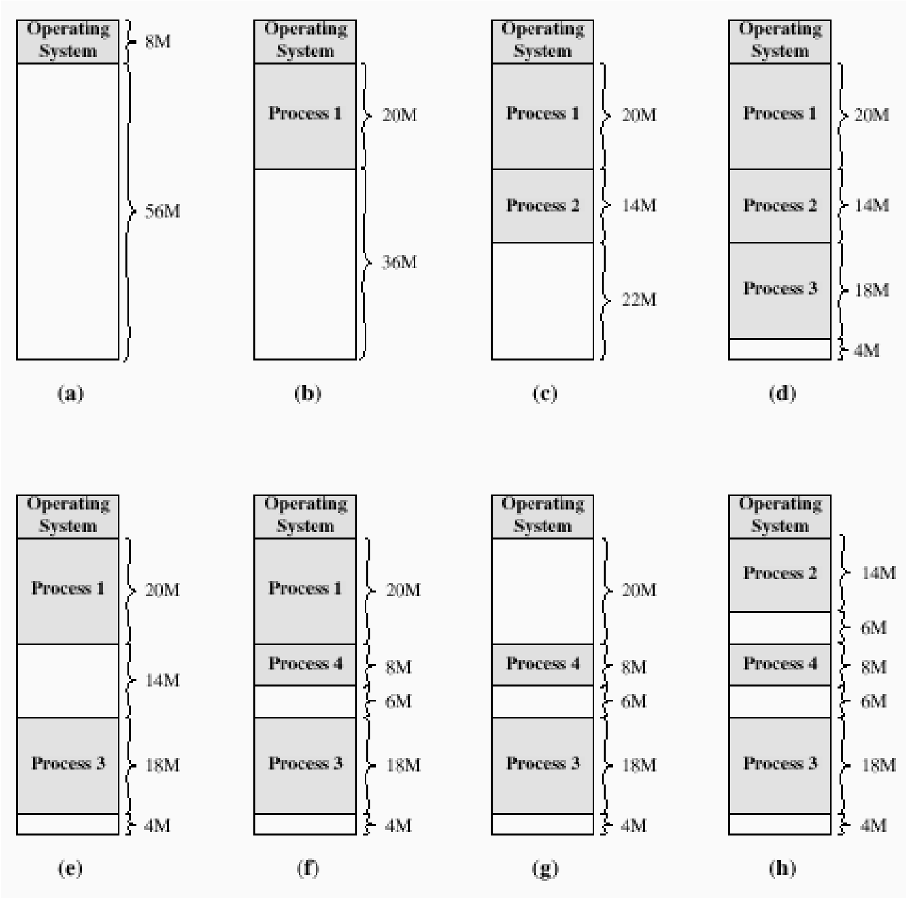
2) a process is placed in the smallest *available* partition.

The initial placement algorithm is again simple, but also introduces excessive internal fragmentation.

Dynamic Memory Partitioning

Dynamic partitioning overcomes some shortcomings of fixed partitioning – partitions are of variable length and number.

When a process commences, it occupies a partition of exactly the required size, and no more.



As this diagram shows, dynamic partitioning introduces the problem of *external fragmentation*, where there is insufficient contiguous memory to hold a new, large, process.

Dynamic Partitioning Placement Algorithms

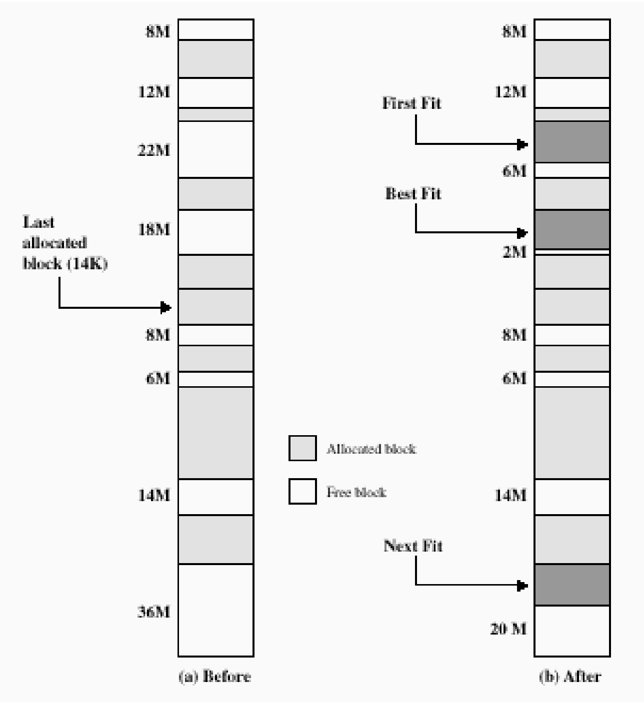
The obvious question suggested by dynamic partitioning is *"Where do we place a new process?"*

Three simple algorithms exist which all choose a block equal to or larger than the new process's requirements:

- *First-fit* – find the first unused block of memory (from address 0) that can contain the process,

- *Best-fit* – find the smallest unused block that can accept the process, or

- *Next-fit* – remember where the last process's memory was allocated, and circularly allocate the next sufficient block.



The Need for Address Relocation

The simpler memory management schemes seen so far have had one significant, simplifying, assumption – that when a process is swapped-out, it will always be swapped-in back to the same memory locations.

This assumption actually complicates the memory management task, and contributes to memory fragmentation.

We need to define three terms:

- A *logical address* – a reference to a memory location independent of any current assignment of data to main memory.

- A *relative address* – a logical address expressed relative to a fixed (logical) location, such as the beginning of the program.

- A *physical address*, or *absolute address*, is the actual location in main (physical) memory.

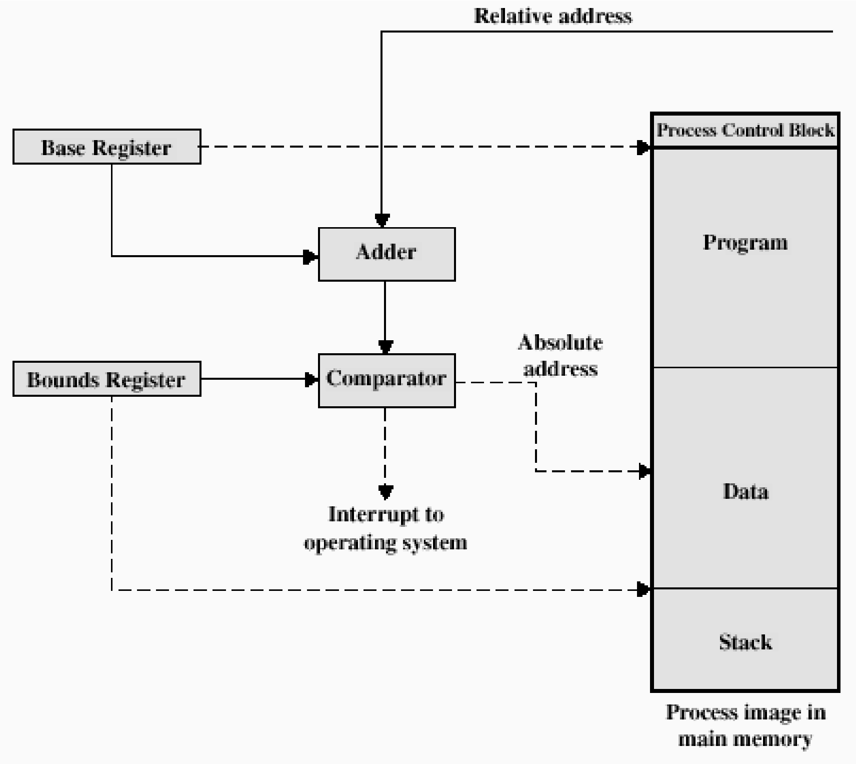
We've previously assumed that when a process is initially loaded (from disk), its relative addresses are replaced by the absolute addresses to be used.

More realistically, we enable processes to be swapped-in to any feasible range of physical memory – and this location is unlikely to be the same as before.

Hardware Address Translation

If a process may be swapped-in (back) to a different range of physical addresses, we need to update its relative addressing.

We could have software *modify* all addresses found, or have hardware *translate* all addresses accessed.

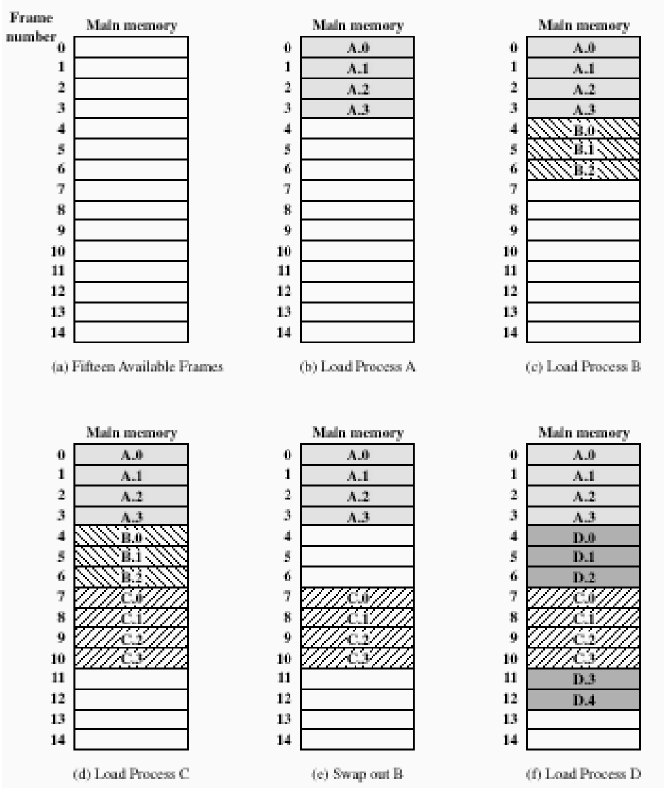


While a process is executing, the *base register* indicates the beginning of the process's partition, and the *bounds register* indicates the partition's extent.

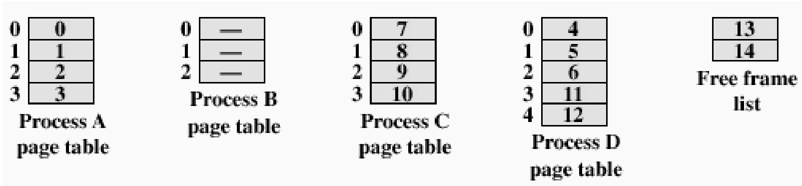
The base and bound (hardware) register pair must be saved and restored as each process is swapped-out and back -in.

Simple Paging of Memory

Fixed-sized partitions introduce internal fragmentation, and variable-sizes partitions introduce external fragmentation. However, if we use fixed-sized fragments that are very *small*, we'll only see minimal internal fragmentation within the last small fragment, and no external fragmentation.



We term the small, equally sized "chunks" of a process's memory as *pages*, and place them in equal-sized '"chunks" of the main memory, variously termed *frames*, or *page frames*.



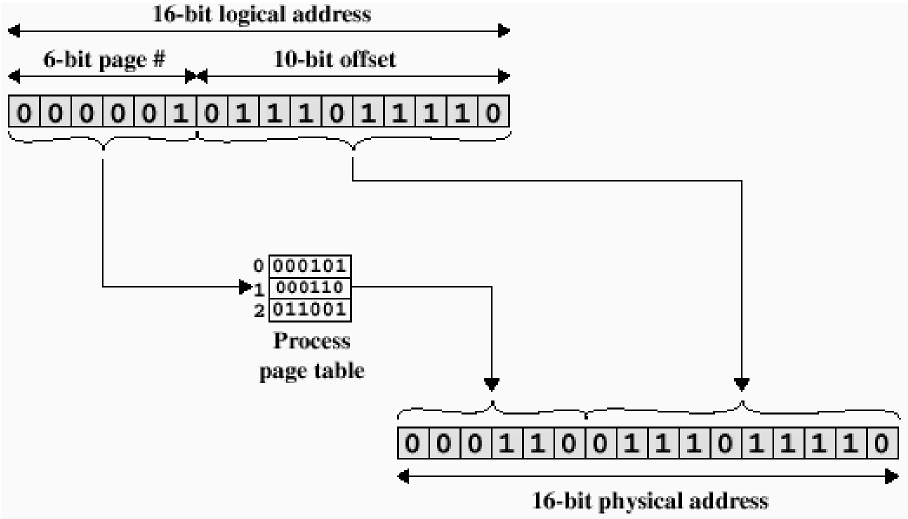
Simple Paging of Memory, *continued*

We'd like to remove the restriction (the assumption) that a process's sections of memory be must contiguous.

Clearly the single base register is insufficient – we need a number of base registers, one for each page, to identify the starting address of that page. (We do not need bounds registers; Why?)

We have the operating system maintain a set of special page registers, or a *page table*. The page table shows the frame (physical) location for each page. Within each process, a logical address now consists of a *page number*, and *an offset* within that page's frame.

In the following example, 6 bits from each 16-bit logical address indicate which *page table entry* to use. The remaining 10 bits of the logical address are *appended* to the contents of the page table entry to provide the actual, physical address:



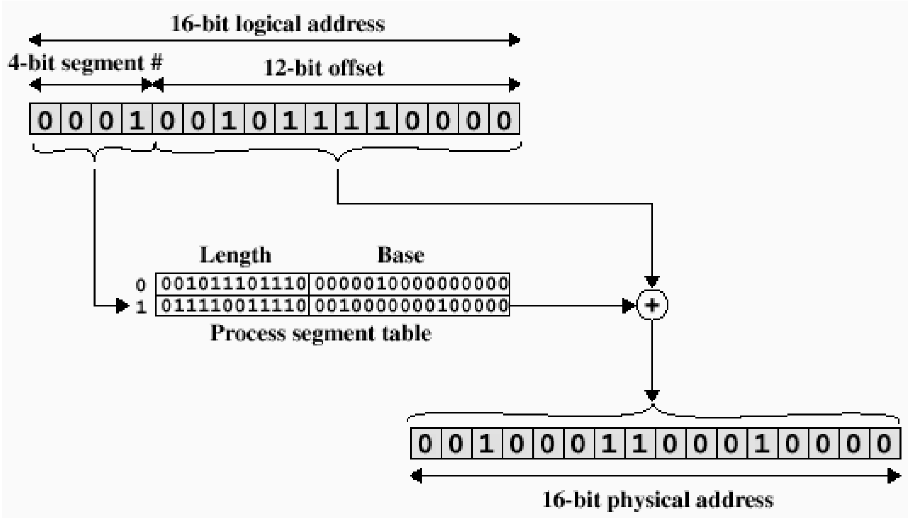
The logical→physical mapping can still be performed in hardware, provided that hardware knows how to access the page table of the current process.

Simple Segmentation of Memory

Another simple scheme under which we can allocate memory to processes is with *memory segmentation*.

Under segmentation, not all chunks are the same size, but each has its own maximum size.

Because segments are of differing lengths, the *segment table* in hardware must now also record the extent of the segment (hmmm, base and bounds).



Each segment could be a distinct logical entity, such as a function, an object in an O-O programming language, stack, read-only data, write-once data, ...

As segments need no longer commence on addresses that are powers-of-2, the hardware must perform a true (slower) *arithmetic addition* during the logical→physical address translation.

The Principal of Referential Locality

Numerous studies of the memory accesses of processes have observed that memory references cluster in certain parts of the program – over long periods, the centres of the clusters move, but over shorter periods they are fairly static.

For most types of programs, it is clear that:

- Except for infrequent branches and function/procedure invocation, program execution is sequential. The next instruction to be fetched typically follows the last executed.
- Programs generally operate at the same "depth" of function-invocation. References to instructions cluster within (and between) a small collection of functions.
- Most iterative control flow (looping) is over short instruction sequences. Instructions from the same memory locations are fetched several times in succession.
- Access to memory locations holding data is, too, constrained to a few frequently required data structures, or sequentially steps through memory (when traversing arrays).

With reference to paging and segmentation schemes, this *locality of reference* suggests that the *next* memory reference will very likely be from the same page, or segment, as the last.

This will impact heavily on our next enhancement to memory management – the use of *virtual memory*.

Paging and Segmentation .vs. Partitioning

When we compare paging and segmentation with the much simpler technique of partitioning, we see two clear benefits:

1) As processes are swapped-out and then back in, they may occupy different regions of physical memory.

   This is possible because hardware efficiently translates each logical address to a physical address, at run-time.

   The operating system's memory management software manipulates the hardware (page or segment table registers) to facilitate the translation.

2) A process is broken into either pages or segments, and these need not be contiguous in memory.

In combination with the principle of program locality, we now have a significant breakthrough:

If the above two characteristics are present, then it is not necessary for all pages (or all segments) to be in memory at any one time during a process's execution.

Advantages of Paging and Segmentation

Execution of any process can continue provided that the instruction it next wants to execute, or data location it next wants to access, is in physical memory.

If not, the operating system must load that required memory from the *swapping (or paging) space* before execution can continue.

However, the swapping space is generally a slow device (a disk), so the *blocking* I/O request forces that process to be `Suspended` until the "piece" of memory is available.

In the interim, another process may be able to execute.

Before we consider how we can achieve this, and introduce additional efficiency, consider what advantages are now introduced:

- More (pieces of) processes may be maintained in main physical memory (either `Ready` or `Running`).

  Not all processes require *all* of their memory before they can execute – memory may be loaded *on demand*.

- If the swapping space is larger than the physical memory, any single process may now demand more memory than amount of physical memory installed.

This last aspect gives the technique its name – *virtual memory*.

"Memory is like an orgasm. It's a lot better if you don't have to fake it."
Seymore Cray (inventor of the first supercomputer), on virtual memory

Virtual Memory and Resident Working Sets

The principle of program locality again tells us that at any time, only a small subset of a process's instructions and data will be required.

We define a process's set of pages, or segments, in physical memory, as its *resident (or working) memory set*.

```
prompt> ps aux
USER     PID %CPU %MEM   VSZ   RSS TTY    STAT START   TIME COMMAND
root       1  0.0  0.1  1372   432 ?      S    Aug11   0:04 init
root       4  0.0  0.0     0     0 ?      SW   Aug11   0:04 [kswapd]
root     692  0.0  0.2  1576   604 ?      S    Aug11   0:00 crond
xfs      742  0.0  0.8  5212  2228 ?      S    Aug11   0:23 xfs -droppr
root     749  0.0  0.1  1344   340 tty1   S    Aug11   0:00 /sbin/minge
root     757  0.2  6.0 23140 15488 ?      S    Aug11  43:10 /usr/bin/X1
 .................
chris   3865  0.0  0.6  2924  1644 pts/1 S    Aug15   0:01 -zsh
chris  25366  0.0  6.0 23816 15428 ?     S    14:34   0:06 /usr/lib/ne
chris  25388  0.0  1.4 17216  3660 ?     S    14:34   0:00 (dns helper
chris  26233  0.0  0.2  2604   688 pts/1 R    19:11   0:00 ps aux
```

In the *steady state*, the memory will be fully occupied by the working sets of all `Ready` and `Running` processes, but:

- If process's working sets are permitted to be too large, only a few processes can ever be `Ready` or `Running`.

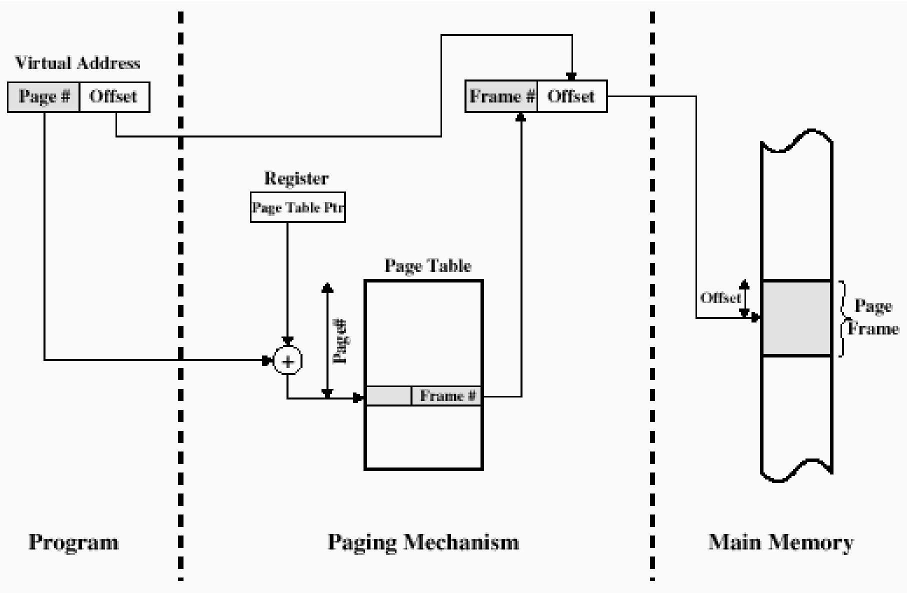
- If process's working sets are forced to be too small, then additional requests must be made of the swapping space to retrieve required pages or segments.

All modern operating systems employ virtual memory based on paging (Q: what is the LINUX page size?). Windows-NT successfully employs virtual memory based on segmentation.

Virtual Memory Hardware using Page Tables

We saw that with simple paging, each process has its own page table entries. When a process's (complete) set of pages were loaded into memory, the current (hardware) page tables were saved and restored by the operating system.

Using virtual memory, the same approach is taken, but the contents of the page tables becomes more complex, because the total size of *all* page table entries becomes too large (slow) to swap in and out:

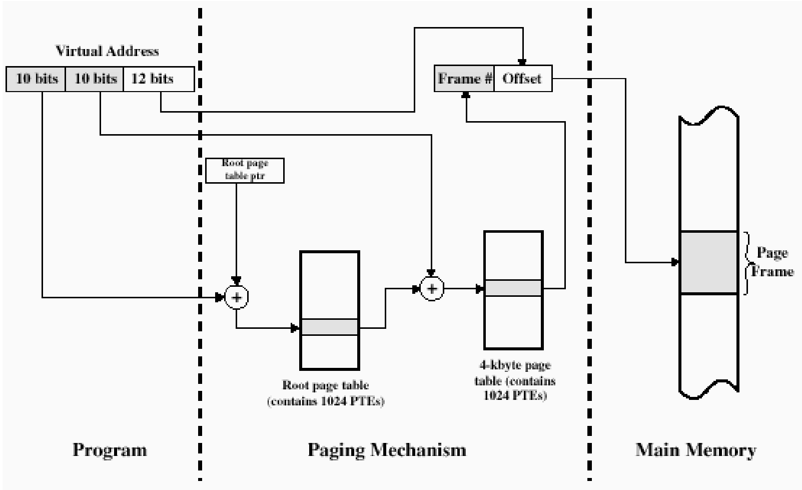


Page table entries must include additional *control information*, indicating at least:

- if the page is *present* in physical memory (a `P` bit), and
- if the page has been *modified* since it was brought into physical memory (a `M` bit).

Virtual Memory Hardware providing Segmentation

When an operating system supports *segmented virtual memory*, similar segmentation tables are required, augmented with the control bits (`P` and `M`).



However, with segmented virtual memory (with unequal, dynamically sized segments), we get some considerable advantages:

- data structures need not be of a fixed maximum size. Data structures may easily grow until they fill their segment, and if they must grow further, they may be swapped-out and swapped back in to a new, larger, segment,
- sections of even *running* programs may be updated and recompiled independently,
- processes may *share* segments, either shared functions in their own segment, or shared data structures, and
- the control information in segment tables may be augmented to provide different protection schemes to a set of procedures or data.

Virtual Memory Page Replacement

When the `Running` process requests a page that is not in memory, a *page fault* results in one of the frames currently in memory being replaced by the required page.

Clearly, the working set of some process must be reduced.

To make room for the required page, an existing page must be "thrown out" (to the swap space).

However, if a page is thrown out just before that page is required (again), it'll just need to be paged back in! If this continues, the activity of *page thrashing* is observed.

We hope that the operating system can avoid thrashing, with an intelligent choice of the page to discard.

Virtual Memory Implementation Considerations

The many different implementations of virtual memory differ in their treatment of some common considerations:

1) When should a process's pages be fetched?
   Clearly a process initially requires the first page containing its starting address (and some initial data structures), but thereafter when should each page be allocated?

   A VM system can employ *demand paging* in which a page is allocated only when a reference to it is made, or *predictive pre-paging* where pages are "intelligently" allocated *before* they are required.

2) Where in physical memory should pages be allocated, using policies such as first-fit, best-fit or next-fit? Does it matter?

VM Implementation Considerations, *continued*

3) Which existing blocks should be replaced – what is the *replacement policy*.

   To avoid *page thrashing*, we wish to replace only pages unlikely to be required soon, but this must be balanced against how many frames can be allocated to a process, and if the `Running` process's pages should be displaced (a local policy) or should *any* frames be displaced (a global policy).

   A number of replacement algorithms exist (seemingly a preoccupation of the '70s) which select pages to be replaced.

   Fundamental algorithms include *first-in, first-out* (obvious, but disadvantages long-running programs with high locality) and *least-recently-used* (almost ideal, but requires time-expensive hardware to maintain time-stamps on page usage).

4) How many processes to admit to the `Ready` and `Running` states?

   The degree of *multiprogramming* permitted must balance processor utilization (minimizing idle states due to I/O blocking) against *utility* (many processes executing with small resident set sizes and possible thrashing).

No-one, yet, claims memory management is easy.