Memory Management

Introduction

The CPU fetches instructions and data of a program from memory; therefore, both the program and its data must reside in the main (RAM and ROM) memory.

Modern multiprogramming systems are capable of storing more than one program, together with the data they access, in the main memory.

A fundamental task of the **memory management** component of an operating system is to ensure safe execution of programs by providing:

- Sharing of memory
- Memory protection

Issues in sharing memory

- *Transparency*

  Several processes may co-exist, unaware of each other, in the main memory and run regardless of the number and location of processes.
- *Safety (or protection)*

  Processes must not corrupt each other (nor the OS!)
- *Efficiency*

  CPU utilization must be preserved and memory must be fairly allocated.
- *Relocation*

  Ability of a program to run in different memory locations.

Storage allocation

Information stored in main memory can be classified in a variety of ways:

- Program (*code*) and data (*variables, constants*)
- Read-only (*code, constants*) and read-write (*variables*)
- Address (*e.g., pointers*) or data (*other variables*); binding (when memory is allocated for the object): static or dynamic

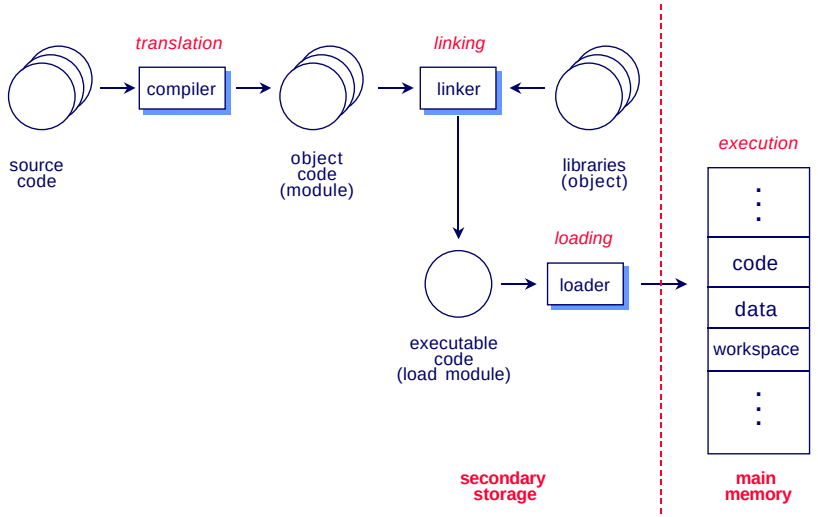
The compiler, linker, loader and run-time libraries all cooperate to manage this information.

Creating an executable code

Before a program can be executed by the CPU, it must go through several steps:

- **Compiling (translating)**—generates the object code.
- **Linking**—combines the object code into a single self-sufficient *executable code*.
- **Loading**—copies the executable code into memory.
- **Execution**—dynamic memory allocation.

From *source* to *executable* code



Address binding (relocation)

The process of associating program instructions and data (addresses) to physical memory addresses is called *address binding*, or *relocation*.

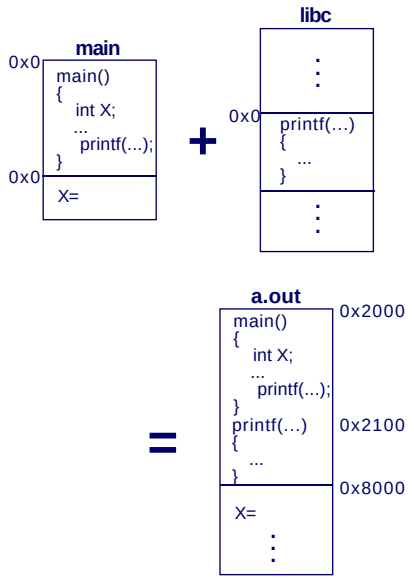
- **Static**—new locations are determined *before* execution.
  - Compile time: The compiler or assembler translates *symbolic* addresses (e.g., variables) to *absolute* addresses.
  - Load time: The compiler translates symbolic addresses to *relative* (*relocatable*) addresses. The loader translates these to absolute addresses.
- **Dynamic**—new locations are determined *during* execution.
  - Run time: The program retains its relative addresses. The absolute addresses are generated by hardware.

Functions of a linker

A *linker* collects (if possible) and puts together all the required pieces to form the executable code.

Issues:

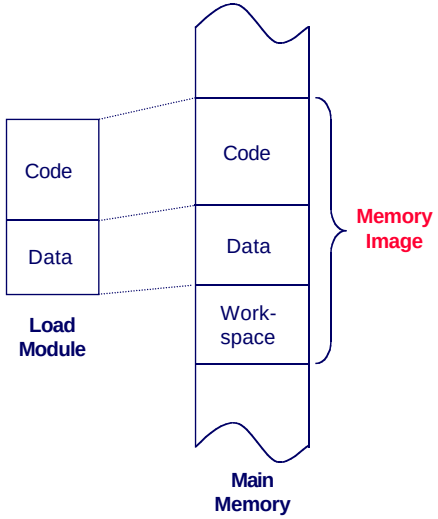
- **Relocation**
  where to *put* pieces.
- **Cross-reference**
  where to *find* pieces.
- **Re-organization**
  new memory layout.



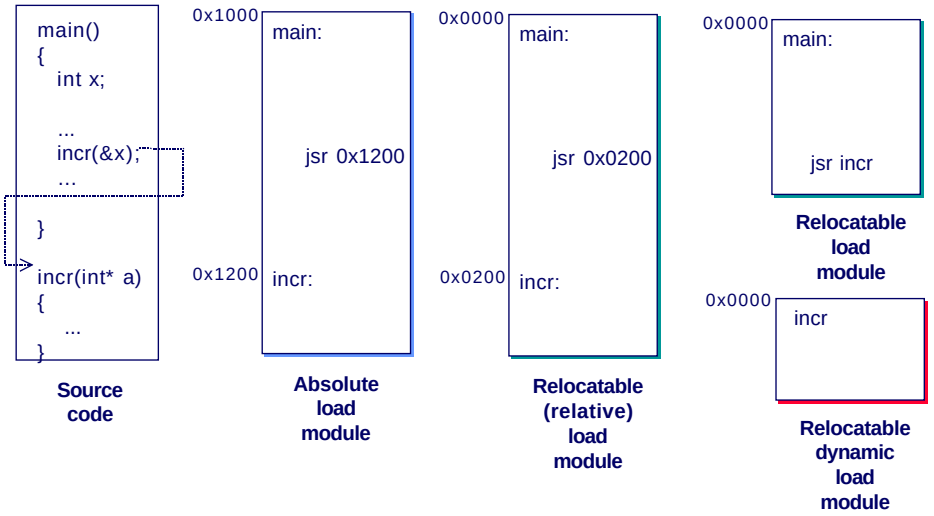
Functions of a loader

A *loader* places the executable code in main memory starting at a pre-determined location (base or start address). This can be done in several ways, depending on hardware architecture:

- *Absolute loading:* always loads programs into a designated memory location.
- *Relocatable loading:* allows loading programs in different memory locations.
- *Dynamic (run-time) loading:* loads functions when first called (if ever).



Absolute and relocatable modules



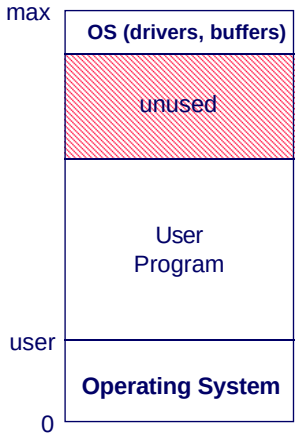
Simple management schemes

An important task of a memory management system is to bring (load) programs into main memory for execution. The following *contiguous memory allocation* techniques were commonly employed by earlier operating systems*:

- Direct placement
- Overlays
- Partitioning

*Note: Techniques similar to those listed above are still used by some modern, dedicated special-purpose operating systems and real-time systems.

Direct placement

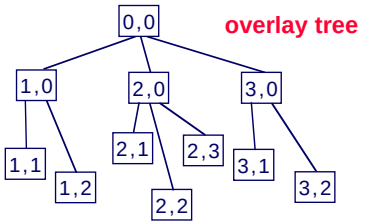


Memory allocation is trivial. No special relocation is needed, because the user programs are always loaded (one at a time) into the same memory location (*absolute loading*). The linker produces the same loading address for every user program.

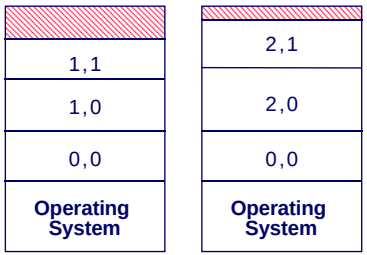
Examples:

- Early batch monitors
- MS-DOS

Overlays



overlay tree



memory snapshots

Historically, before the sharing of main memory among several programs was automated, users developed techniques to allow large programs to execute (fit) in smaller memory.

A program was organized (by the user) into a tree-like structure of object modules, called *overlays*.

The *root* overlay was always loaded into the memory, whereas the sub-trees were (re-)loaded as needed by simply overlaying existing code.

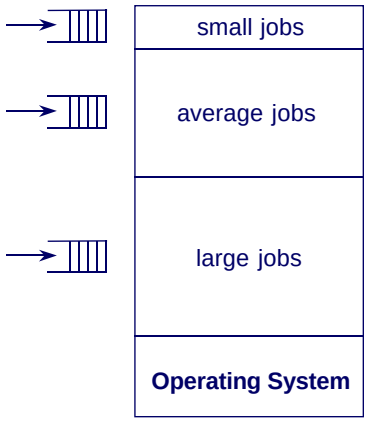
Partitioning

A simple method to accommodate several programs in memory at the same time (to support multiprogramming) is partitioning. In this scheme, the memory is divided into a number of contiguous regions, called *partitions*. Two forms of memory partitioning, depending on when and how partitions are created (and modified), are possible:

- Static partitioning
- Dynamic partitioning

These techniques were used by the IBM OS/360 operating system—*MFT* (*M*ultiprogramming with *F*ixed Number of *T*asks) and *MVT* (*M*ultiprogramming with *V*ariable Number of *T*asks.)

Static partitioning

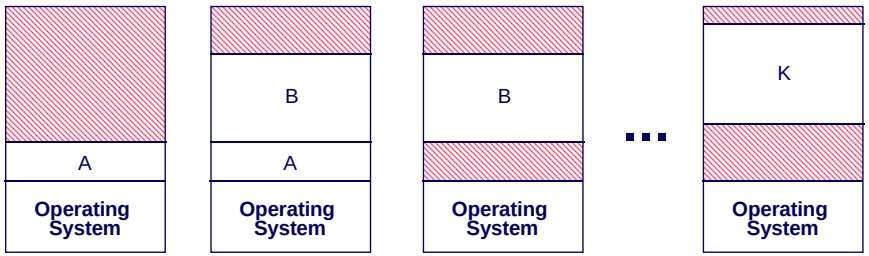


Main memory is divided into *fixed* number of (fixed size) partitions during system generation or startup.

Programs are queued to run in the smallest available partition. An executable prepared to run in one partition may not be able to run in another, without being re-linked. This technique uses *absolute loading*.

Dynamic partitioning

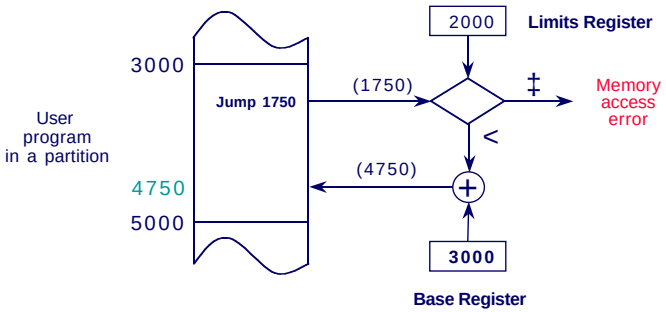
Any number of programs can be loaded to memory as long as there is room for each. When a program is loaded (*relocatable loading*), it is allocated memory in exact amount as it needs. Also, the addresses in the program are fixed after loaded, so it cannot move. The operating system keeps track of each partition (their size and locations in the memory.)



Partition allocation at different times

Address translation

In order to provide basic protection among programs sharing the memory, both of the above partitioning techniques use a hardware capability known as *memory address mapping*, or address translation. In its simplest form, this mapping works as follows:



Fragmentation

Fragmentation refers to the unused memory that the management system cannot allocate.

- *Internal fragmentation*

  Waste of memory *within* a partition, caused by the difference between the size of a partition and the process loaded.
  Severe in static (fixed) partitioning schemes.

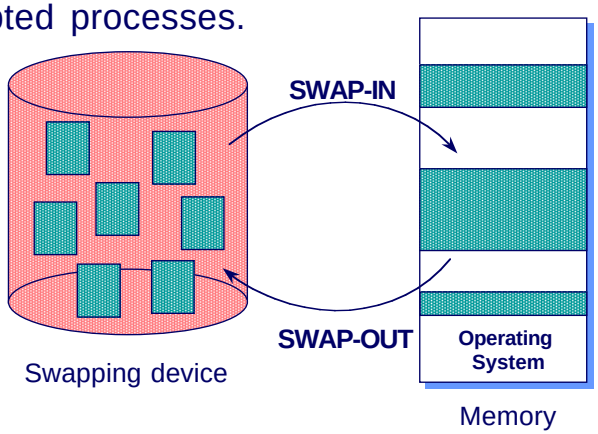
- *External fragmentation*

  Waste of memory *between* partitions, caused by scattered noncontiguous free space. Severe in dynamic (variable size) partitioning schemes.

Compaction is a technique that is used to overcome external fragmentation.

Swapping

The basic idea of swapping is to treat main memory as a ''*pre-emptable*'' resource. A high-speed swapping device is used as the backing storage of the preempted processes.



Swapping continued

Swapping is a medium-term scheduling method.

Swapping brings flexibility even to systems with fixed partitions, because:

> *"if needed, the operating system can always make room for high-priority jobs, no matter what!''*

Note that, although the mechanics of swapping are fairly simple in principle, its implementation requires specific support (e.g., special file system and dynamic relocation) from the OS that uses it.

More on swapping

The responsibilities of a swapper include:

- Selection of processes to swap out
  criteria: suspended/blocked state, low priority, time spent in memory
- Selection of processes to swap in
  criteria: time spent on swapping device, priority
- Allocation and management of swap space on a swapping device. Swap space can be:
  - system wide
  - dedicated

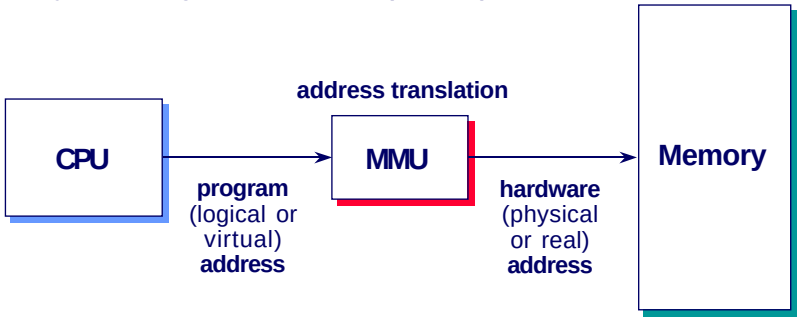
Memory protection

The second fundamental task of a memory management system is to *protect* programs *sharing* the memory from each other. This protection also covers the operating system itself. Memory protection can be provided at either of the two levels:

- Hardware:
  – *address translation*
- Software:
  – language dependent: *strong typing*
  – language independent: *software fault isolation*

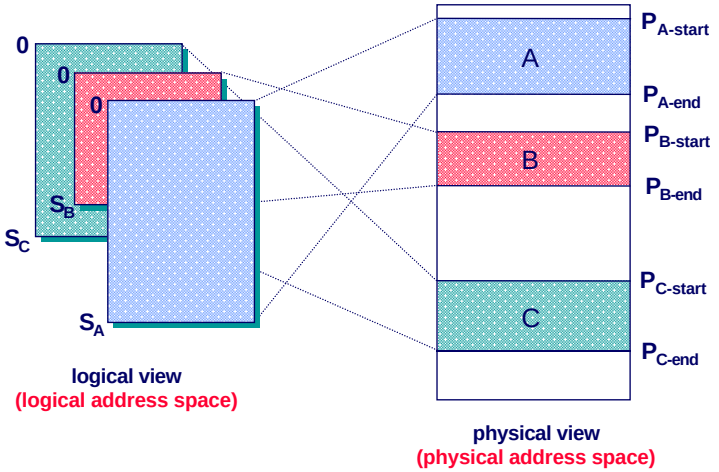
Dynamic relocation

With *dynamic relocation*, each program-generated address (*logical address*) is translated to hardware address (*physical address*) at runtime for *every* reference, by a hardware device known as the memory management unit (MMU).

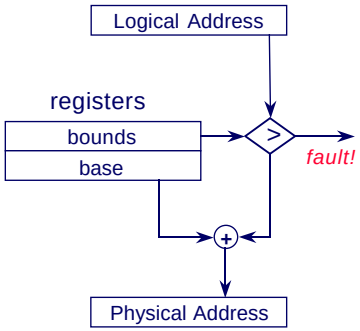


Two views of memory

Dynamic relocation leads to two different views of main memory, called *address spaces*.



Base and bounds relocation



In principle, this is the same technique used earlier by IBM 360 mainframes. Each program is loaded into a contiguous region of memory. This region appears to be ‘‘private’’ and the bounds register limits the range of the logical address of each program.

Hardware implementation is cheap and efficient: 2 registers plus an adder and a comparator.

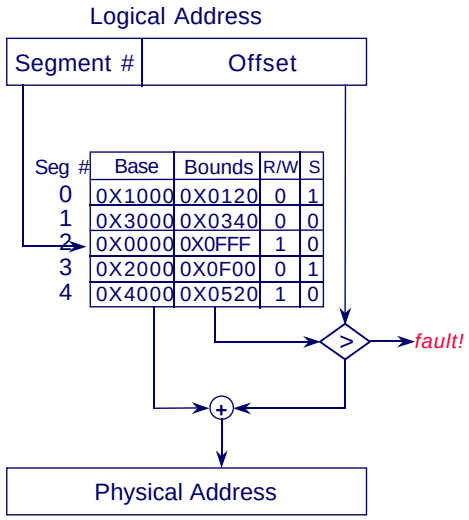
Segmentation

The most important problem with *base-and-bounds* relocation is that there is only one segment for each process. A segment is a region of contiguous memory.

Segmentation generalizes the base-and-bounds technique by allowing each process to be split over several segments. A segment table holds the base and bounds of each segment. Although the segments may be scattered in memory, each segment is mapped to a contiguous region.

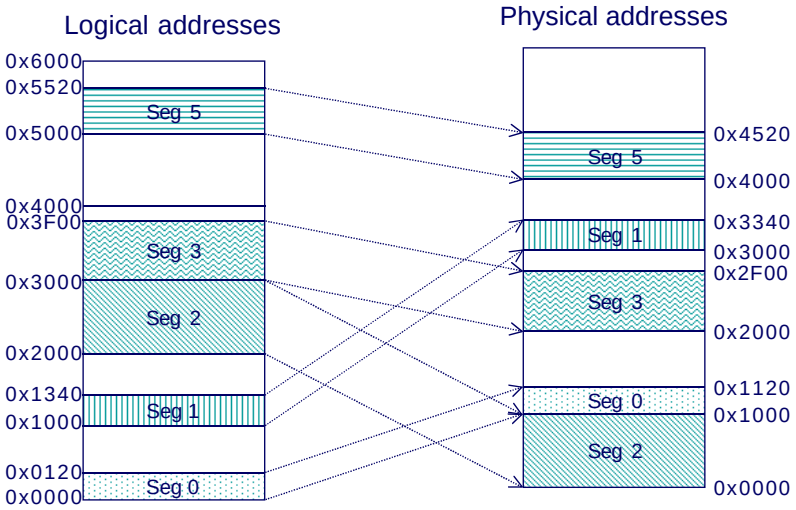
Additional fields (Read/Write and Shared) in the segment table adds protection and sharing capabilities to segments.

Relocation with segmentation



See next slide for memory allocation example.

Segmentation—*an example*



More on segments

When a process is created, a pointer to an empty segment table is inserted into the process control block. Table entries are filled as new segments are allocated for the process.

The segments are returned to the free segment pool when the process terminates.

Segmentation, as well as the base and bounds approach, causes external fragmentation and requires memory compaction.

An advantage of the approach is that only a segment, instead of a whole process, may be swapped to make room for the (new) process.

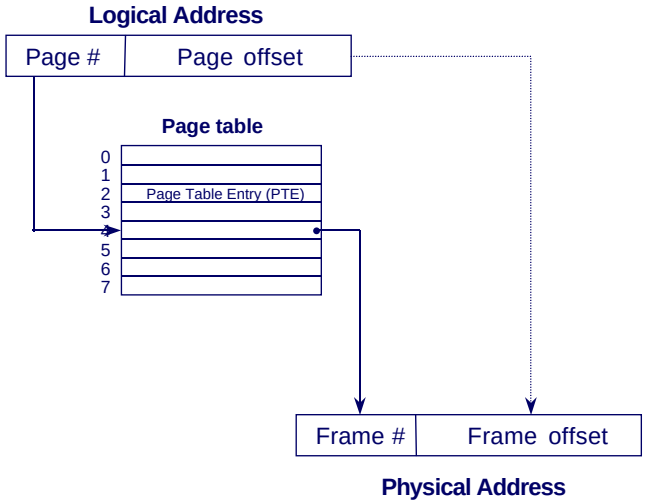
Paging

Physical memory is divided into a number of fixed size blocks, called *frames*. The logical memory is also divided into chunks of the same size, called *pages*. The size of frame/page is determined by the hardware and typically is some value between 512 bytes (VAX) and 16 megabytes (MIPS 10000)!

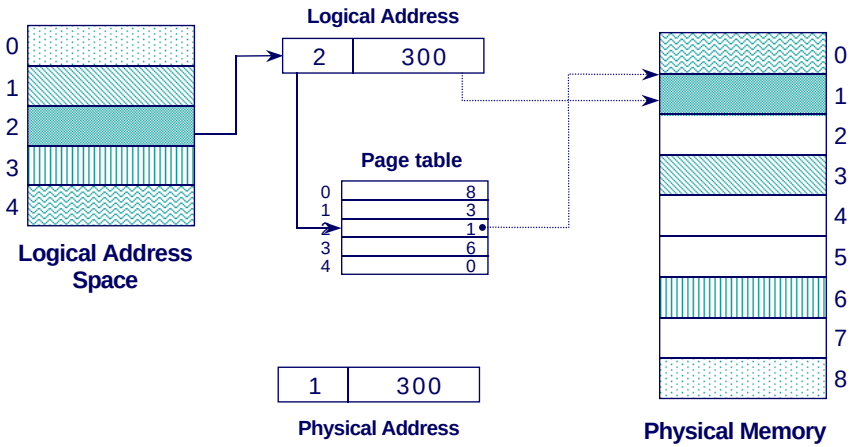
A *page table* defines (maps) the base address of pages for each frame in the main memory.

The major goals of paging are to make memory allocation and swapping easier and to reduce fragmentation. Paging also allows allocation of non-contiguous memory (i.e., pages need not be adjacent.)

Relocation with paging



Paging—*an example*



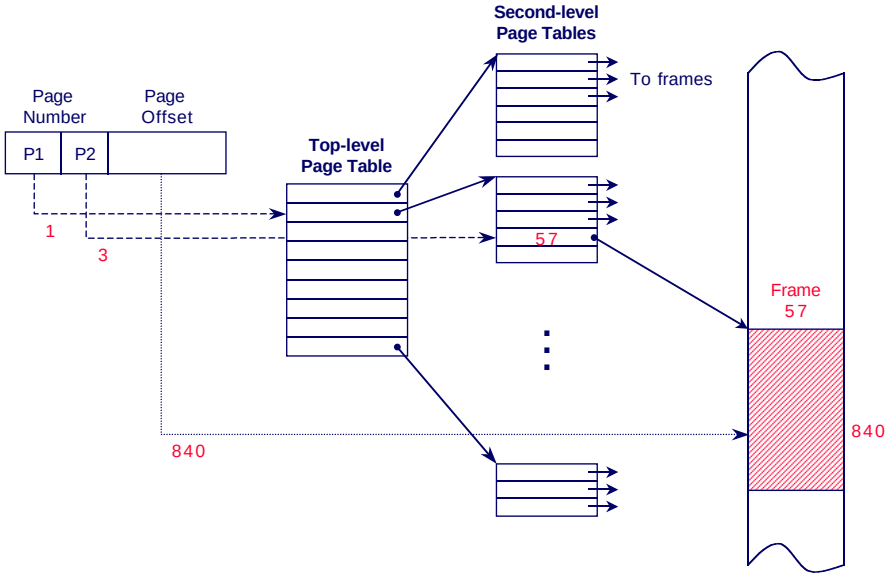
Hardware support for paging

There are different hardware implementations of page tables to support paging.

- A set of dedicated registers, holding base addresses of frames.
- In memory page table with a *page table base register* (PTBR).
- Same as above with multi-level page tables.

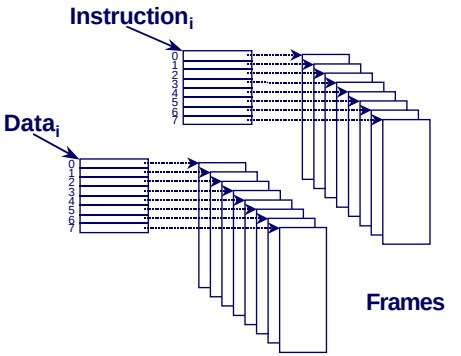
With the latter two approaches, there is a constant overhead of accessing a memory location (*What? Why?*)

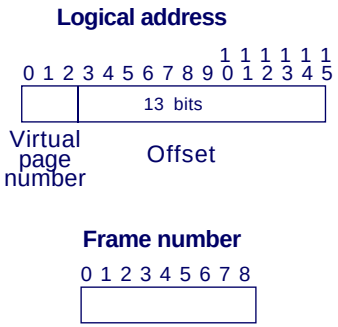
Multi-level paging—*an example*



One level paging—the PDP-11

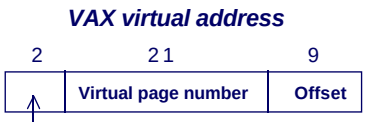
The larger PDP-11 models have 16-bit logical addresses and up to 4MB of memory with page size of 8KB. There are two separate logical address spaces; one for instructions and one for data. The two page tables have eight entries, each controlling one of the eight frames per process.





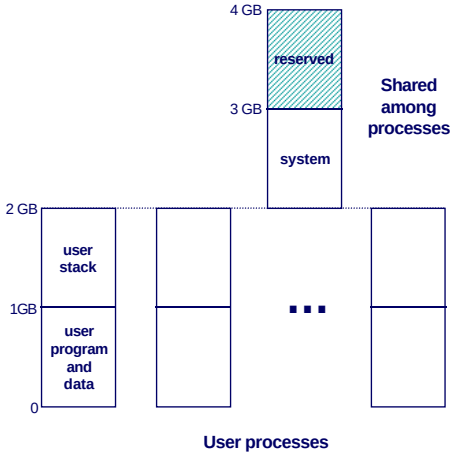
Two level paging—the VAX

The VAX is the successor of the PDP-11, with 32-bit logical addresses. The VAX has 512 byte pages.



Space: 00 — User program and data
01 — User stack
10 — System
11 — Reserved

$2^9 = 512$
$2^{13} = 8192$
$2^{21} = 2097152$



Other MMU architectures

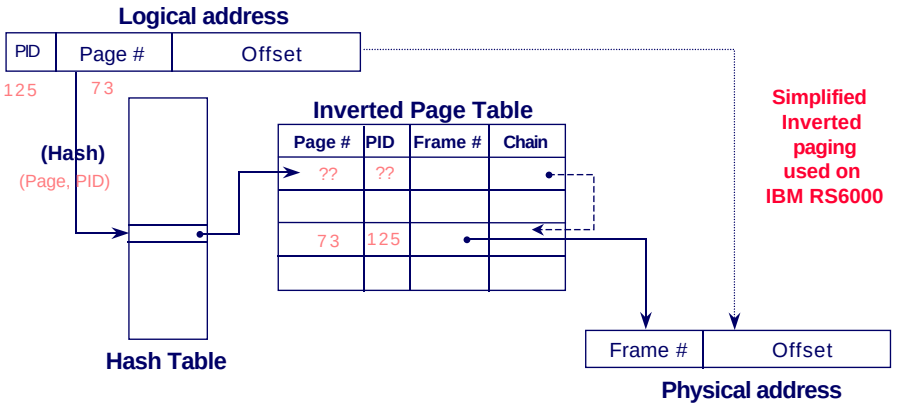
Some SPARC processors used by the Sun workstations have a paging MMU with three-level page tables and 4KB pages.

Motorola 68030 processor uses on-chip MMU with programmable multi-level (1 to 5) page tables and 256 byte to 32KB pages.

PowerPC processors support complex address translation mechanisms and, based on the implementation, provide 2^{80} (64-bit) and 2^{52} (32-bit) byte long logical address spaces.

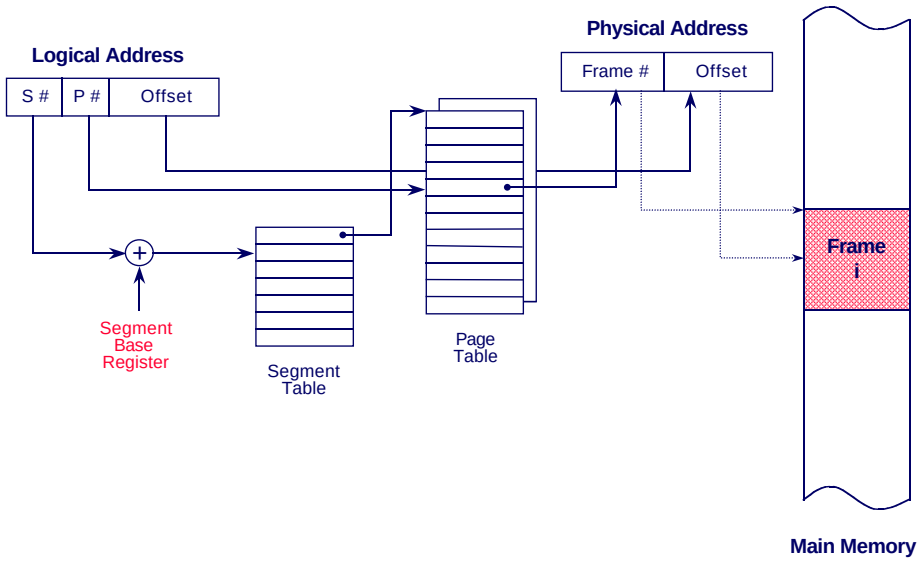
Intel Pentium processors support both segmented and paged memory with 4KB pages.

Inverted page tables

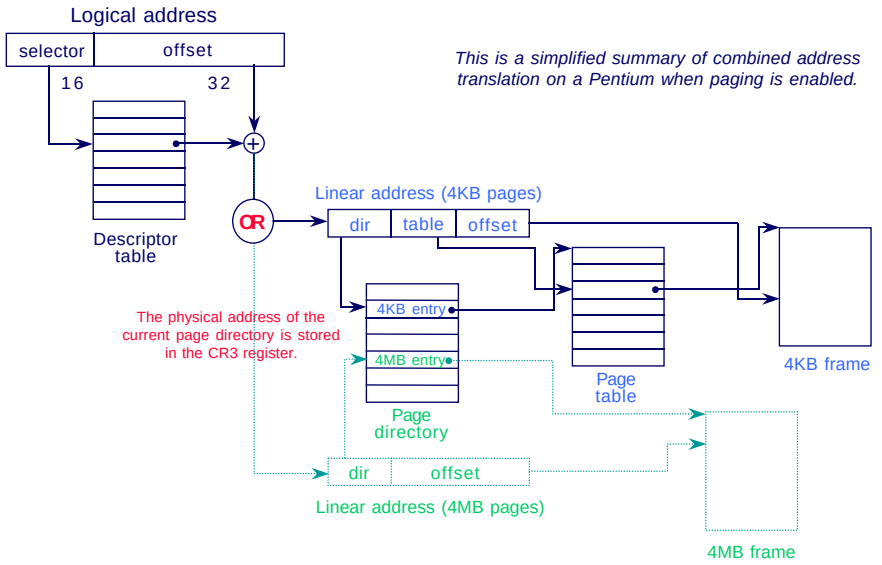


The inverted page table has one entry for each memory frame. **Adv**: independent of size of address space; small table(s).
Hashing is used to speedup table search. Here the inverted page table is system-wide, since the PID is shown. The Inverted Page Table can also be one per process.

Segmentation with paging



Address translation on a Pentium

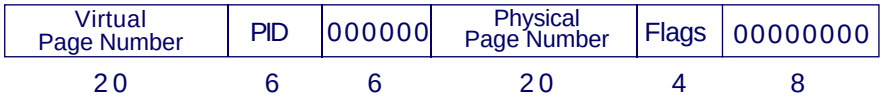


Associative memory

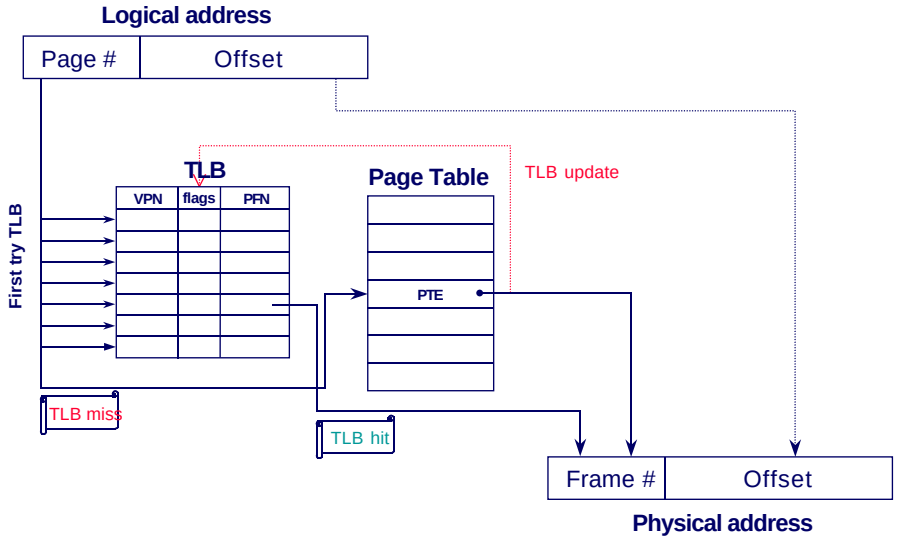
Problem: Both paging and segmentation schemes introduce extra memory references to access translation tables.
Solution? Translation buffers.

Based on the notion of *locality (at a given time a process is only using a few pages or segments)*, a very fast but small *associative* (content addressable) *memory* is used to store a few of the translation table entries. This memory is known as a *translation look-aside buffer* or *TLB.*

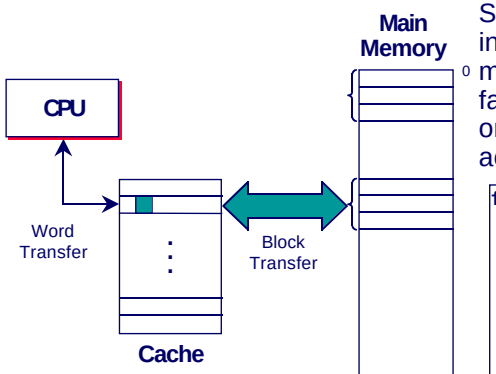
MIPS R2000/R3000 TLB entry (64 bits)



Address translation with TLB



Memory caching



Similar to storing memory addresses in TLBs, frequently used data in main memory can also be stored in fast buffers, called *cache memory*, or simply *cache*. Basically, memory access occurs as follows:

```
for each memory reference
  if data is not in cache <miss>
     if cache is full
        remove some data (make space)
     if read access
        issue memory read
        place data in cache
        return data
  else <hit>
     if read access
        return data
     else
        update data in cache & memory
```

The idea is to make *frequent* memory accesses faster!

Cache terminology

Cache hit: item is in the cache.

Cache miss: item is not in the cache; must do a full operation.

Categories of cache miss:

- *Compulsory*: the first reference will always miss.
- *Capacity*: non-compulsory misses because of limited cache size.

Effective access time:

P(hit) * cost of hit + P(miss)* cost of miss

P(miss) = 1 - P(hit)

Issues in cache design

Although there are many different cache designs, all share a few common design elements:

- **Cache size**—how big is the cache?

  The cache only contains a copy of portions of main memory. The larger the cache the slower it is. Common sizes vary between *4KB* and *4MB*.

- **Mapping function**—how to map main memory blocks into cache lines?

  Common schemes are: *direct, fully associative, and set associative.* (see later)

- **Replacement algorithm**—which line will be evicted if the cache lines are full and a new block of memory is needed.

  A replacement algorithm, such as *LRU, FIFO, LFU, or Random* is needed only for *associative* mapping (Why?)

Issues in cache design continued

- **Write policy**—What if CPU modifies a (cached) location?

  This design issue deals with store operations to cached memory locations. Two basic approaches are: *write through* (modify the original memory location as well as the cached data) *and write back* (update the memory location only when the cache line is evicted.)

- **Block (or line) size**—how many words can each line hold?

  Studies have shown that a cache line width of 4 to8 addressable units (bytes or words) provide close to optimal number of hits.

- **Number of caches**—how many levels? Unified or split cache for data and instructions?

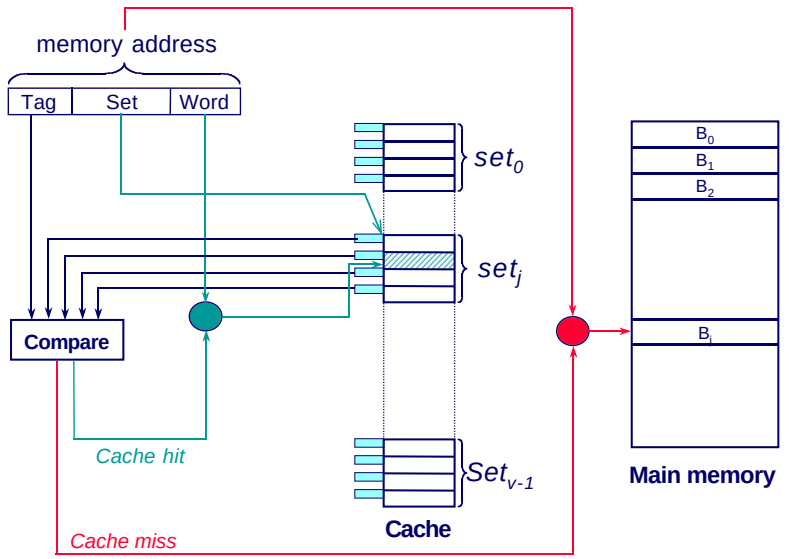
  Studies have shown that a second level cache improves performance. Pentium and Power PC processors each have on-chip level-1 (L1) split caches. Pentium Pro processors have on-chip level-2 (L2) cache, as well.

Mapping function

Since there are more main memory blocks ($Block_i$ *for i=0 to n*) than cache lines ($Line_j$ for *j=0 to m, and n >> m*), an algorithm is needed for mapping main memory blocks to cache lines.

- *Direct mapping*—maps each block of memory into only one possible cache line. $Block_i$ maps to $Line_j$, where *i = j modulo m.*
- *Associative mapping*—maps any memory block to any line of the cache.
- *Set associative mapping*—cache lines are grouped into sets and a memory block can be mapped to any line of a cache set. $Block_i$ maps to Set_j where *i=j modulo v and v is the number of sets with k lines each.*

Set associative cache organization



Dynamic memory allocation

Static memory allocation schemes are not sufficient at run-time because of the unpredictable nature of executing programs. For certain programs, such as recursive functions or programs that use complex data structures, the memory needs cannot be known in advance.

Two basic operations are needed: *allocate* and *free*.

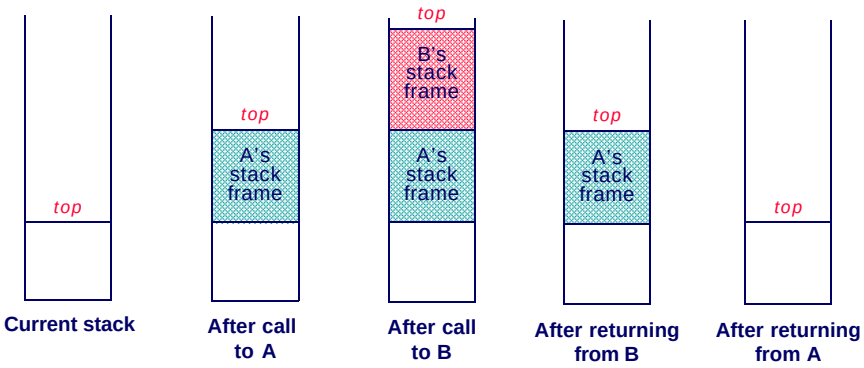
For example, in UNIX, these are **`malloc()`** and **`free()`**.

`new []` **`delete []`**

Dynamic allocation can be handled using either *stack* (hierarchical, restrictive) or *heap* (more general, but less efficient) allocation.

Stack organization

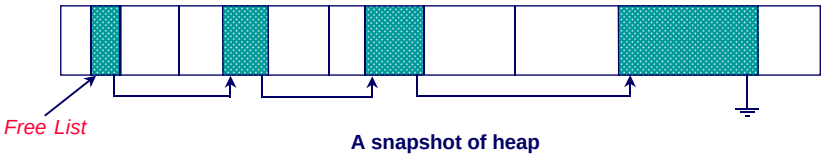
Memory allocation and freeing operations are partially predictable. Since the organization is hierarchical, the freeing operates in reverse (opposite) order.



Heap organization

Allocation and release of heap space is totally random. Heaps are used for allocation of arbitrary list structures and complex data organizations. As programs execute (and allocate and free structures), heap space will fill with holes (unallocated space.)

Analysis of memory allocation strategies indicates that, when a system reaches a steady state condition, there will be half as many holes as in-use segments in the system. This result is known as the *fifty percent rule*.



A snapshot of heap

"Free" memory management

- **Bit maps**

  This technique divides memory into fixed-size blocks (e.g., sectors of 256-byte blocks) and keeps an array of bits (bit map), one bit for each block.

- **Linked lists**

  A *free list* keeps track of the unused memory. There are several algorithms that can be used, depending on the way the unused memory blocks are allocated: *first fit, best fit, next fit, and worst fit*.

- **Buddy system**

  This algorithm takes advantage of binary systems. As a process requests memory, it is given the smallest block (with a size of power two) where it can fit.

Reclaiming memory

How do we know *when* memory can be freed?

It is trivial when a memory block is used by one process. However, this task becomes difficult when a block is shared (e.g., accessed through pointers) by several processes.

Two problems with reclaiming memory:

- **Dangling pointers**: occur when the original allocator *frees* a shared pointer.
- **Memory leaks**: occur when we forget to free storage, even when it will not or cannot be used again.

Reference counts

Memory reclamation can be done by keeping track of reference counters (i.e., the outstanding pointers to each block of memory.) When the counter goes down to zero, the memory block is freed. This scheme works fine with hierarchical structures, but becomes tricky with circular structures.

Examples:

- Smalltalk uses a similar scheme.
- UNIX file descriptors. After a system crash, **`fsck`** program runs (during rebooting) to check the integrity of file systems.

Garbage collection

As an alternative to an explicit free operation, some systems implicitly free storage by simply deleting pointers. These systems search through all deleted pointers and reclaim the storage referenced by them.

Some languages, e.g., Lisp and Java, support this kind of “reclaimed” (free) memory management.

Garbage collection is often expensive; it could use more than 20% of the total CPU time! Also, it is difficult to code and debug garbage collectors.