MAIN MEMORY

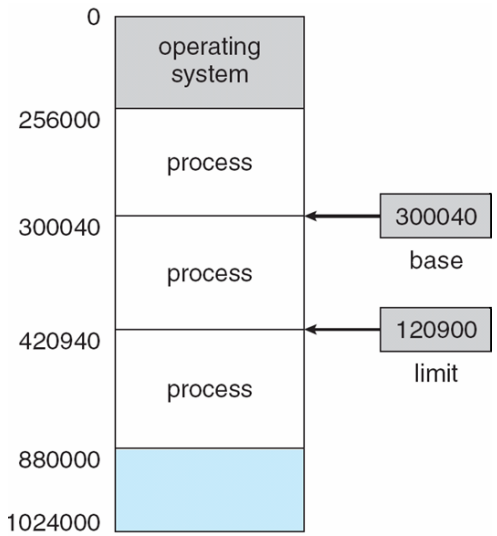
Ruben Acuña

Spring 2018

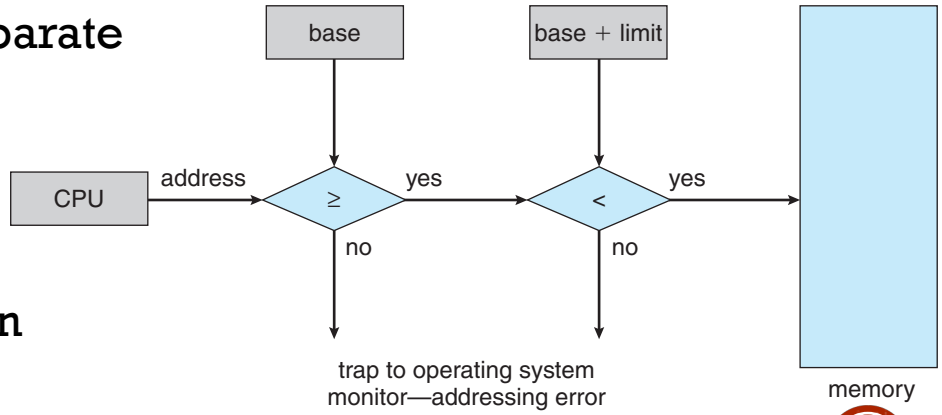
BACKGROUND (1)

HARDWARE OUTLINE

- General paradigm is of a main system memory, which programs use to populate local registers on the CPU.
  - Modern systems generally have some sort of caching mechanism for reducing the overhead of using the memory bus.

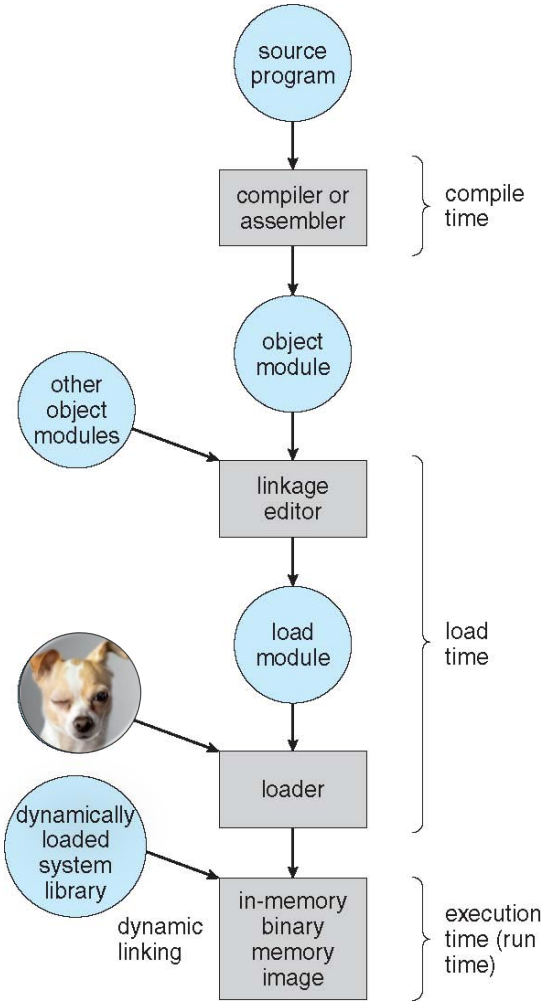


- Each process will be assigned a separate region within main memory.
- For protection, processes have two registers:
  - *base*: the start of the processes memory with respective to main memory.
  - *limit*: the size of space allocated to it.



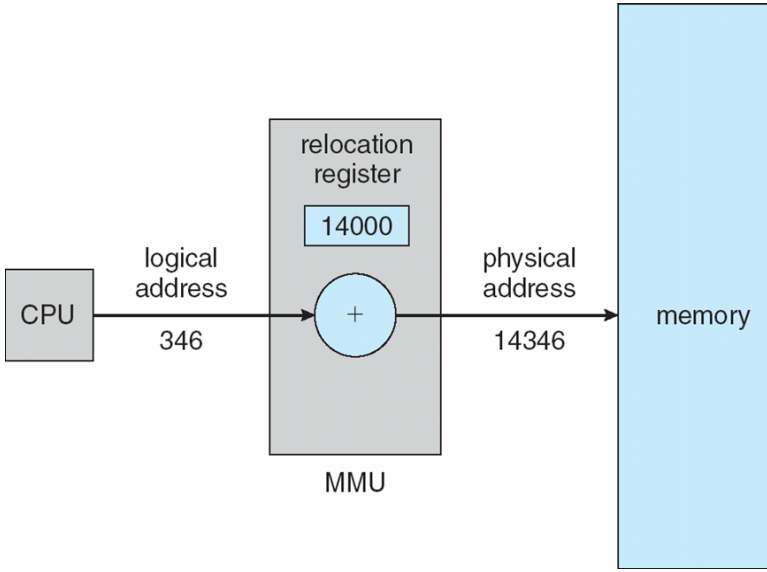
ADDRESS BINDING

- Every piece of data in a program must exist at some physical memory location. (not in secondary storage…)
- In terms of source code, we mainly know about locations symbolically (int x := y).
- At compile time: we can re*bind* these symbolic locations to *relocatable addresses* (start+10 := start+24).
- At binary load time, the OS picks a memory location for the process and generates actual addresses: 1010 :=1024).
  - On older systems (DOS), the location where a binary is loaded is a constant. Modern systems pick it for obvious reasons (which are?).
- An extra concern is that processes may be moved address during *execution time*, in which case address need to be remapped on the fly.



LOGICAL VS PHYSICAL ADDRESSES

- *There are two types of addresses:*
  - *Logical (virtual) Address*: address computed by CPU
    …which the Memory-Management Unit (MMU) maps to…
  - *Physical Address*: seen by memory actual hardware load unit.
- Both of these define a "*space*" in memory of what might be accessed.
- Simple MMU just adds an offset (a base stored in a *relocation register*) to all logical addresses generated by a program.



DYNAMIC LOADING AND LINKING

- When loading a process into memory, there is no requirement we load the entire binary into memory. We can dynamically load it in a piecewise fashion.

- More interesting: when using a library, we have a choice between *static* and *dynamic linking*.
- For dynamic linking, a table of functions is introduced into your program where each entry is a *stub* which is used to locate/load the actual library code.
- During execution, flow of control in a program jumps to the table, where either a function has already been loaded or a stub needs to run.

- What are the advantages to doing this?
- What are the disadvantages to doing this?

SWAPPING (2)

SWAPPING

- A process may be stored in one of two places: main memory or a *backing store*.
- The store may be located on many types of devices, and we say that processes are *swapped* between them.
  - Typically a HDD is used, not so much an SSD…?
- In a typical implementation, processes are only moved to the backing store when overall system memory runs too low.
- By the nature of the store, loading processes into and out of it will be slow.
  - Anyway we can avoid moving the entire process between locations?

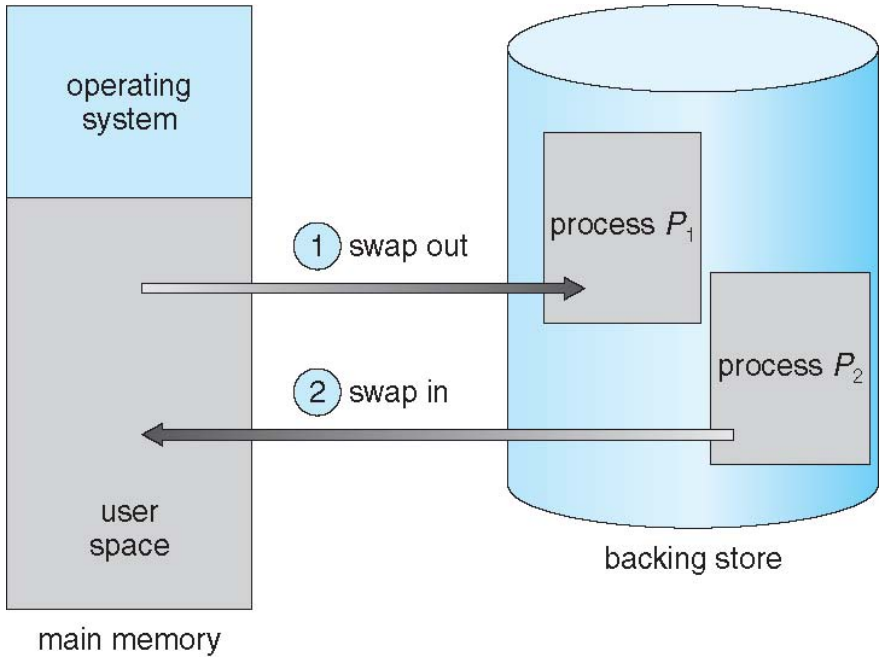


Image from Operating System Concepts.

CONTIGUOUS MEMORY ALLOCATION (3)

MEMORY MODEL

- For now, we will assume a *contiguous* memory allocation model for processes.
  - Stores information about where a process starts and where it ends.

MEMORY PROTECTION

- Memory is divided into two regions: space for system applications versus space for user applications.
  - Why?

- Mechanism: during a context switch the *relocation* register and *limit* register are loaded in kernel mode.
- During program execution, addresses generated by the user program are checked against this safe range.

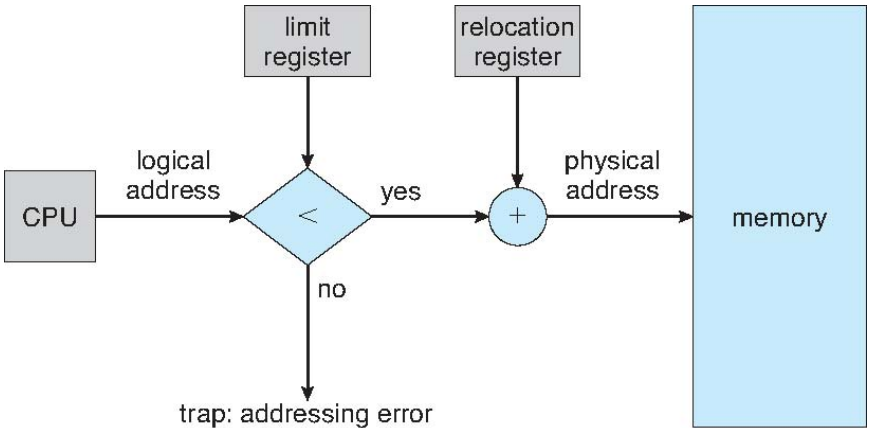


Image from Operating System Concepts.

MEMORY ALLOCATION

- A simple way to handle memory is to *partition* it into constant size chunks that can allocated for processes.
  - A better generalization is to consider *variable-partition* sizes.
- When there is a region of memory that is available, we will call it a *hole*.
- As processes are spawned, holes are allocated, and as they are killed, holes are released. For variable-partition schemes:
  - During allocation, if a hole is larger than a process needs, then it may be split.
  - For deallocation, if a hole is created next to an existing hole, they are merged.

MEMORY ALLOCATION

- Hole selection can be done in three ways:
  - *First Fit*: given a pool (list?) of holes, select the first one that is large enough for a process.
  - *Best Fit*: search over the entire pool of holes, and select the one closest (without being under sized) to the process.
  - *Worse Fit*: search over the entire pool of holes, select the largest one. (The plan here is that the hole can be split.)

MEMORY FRAGMENTATION

- There are two types of fragmentation:
  - *External*: sum of free memory available across empty holes. A particular hole may not be large enough for a process to occupy.
  - *Internal*: sum of free memory available within allocated memory partitions.

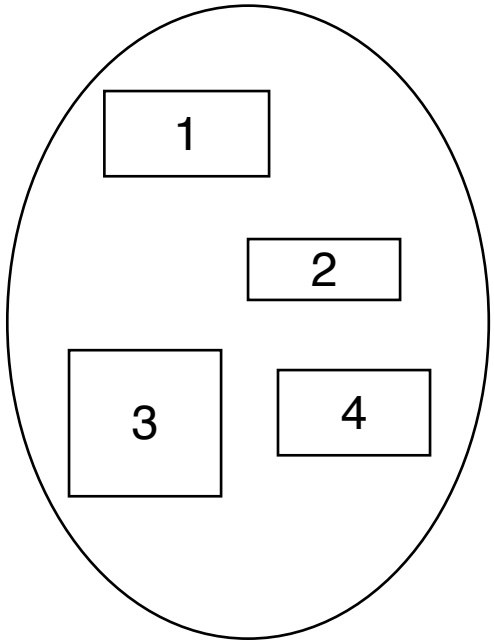
- On average, for a system with N allocations with first fit selection, ~1/3 of memory will be unusable due to external and internal fragmentation.

- Ideally, we can reclaim some of this memory by doing compaction (defragmentation). Must be a little a careful though:
  - Can only do so if memory addresses are dynamically determined.
  - Processes that are waiting on I/O calls should not be moved. Why?

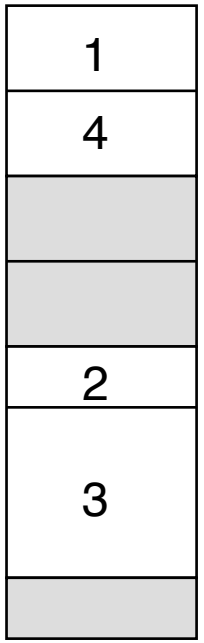
SEGMENTATION (4)

METHOD

- Basic idea, partition a program into different memory segments based on logical units in the code.
- What might logical units be?
- Each logical address will be represented as a tuple of integers: (segment-number, offset).



user space

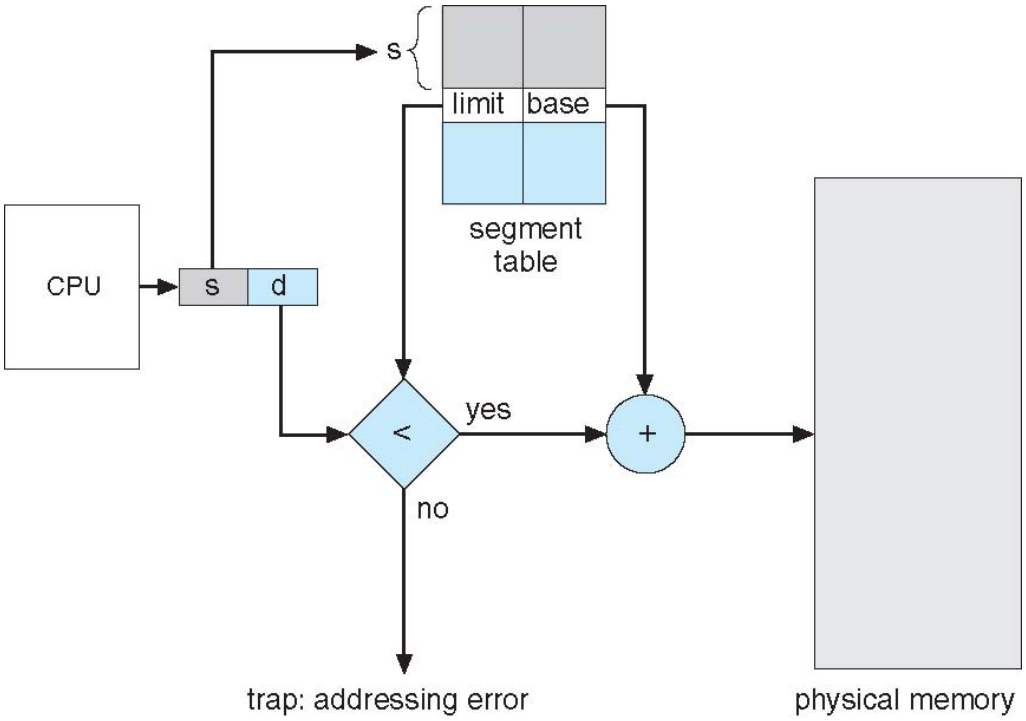


physical memory space

"Image" from Operating System Concepts.

HARDWARE

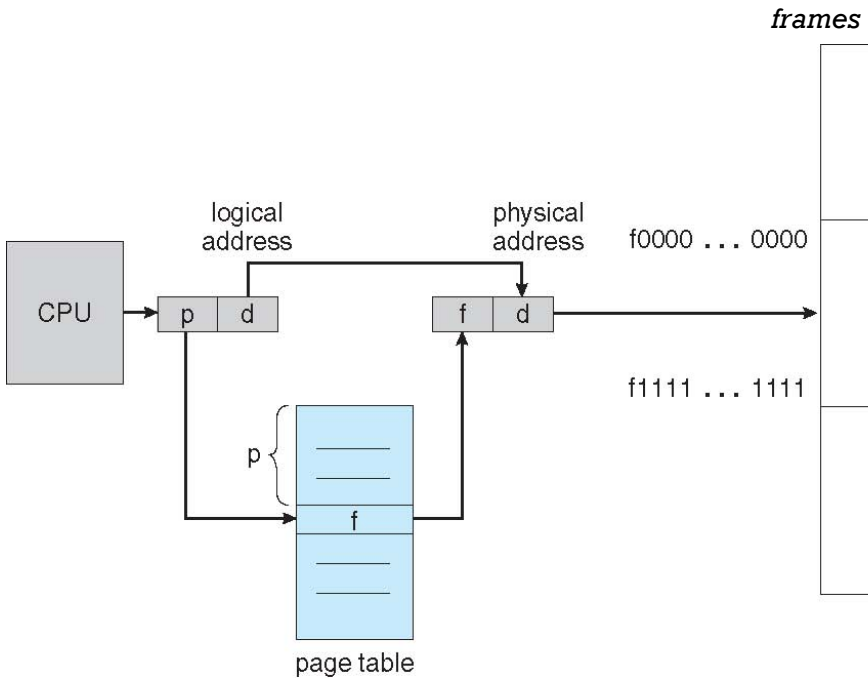
- The mapping of segments to memory is stored in a *Segment Table.*
- For each entry, the table stores a *Segment Base* and *Segment Limit.*



PAGING (5)

THE PAGE CONCEPT

- Physical memory is divided into *frames* of some size. Logical memory is also divided into *pages* of the same size.
- This means we can build a correspondence between pages and *frames* such that there is a level of indirection between what the process sees as global addresses (really: logical addresses) and the real physical address.
- Programs will require space based on a number of pages, so we can check for that many frames and allocate them.
  - Note that we will have to construct a *page table* to show the correspondence (which page maps to which frame and where that frame is stored).
- In this model, logical addresses are composed of two parts: a page number and a page offset. Say that the size of a logical address is *m*, and so page number can be stored in m-n bits, and the page offset in *n* bits.



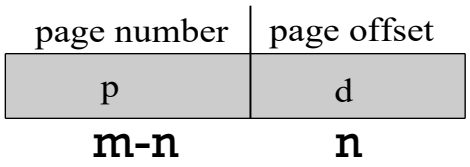


Image from Operating System Concepts.

PAGING METHOD

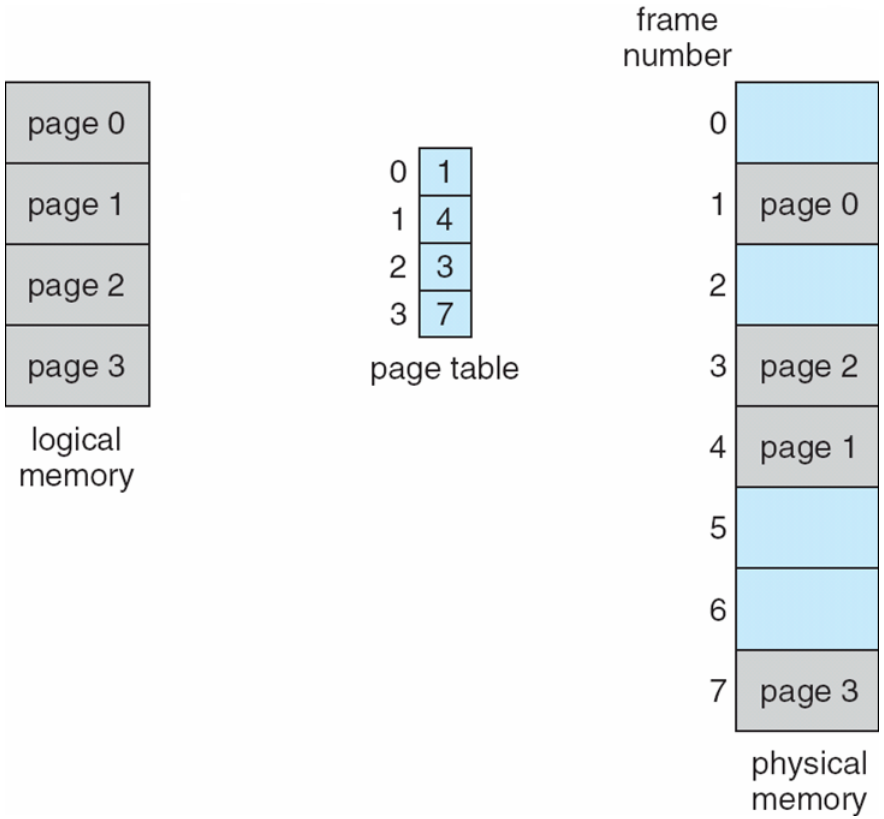


Image from Operating System Concepts.

EXAMPLE

- In each logical address, n=2 and m=4.
- Meaning, it uses 2-bits for offset and uses 2-bits for page #.
- Consider loading logical address 3 (0011) vs address 4 (0100).

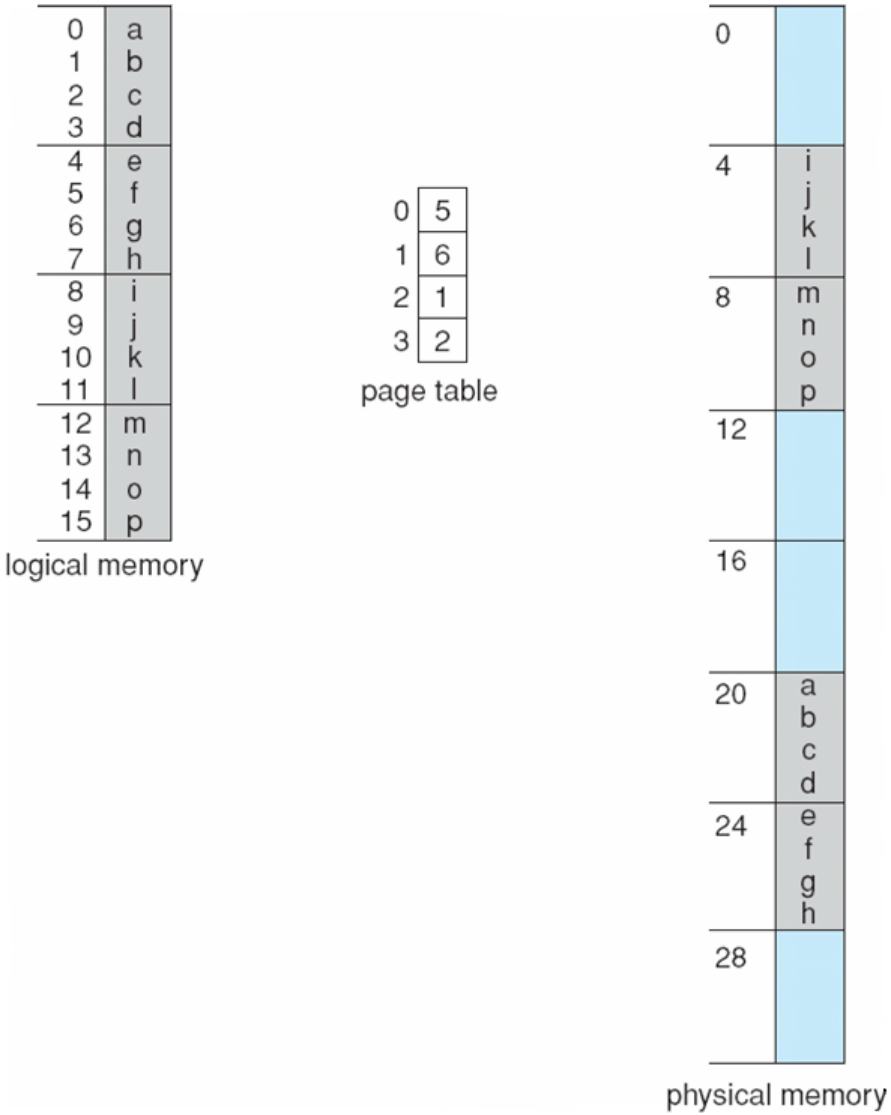


Image from Operating System Concepts.

PAGE MANAGEMENT

- Allocation is straightforward.
- Is fragmentation of external space still an issue?
  - Worse case? Average case?
- Is fragmentation of internal space still an issue?
  - Worse case? Average case?
- In general, modern operating systems use a 4KB paging system.

Image from Operating System Concepts.

HARDWARE SUPPORT

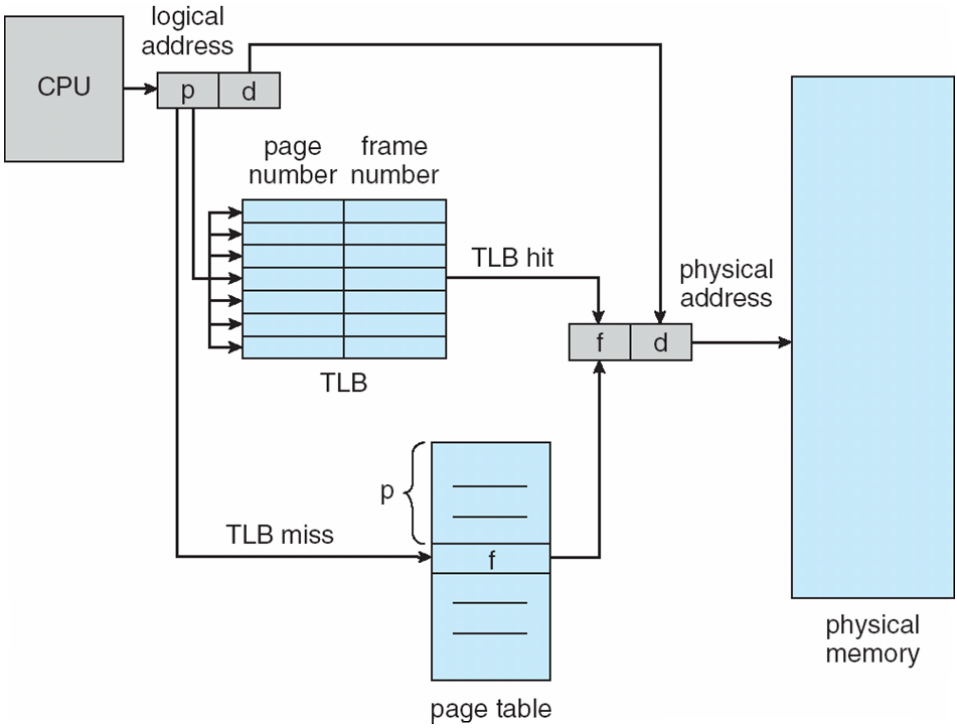
- We need somewhere to store page tables.
- One approach would be to have a set of registers (say 256) which each store a physical address for a page used by the current process.
  - This is not a such a good idea though – any idea why not?
- Another approach would be to store them at some location in system memory, and keep track of a *Page-Table Base Register* (PTBR) that stores an offset where the table begins.
  - Again, this is not a such a good idea though – any idea why not?

- The way this is implemented on most modern processes is via Translation Look-aside Buffer (TLB), implemented with associative memory.
- In *associative memory*, keys are paired with values. In hardware, a lookup is comprised of checking each entry simultaneously in a single tick.

Also called an associative array.

HARDWARE SUPPORT

- A TLB is a CPU local map from a page number to a physical address.
- When the system needs to derive a physical address from a logical address, a TLB lookup is performed to find the page location. If it does not find the page in the TLB, then a *TLB miss* occurs and it must be looked up from main memory.
- As TLB misses occur, the address that was look up is stored into the TLB and replaces the *Least Recently Used* (LRU) entry.
- Some addresses will be *wired down*, meaning they are not subject to LRU replacement.
  - Which addresses will these be?
- Some TLBs include Address-Space Identifiers (ASIDs), which indicate which process that owns that particular page-address pair.

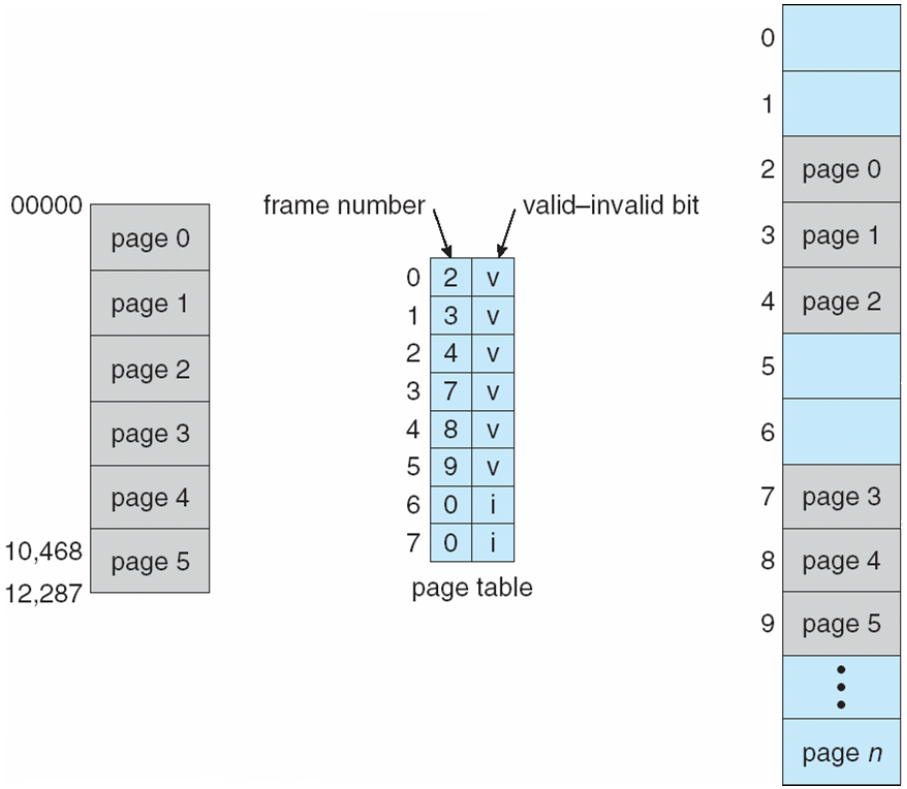


- The number of hits to misses in TLB looks is called the hit ratio. A 90% ratio means 90% of addresses were found in the TLB.
- Based on the hit ratio, and the look up time, we can compute *effective memory-access* time: *(hit ratio)*TLBaccesstime+(1-hitrate)*mainaccesstime*

This is the book's term, it should be "expected" not "effective".

PROTECTION

- Within the TLB, we can a second value along side ASIDs that represent the IO permissions that a process has on that page: read/write/readwrite, executable/data.
- We can also add a *valid-invalid bit* to each page entry.
- This prevents an application from accessing a page that is not assigned to it.
- This gives us a little more flexibility than just storing a ASID – how?



The textbook says that sharing happens for *reentrant code* – however, this is not correct terminology. Reentrant code is code that can be interrupted.

SHARED PAGES

- Consider the following scenario: someone opens three instances of a text editor.
- Do we really need to allocate new pages for the entirety of each program?
  - If we try to make do with fewer pages, are there are requirements on the process?

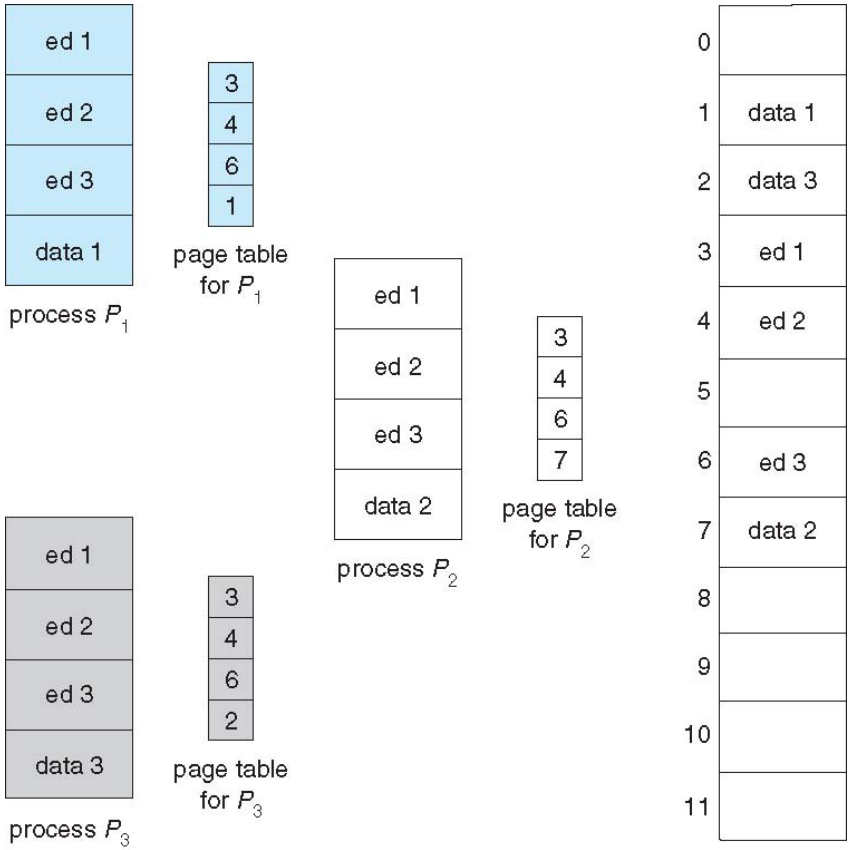


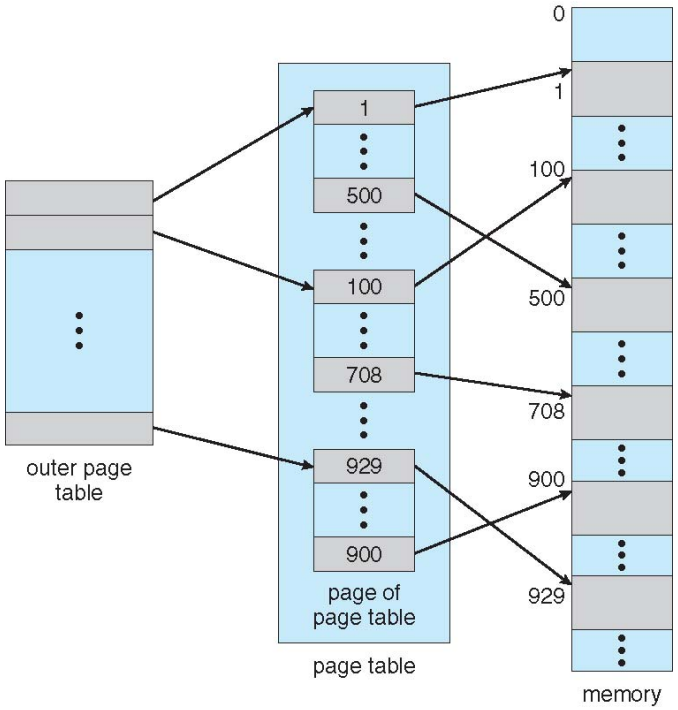
Image from Operating System Concepts.

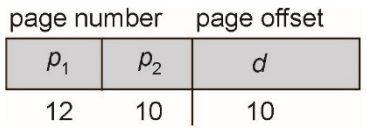
STRUCTURE OF THE PAGE TABLE (6)

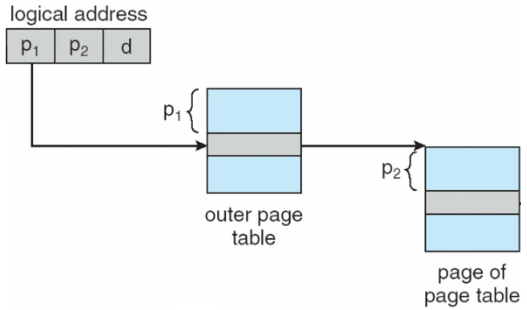
Skip 8.6.4 Oracle SPARC Solaris.

HIERARCHICAL PAGING

- A potential issue with the direct paging method we saw earlier is that the page table may be very large – too large to store locally and/or too large to cache properly.
- Solution: page the page table.
- Logical addresses (in 32-bits) will be structured as a page number in 12+10bits, and a 10-bit offset (*d*).
  - The page number region will itself be divided into 12 and 10-bit regions that each define an index into a page table (*p1* and *p2*).
  - We'll call this particular technique a *forward mapped page table*.



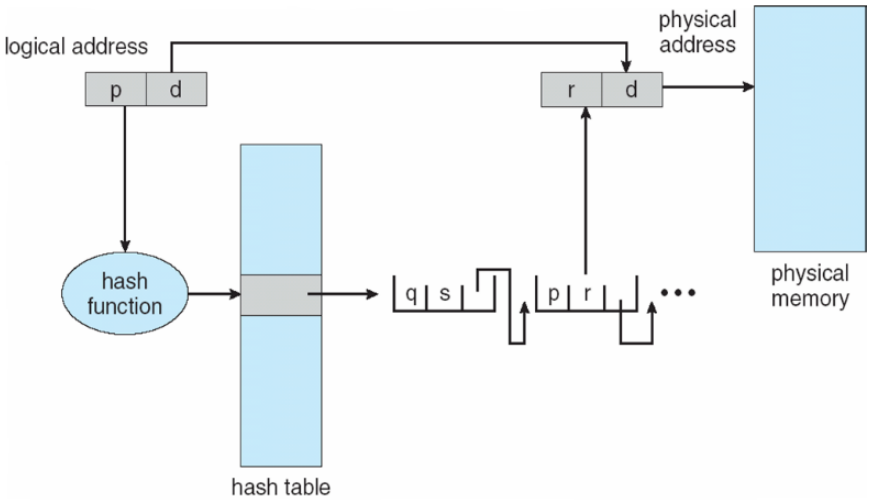




Images from Operating System Concepts.

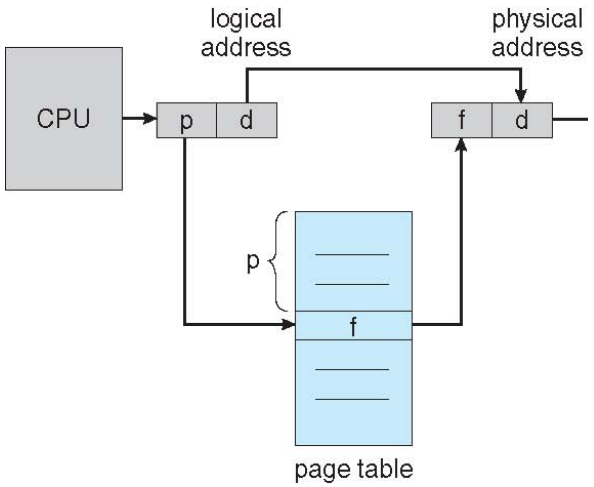
HASHED PAGE TABLES

- Another approach to dealing with a page table that maps a large number of pages to frames is to store the mapping in a hash table.
- Also can address the issue of *sparse* logical address space.
- Typical implementation would be chaining.

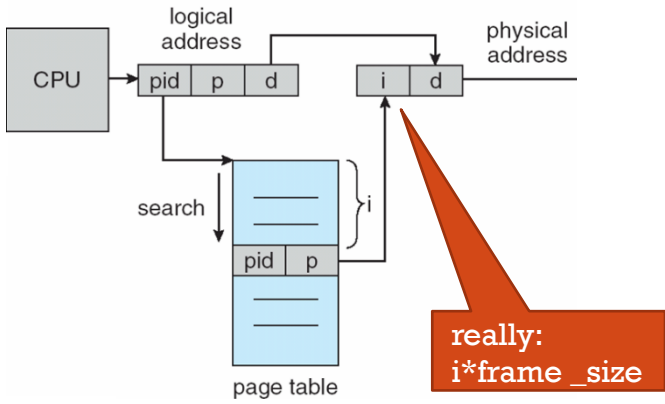


INVERTED PAGE TABLES

- In a normal page table, information about which frame to select is encoded in the page number / table index.
- Recalling our problem of large and/or spare page tables, this method is problematic since pages 0 to n must be in the table, even if they are not all used.
- Thought: what if array indices indicated a frame number?
- Then, we search the table contents for a page number and check the index of where we found it, in order to determine its location in physical memory.
- Any downside?
- Note that we are assuming that physical memory is being used contiguously.
  - Why?
  - And is this okay to assume?



Normal Page Table



Inverted Page Table