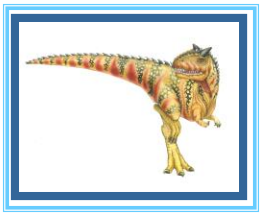
Chapter 8: Main Memory



Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table

Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation

Background

- A typical instruction-execution cycle, for example, first fetches an instruction from memory.
- The instruction is then decoded and may cause operands to be fetched from memory.
- After the instruction has been executed on the operands, results may be stored back in memory.
- The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data).
- Accordingly, we can ignore how a program generates a memory address. We are interested only in the sequence of memory addresses generated by the running program.

Background

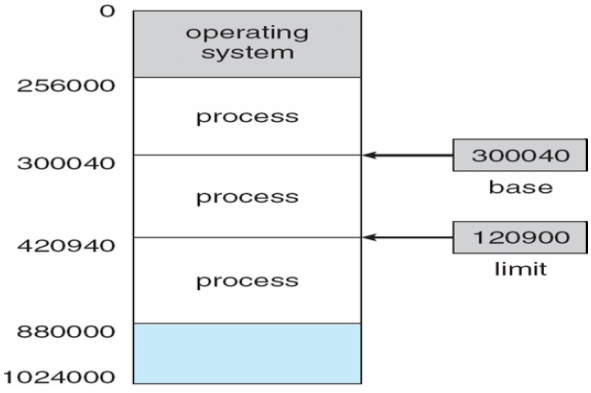
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage that CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles to make available data to the processor, causing a **memory stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation On multiuser systems, we must additionally protect user processes from one another.

Base and Limit Registers

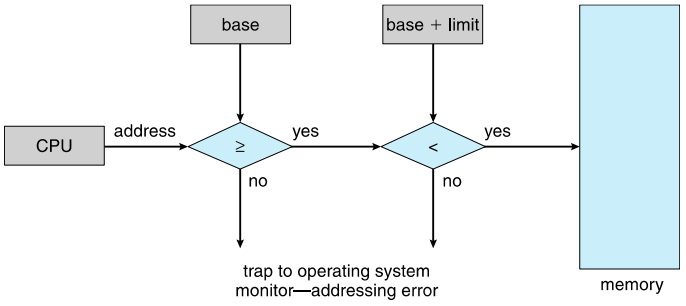
- We first need to make sure that each process has a separate memory space.
- Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution.
- To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.
- We can provide this protection by using two registers, usually a base and a limit.
- The base register holds the smallest legal physical memory address; the limit register specifies the size of the range.

Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



Hardware Address Protection



Hardware Address Protection

- This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.
- The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers.
- This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents

Address Binding

- **Usually**, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process.
- **Depending** on the memory management in use, the process may be moved between disk and memory during its execution.
- **The processes** on the disk that are waiting to be brought into memory for execution form the input queue.
- **The normal** single-tasking procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory.
- **Eventually,** the process terminates, and its memory space is declared available.
- **Most systems** allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer may start at 00000, the first address of the user process need not be 00000. Y

Address Binding

- In most cases, a user program goes through several steps—some of which may be optional—before being executed
- Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as the variable count).
- A compiler typically binds these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module").
- Relocatable code is software whose execution address can be changed. A relocatable program might run at address 0 in one instance, and at 10000 in another.
- The linkage editor or loader in turn binds the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another.

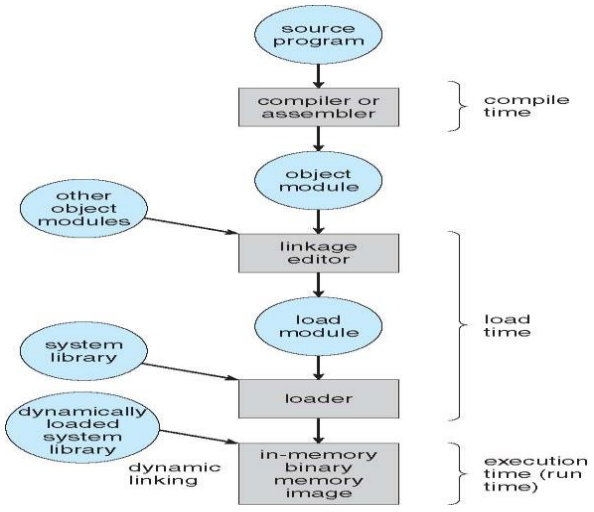
Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time.** If you know at compile time where the process will reside in memory, then absolute code can be generated. For example, if you know that a user process will reside starting at location R, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code.

Binding of Instructions and Data to Memory

- **Load time.** If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value
- **Execution time.** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work, most general-purpose operating systems use this method.

Multistep Processing of a User Program



Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address.** The logical address is virtual address as it does not exist physically. It's typically converted to a physical address later by a hardware chip(MMU) (mostly, not even the CPU is aware really of this conversion).
  - **Physical address** – address seen by the memory unit. The address of where something is physically located in the RAM chip.
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses **Physical address space** is the set of all physical addresses

Logical vs. Physical Address Space

Parameter	LOGICAL ADDRESS	PHYSICAL ADDRESS
Basic	generated by CPU	location in a memory unit
Address Space	Logical Address Space is set of all logical addresses generated by CPU in reference to a program.	Physical Address is set of all physical addresses mapped to the corresponding logical addresses.
Visibility	User can view the logical address of a program.	User can never view physical address of program.
Generation	generated by the CPU	Computed by MMU
Access	The user can use the logical address to access the physical address.	The user can indirectly access physical address but not directly.
Editable	Logical address can be change.	Physical address will not change.
Also called	virtual address.	real address.

Logical vs. Physical Address Space

- **The user program** never sees the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses—all as the number 346. Only when it is used as a memory address (in an indirect load or store, perhaps) it is relocated relative to the base register.
- **The user program** deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses. The final location of a referenced memory address is not determined until the reference is made.
- **We now have two** different types of addresses: logical addresses (in the range 0 to max) and physical addresses (in the range R + 0 to R + max for a base value R). The user program generates only logical addresses and thinks that the process runs in locations 0 to max. However, these logical addresses must be mapped to physical addresses before they are used.
- **The concept of a** logical address space that is bound to a separate physical address space is central to proper memory management.

Logical vs. Physical Address Space

Example:

- If Logical Address = 31 bit, then Logical Address Space = 2^{31} words = 2 G words (1 G = 2^{30})
- If Logical Address Space = 128 M words = $2^7 * 2^{20}$ words, then Logical Address = $\log_2 2^{27} = 27$ bits
- If Physical Address = 22 bit, then Physical Address Space = 2^{22} words = 4 M words (1 M = 2^{20})
- If Physical Address Space = 16 M words = $2^4 * 2^{20}$ words, then Physical Address = $\log_2 2^{24} = 24$ bits

Memory-Management Unit (MMU)

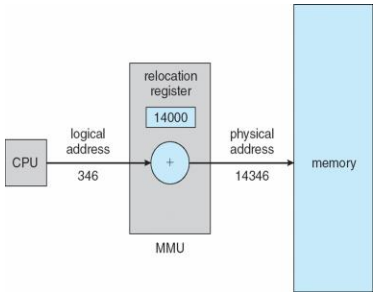
- Hardware device at run time maps virtual to physical address. Many methods possible, covered in the rest of this chapter
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address is bounded to physical address

Dynamic relocation using a relocation register

- It has been necessary for the entire program and all data of a process to be in physical memory for the process to execute. The size of a process has thus been limited to the size of physical memory.
- To obtain better memory-space utilization, we can use dynamic loading. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed.
- When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.
- The advantage of dynamic loading is that a routine is loaded only when it is needed. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.
- In this case, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller. Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a method. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

Dynamic relocation using a relocation register

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading



Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking –linking postponed until execution time
- A stub is included in the image for each library routine reference. The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library.
- If the routine is not already present. When the stub is executed, it checks to see whether the needed routine is already in memory. If it is not, the program loads the routine into memory. Either way, the stub replaces itself with the address of the routine and executes the routine.
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed

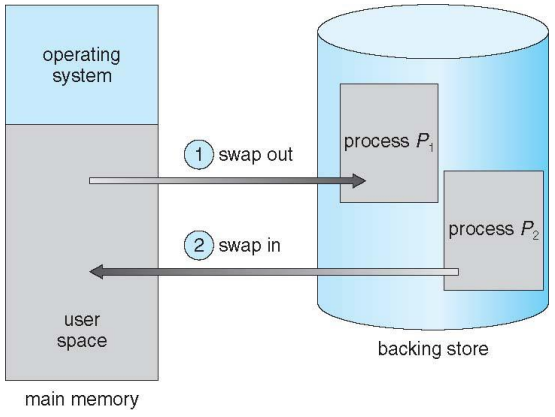
Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed real physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Swapping (Cont.)

- Does the swapped-out process need to swap back into same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Schematic View of Swapping



Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus, swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- it would be useful to know exactly how much memory a user process is using, not simply how much it might be using. Then we would need to swap only what is actually used, reducing swap time.
- System calls to inform OS of memory use via `request_memory()` and `release_memory()`

Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems. It requires too much swapping time and provides too little execution time to be a reasonable memory-management solution.
  - But modified version common
    - Swap only when free memory extremely low

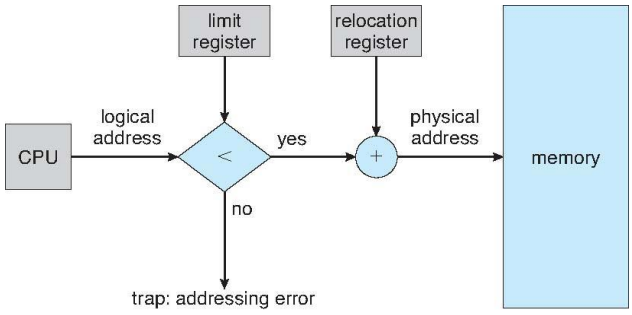
Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually divided into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory

Contiguous Allocation (Cont.)

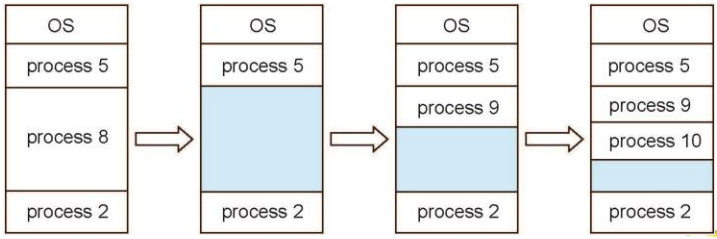
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address ***dynamically***
  - If a device driver (or other operating-system service) is not commonly used, we do not want to keep the code and data in memory, as we might be able to use that space for other purposes. Such code is sometimes called **transient operating-system code**; it comes and goes as needed. Thus, using this code changes the size of the operating system during program execution.

Hardware Support for Relocation and Limit Registers



Multiple-partition allocation

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)



Dynamic Storage-Allocation Problem

How to satisfy a request of size ***n*** from a list of free holes?

- **First-fit**: Allocate the ***first*** hole that is big enough
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
  - Produces the largest leftover hole
- **Next-fit**: It differs from first-fit in that a first-fit allocator commences its search for free space at a fixed end of memory, whereas a next-fit allocator commences its search wherever it previously stopped searching. This strategy is called "modified first-fit"

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Dynamic Storage-Allocation Problem

1. **Given memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB, how would each of the first fit, best fit and worst fit algorithms place processes of 212 KB, 417 KB, 112 KB and 426 KB? Which algorithm makes the most efficient use of memory?**

   Ans: a) First-fit

   The first-fit algorithm selects the first free partition that is large enough to accommodate the request.

   First-fit would allocate in the following manner:

   212 KB => 500 KB partition, leaves a 288 KB partition

   417 KB => 600 KB partition, leaves a 183 KB partition

   112 KB => 288 KB partition, leaves a 176 KB partition

   426 KB would not be able to allocate, no partition large enough!

Dynamic Storage-Allocation Problem

Memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB

Processes of 212 KB, 417 KB, 112 KB and 426 KB

b) Best-fit

The best-fit algorithm selects the partition whose size is closest in size (and large enough) to the requested size.

Best-fit would allocate in the following manner:

212 KB => 300 KB, leaving a 88 KB partition

417 KB => 500 KB, leaving a 83 KB partition

112 KB => 200 KB, leaving a 88 KB partition

426 KB => 600 KB, leaving a 174 KB partition

Dynamic Storage-Allocation Problem

Memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB

Processes of 212 KB, 417 KB, 112 KB and 426 KB

c) Worst-fit

The worst-fit algorithm effectively selects the largest partition for each request.

Worst-fit would allocate in the following manner:

212 KB => 600 KB, leaving a 388 KB partition

417 KB => 500 KB, leaving a 83 KB partition

112 KB => 388 KB, leaving a 276 KB partition

426 KB would not be allowed to allocate as no partition is large enough!

Which algorithm makes the most efficient use of memory?

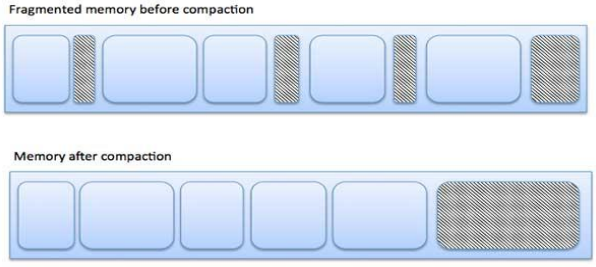
The best-fit algorithm performed the best of the three algorithms, as it was the only algorithm to meet all the memory requests.

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit Statistical analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> This property is known as the **50-percent rule**

Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block

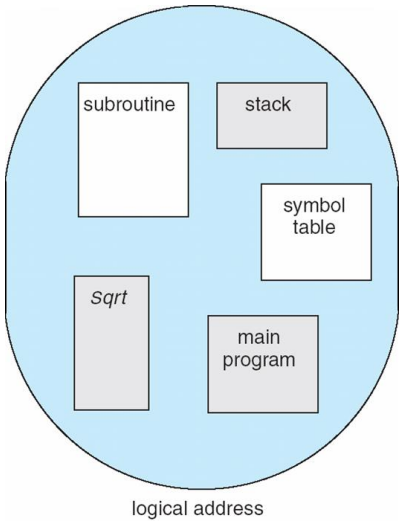


  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems

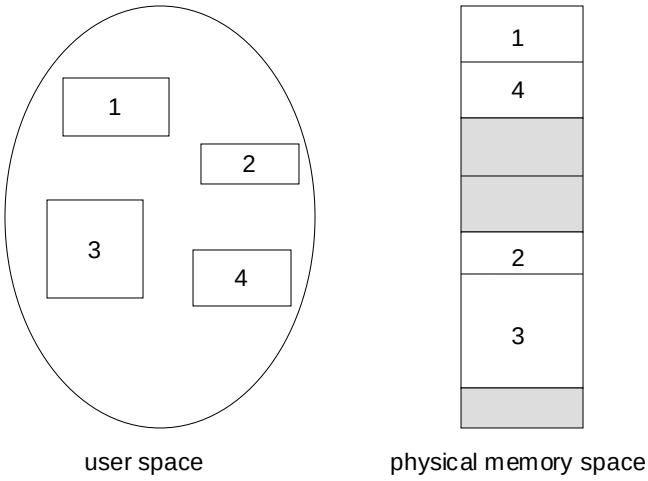
Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:
    - main program
    - procedure
    - function
    - method
    - object
    - local variables, global variables
    - common block
    - stack
    - symbol table
    - arrays

User's View of a Program



Logical View of Segmentation



Segmentation Architecture

- Logical address consists of a two tuple:

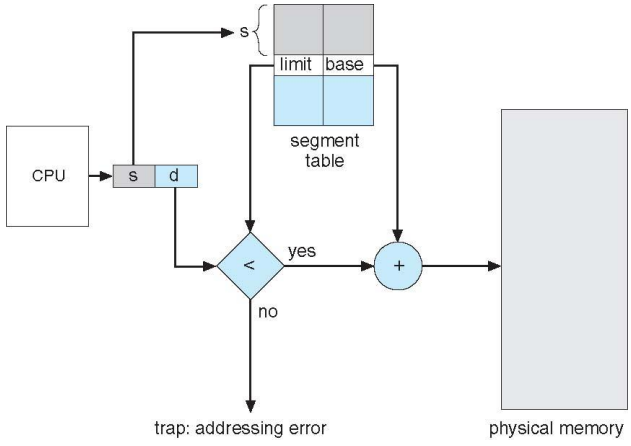
  <segment-number, offset>,

- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Segmentation Hardware



Segmentation Practice Problem

Problem: Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

- What are the physical addresses for the following logical addresses? If the address generates a segment fault, indicate so.
  - a. 0,430
  - b. 1,10
  - c. 2,500
  - d. 3,400
  - e. 4,112

Solution:

a. 219 + 430 = 649

b. 2300 + 10 = 2310

c. illegal reference, trap to operating system

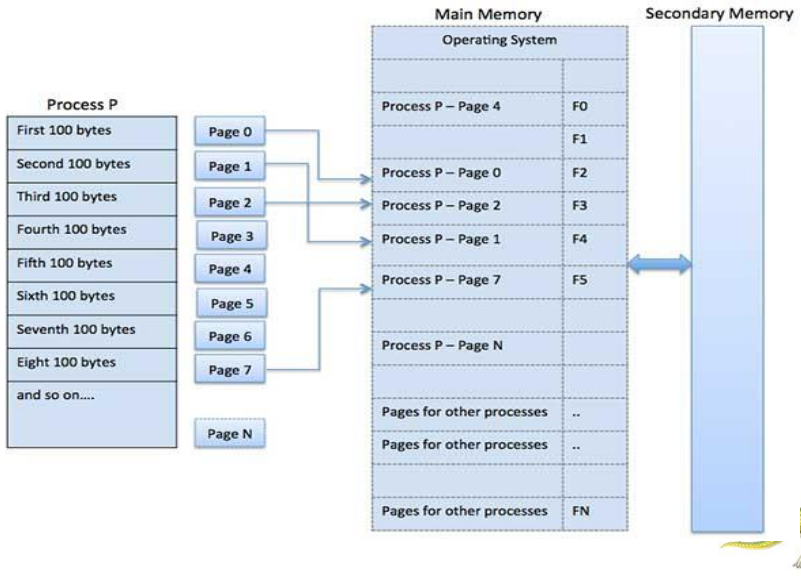
d. 1327 + 400 = 1727

e. illegal reference, trap to operating system

Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation

Paging


Main Memory
Secondary Memory
Operating System
Process P
First 100 bytes
Second 100 bytes
Third 100 bytes
Fourth 100 bytes
Fifth 100 bytes
Sixth 100 bytes
Seventh 100 bytes
Eight 100 bytes
and so on....
Page 0
Page 1
Page 2
Page 3
Page 4
Page 5
Page 6
Page 7
Page N
Process P – Page 4
F0
F1
Process P – Page 0
F2
Process P – Page 2
F3
Process P – Page 1
F4
Process P – Page 7
F5
Process P – Page N
Pages for other processes
..
Pages for other processes
..
Pages for other processes
FN

Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number** (*p*) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (*d*) – combined with base address to define the physical memory address that is sent to the memory unit

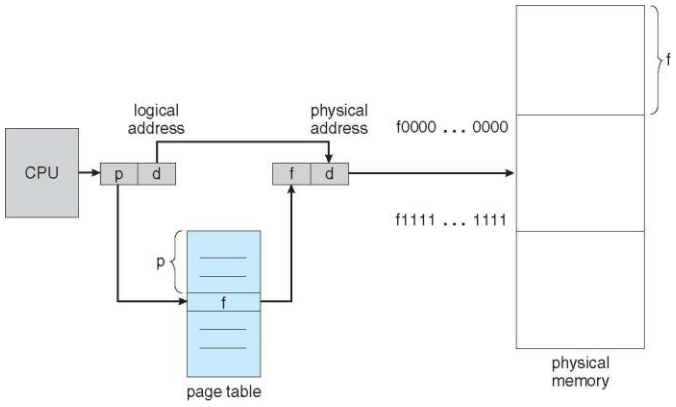
page number	page offset
p	d
m -n	n

  - For given logical address space 2^m and page size 2^n

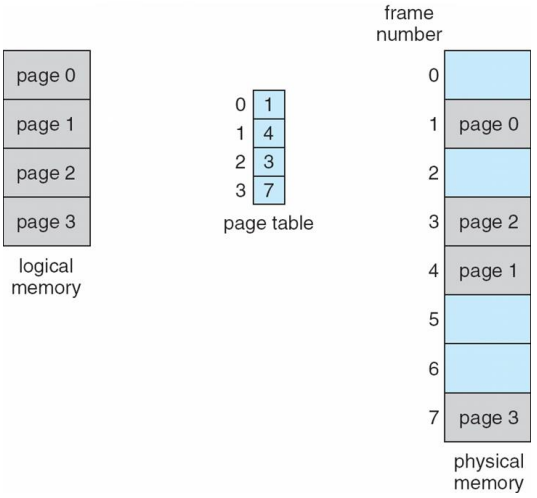
Physical Address is divided into

- **Frame number(f)**
- **Frame offset(d)**

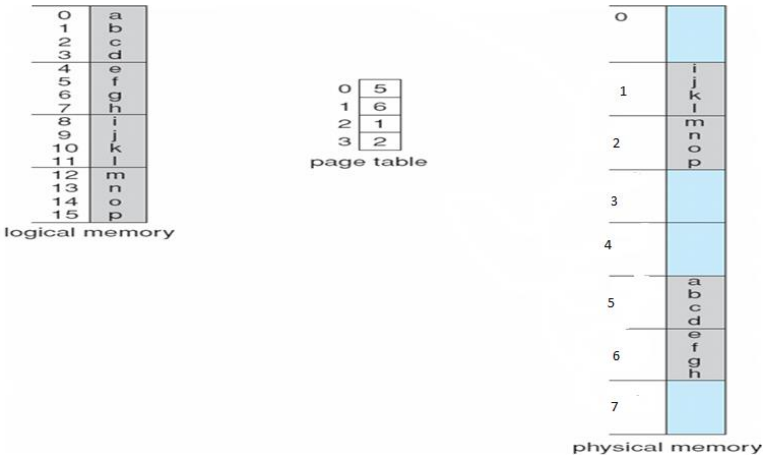
Paging Hardware



Paging Model of Logical and Physical Memory



Paging Example



n=2 (Page size -2^n = 4 Bytes)

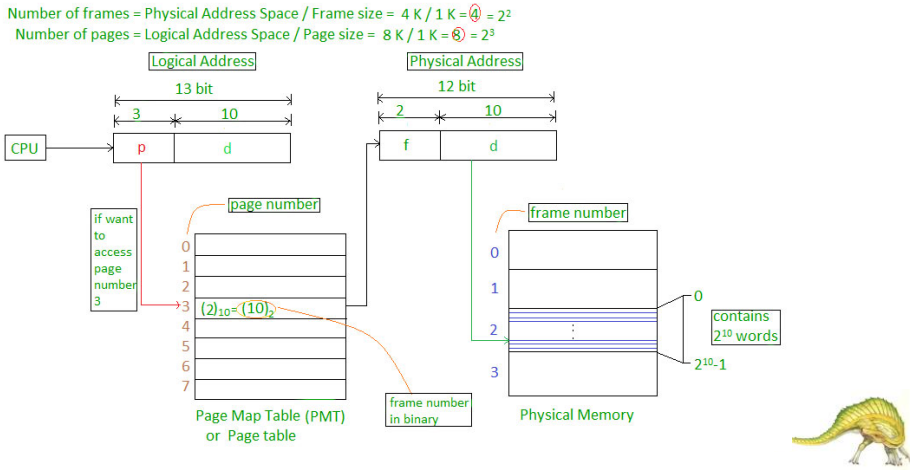
m=4 (size of logical memory 2^m = 16 byte)

32-byte physical memory (2^5 = 32 byte)

Paging (Cont.)

Let us consider an example:

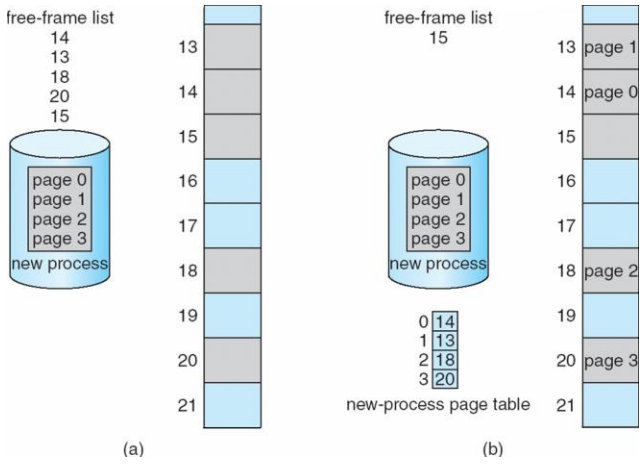
- Physical Address = 12 bits, then Physical Address Space = 4 K
- Logical Address = 13 bits, then Logical Address Space = 8 K
- Page size = frame size = 1 K (assumption)



Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track

Free Frames



Before allocation

After allocation

Paging Practice Problems

Problem: For each of the following decimal virtual addresses, compute the virtual page number and offset for a 4 KB page and for an 8 KB page: 20000, 32768, 60000, 63540.

Solution:

page number = A / page_size

(Note: This is the page number within the process address space)

Address in the process, A = 20,000

page size = 4 KB

page number = 20000 / 4k = 20,000 / 4096 = 4.xxx = truncate to 4

offset = A mod page_size

(Note: this is the distance from the beginning of the page)

page offset = 20000 mod 4k = 20,000 mod 4096 = 3616

OR

Offset = 20000 – 4*4096 = 3616

	20000	32768	60000	63540
4 KB	4, 3616	8, 0	14, 2656	15, 2100
8 KB	2, 3616	4, 0	7, 2656	7, 6196

Paging Practice Problems

Problem: For 32-bit hexadecimal address FFFFFFFF, what is the virtual page number and offset for a 4 KB page?

Solution:

Hexadecimal Address A = FFFFFFFF = $(4294967295)_{10}$

Virtual Address Space 2^32 B

Page size = 4 KB = 2^12 B (12 bit to denote offset)

Virtual Page Number = 2^32 / 2^12 = 2^20 (20 bit to denote Page no.)

page number = A / page_size

4294967295 / 4096 = 1048576

offset = A mod page_size

4294967295 mod 4096 = 4095

Paging Practice Problems

Problem: **Consider a logical address space of eight pages of 1024 words each, mapped onto a physical memory of 32 page frames.**

(a) **How many bits are in logical address? How many bits represent for the page number and for the offset within a page?**

(b) **How many bits are in physical address? How many bits represent for the page number and for the offset within a page?**

Solution:

a) How many bits are in the logical address?

The logical address is split into two parts: the page address and then the offset.

The page address must be able to reference 8 (or 2^3) distinct pages. This requires 3 address bits.

The offset must be able to reference 1,024 (or 2^10) distinct words on each page. This requires 10 address bits.

Therefore, 13 bits are required for the logical address.

Paging Practice Problems

Problem: **Consider a logical address space of eight pages of 1024 words each, mapped onto a physical memory of 32 page frames.**

(a) **How many bits are in logical address? How many bits represent for the page number and for the offset within a page?**

(b) **How many bits are in physical address? How many bits represent for the page number and for the offset within a page?**

Solution:

b) How many bits are in the physical address?

The physical address is also split into two parts: the frame address and then the offset.

The number of frames is 32 (or 2^5), so that will require 5 bits.

The size of the frame is the same as the size of the page, therefore this will require 10 bits, as described above.

Therefore, 15 bits are required for the physical address.

Paging Practice Problems

Problem: A machine has 48-bit virtual addresses and 32-bit physical addresses. Pages are 8 KB. How many virtual and physical pages can the address space support? How many entries are needed for the page table?

Solution:

- With 8K pages and a 48-bit virtual address space, the number of virtual pages are (2^48)/(2^13), which is 2^35
- With 8K pages and a 32-bit Physical address space, the number of Physical pages are (2^32)/(2^13), which is 2^19
- The Page Table will have 2^35 = 2^5 * 2^10 * 2^10 * 2^10 = 34359738368 entries

Paging Practice Problems

Problem: On a system using paging and segmentation, the virtual address space consists of upto 8 segments where each segment can be up to 2^{29} bytes long. The hardware pages each segment into 256 byte pages. How many bits in the virtual address to specify the:

- **(a) Segment number?**
- **(b) Page number?**
- **(c) Offset within a page?**
- **(d) Entire virtual address?**

Solution:

i) Segment number: Virtual Address Space consists of up to 8 segments so 8 = 2^3

So 3 bits are needed to specify segment number

i) Page Number: Hardware pages each segment into 256 bytes pages.

So 256 = 2^8 byte pages

Size of segment is 2^29 bytes

So 2^29 / 2^8 = 2^21 pages

So 21 bits are required to specify the page no.

i) Offset within the page: For 2^8 byte page, 8 bits are needed.

ii) Entire virtual address = segment number + page number + offset

= 3 + 21 + 8

= 32

Paging Practice Problems

Problem: A computer provides each process with 65536 bytes of address space divided into pages of 4096 bytes. A particular program has a text size of 32768 bytes, a data size of 16, 386 bytes and a stack size of 15, 870 bytes. Will the program fit in the given address space? If the page size is 512 bytes, would it fit? Explain.

Solution:

4096 bytes:

65536/4096 = 16 pages (available pages)

32768/4096 = 8 pages

16386/4096 = 4.0005 pages -> 5 pages

15870/4096 = 3.875 pages -> 4 pages

8+5+4 = 17 pages > 16 pages which is doesn't fit

512 bytes:

65536/512 = 128 pages (Available to allocate)

32768/512 = 64 pages

16386/512 = 32.004 pages -> 33 pages

15870/512 = 30.996 pages -> 31 pages

64+33+31 = 128 pages which is fit

Paging Practice Problems

Problem: Consider a system with byte-addressable memory, 32-bit logical addresses, 4 kilobyte page size and page table entries of 4 bytes each. What is the size of the page table in the system in megabytes?

Solution:

Number of entries in page table = 2^32 / 4Kbyte

= 2^32 / 2^12

= 2^20

Size of page table = (No. page table entries)* (Size of an entry)

= 2^20 * 4 bytes

= 2^22

= 4 Megabytes

Important Formulaes:

1. **Number of Pages / entries in page table = (virtual address space size)/ (page size)**
2. **Size of page table = total number of page table entries) *(size of a page table entry)**
3. **No. of bits required to specify the frame in Page Table= Entire Physical Address Space – Displacement**

Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**PTLR**) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- The two-memory access problem can be solved using a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)

Implementation of Page Table (Cont.)

- The hardware implementation of page table can be done by using dedicated registers. But the usage of register for the page table is satisfactory only if page table is small. If page table contain large number of entries, then we can use TLB(translation Look-aside buffer), a special, small, fast look up hardware cache.
- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access

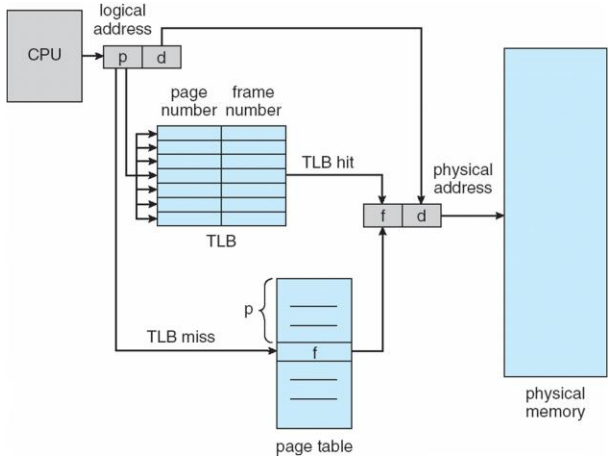
Associative Memory

- Associative memory – parallel search

Page #	Frame #

- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

Paging Hardware With TLB

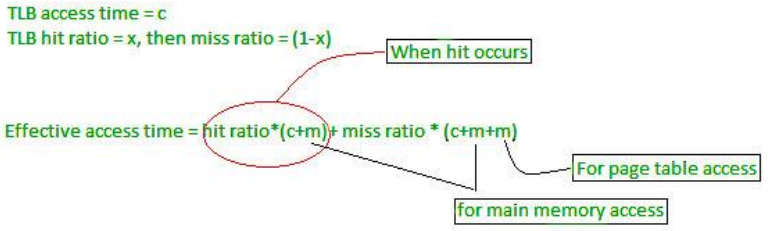


Effective Access Time

Main memory access time = m

If page table are kept in main memory,

Effective access time = m(for page table) + m(for particular page in page table)



Effective Access Time

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- An 80-percent hit ratio means that we find the desired page number in the associative registers 80 percent's of the time. If it takes 20 nanoseconds to search the associative registers, and 100 nanoseconds to access memory, then a mapped memory access takes 120 nanoseconds when the page number is in the associative registers. If we fail to find the page number in the associative registers (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds), and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds.
- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
- **Effective Access Time** (**EAT**)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$

Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access

- EAT = 0.80 x 120 + 0.20 x 220 = 140ns
- Consider more realistic hit ratio -> α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 120 + 0.01 x 220 = 122ns

Paging Practice Problems

Problem: A m/c has a 32- bit address space and an 8 KB page. The page table is entirely in H/W, with one 32-bit word per entry. When a process starts, the page table is copied to the H/W from memory, at one word every 100 nsec. If each process runs for 100msec (including the time to load the PT), what fraction of CPU time is devoted to loading the page tables?

Solution:

8 KB pages will lead to 13 bits for the offset → 2^{19} entries in the page table

Time to upload page table = 2^{19} x 100 ns = 52.4288 msec

Loading of pages takes 52 msec.

If each process runs for 100 msec (including the time to load the page table), then 52 msec is used to load the page table and remaining 48 msec is used for running.

So total of 52% of time is used in loading the page tables.

Paging Practice Problems

1. **Problem: Consider a paging system with the page table stored in memory.**
   i. **If a memory reference takes 200 nsecs, how long does a paged memory reference take?**
   ii. **If we add TLBs, and 75 % of all the page table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page table entry in the TLB takes zero time)**

Solution:

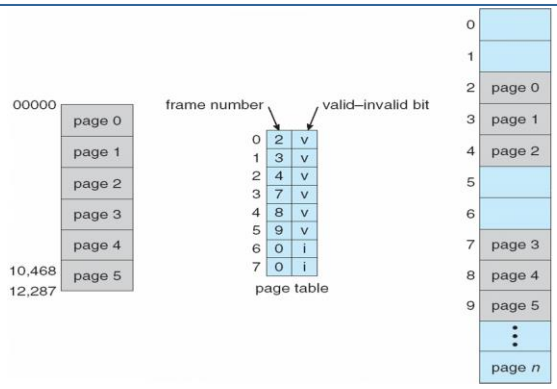
a. 400 nanoseconds: 200 nanoseconds to access the page table and 200 nanoseconds to access the word in memory.

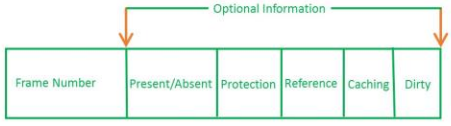
b. Effective access time = 0.75 × (200 nanoseconds) + 0.25 × (400 nanoseconds) = 250 nanoseconds.

Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space
  - Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel

Valid (v) or Invalid (i) Bit In A Page Table





PAGE TABLE ENTRY

Paging Practice Problems

Problem: **On a paging system with 2^{24} bytes of physical memory, 256 pages of logical address space, and a page size of 2^{10} bytes, how many bits are needed to store an entry in the page table? Assume each page table entry contains a valid/invalid bit in addition to the page frame number.**

Solution:

No. of bits required to specify the frame in Page Table

= 24 (Entire Physical Address Space) – 10 (Displacement) = 14 bits

bits required in each page table entry (assume each page table entry includes a valid/invalid bit).

= 14 (Page Frame) + 1 (Valid/Invalid) = 15 bits

Paging Practice Problems

Problem: On a paging system with a page table containing 64 entries of 11 bits (including valid/invalid bit) each, and a page size of 512 bytes,

- (a) How many bits in the logical address specify the page number and offset within the page?
- (b) How many bits are in a logical address and what is the size of logical address space?
- (c) How many bits in the physical address specify the page frame number and offset within the frame?
- (d) How many bits are in a physical address and what is the size of physical address space?

Solution:

a)number of bits needed to specify page number : number of entries in page table= number of pages in logical address. 64 entries so 64 pages so number of bits= log64= 6bits

number of bits for offset within the page is nothing but number of bits needed to address each word within the page = log(512)=9 bits

b)size of logical address=<pagenumber,offset>=6+9 =15 bits

if you are asking size of logical address space, then number of pages*pagesize= 64*512 bytes= 32KB

c)number of bits in physical address = <framenum,offset>

number of bits to address a frame= 10 (as entry size is 10 bits excluding valid/invalid)

number of bits =10+9=19

d)size of physical address= number of frames*size of each frame= 2^10*2^9 = 512KB

Paging Practice Problems

Problem: Consider a machine with 64 MB physical memory and a 32 bit virtual address space. If the page size is 4 KB, what is the approximate size of page table?

Solution:

A page entry is used to get address of physical memory. Here we assume that single level of Paging is happening. So, the resulting page table will contain entries for all the pages of the Virtual address space.

Number of entries in page table = (virtual address space size)/(page size)

Using above formula we can say that there will be 2^(32-12) = 2^20 entries in page table.

No. of bits required to address the 64MB Physical memory = 26.

So there will be 2^(26-12) = 2^14 page frames in the physical memory. Therefore, each page table entry will contain 14 bits address of the page frame and 1 bit for valid-invalid bit.

Since memory is byte addressable. So we take that each page table entry is 16 bits i.e. 2 bytes long.

Size of page table = (total number of page table entries) *(size of a page table entry)

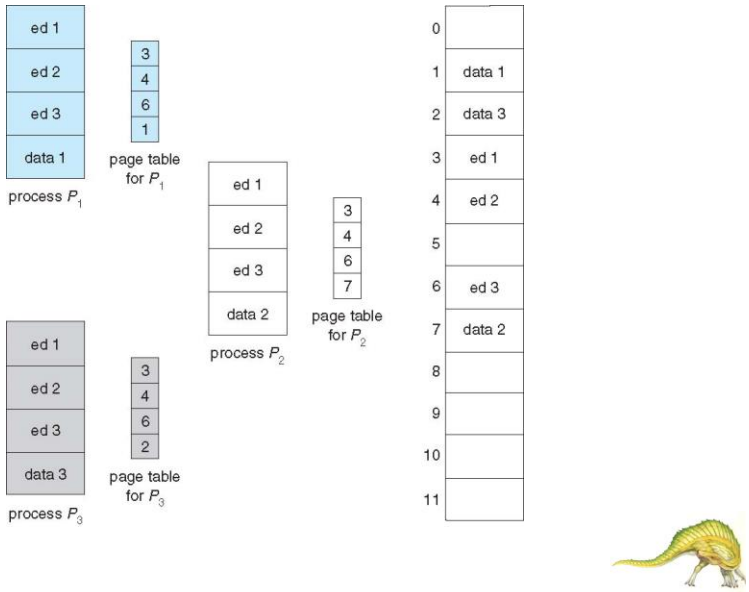
= (2^20 *2) = 2MB

Shared Pages

- An advantage of paging is the possibility of sharing common code.
- Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users
- **Shared code**
  - One copy of read-only (**reentrant**-non-self-modifying code) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for inter-process communication if sharing of read-write pages is allowed
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Only one copy of the editor need to be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB—a significant savings

Shared Pages Example



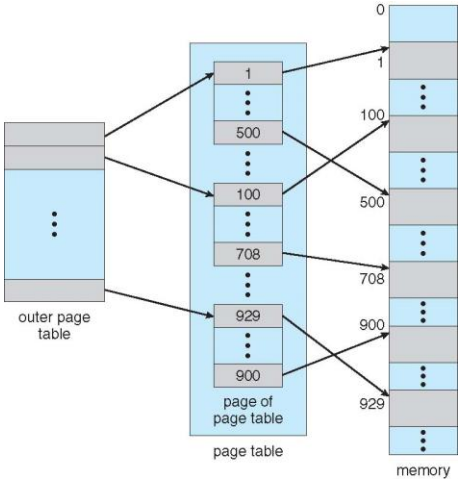
Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
    - That amount of memory used to cost a lot
    - Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces.
    - We can accomplish this division using following ways.
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

Two-Level Page-Table Scheme



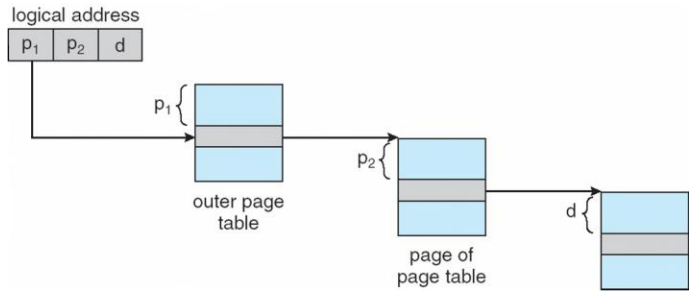
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
12	10	10

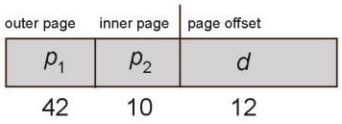
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address-Translation Scheme



64-bit Logical Address Space

- Even two-level paging scheme not sufficient for 64-bit logical address space
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



  - Outer page table has 2^{42} entries
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} outer page table is still 2^{34} bytes in size
    - And possibly 4 memory access to get to one physical memory location
    - The next step would be a four-level paging scheme, where the second-level outer page table itself is also paged, and so forth.

Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Two Level Paging Practice Problems

Problem: A computer with a 32-bit address uses a two-level page table. Virtual addresses are split into 9-bit toplevel page table field, an 11 bit second-level page table field, and an offset. How large are the pages and how many are there in the address space?

Solution:

Offset = 32 – 9 – 11 = 12 bits

Page size = 2^12 B = 4 KB

Total number of pages possible = 2^9 * 2^11 = 2^20

Two Level Paging Practice Problems

1. **Problem:** **Suppose we have a computer system with a 44-bit virtual address, page size of 64K, and 4 bytes per page table entry.**

(i) **How many pages are in the virtual address space?**

(ii) **Suppose we use two-level paging and arrange for all page tables to fit into a single page frame. How will the bits of the addresses be divided up?**

(iii) **Suppose we have a 4GB program such that the entire program and all necessary page tables are in memory. How much memory (in page frames) is used by the program, including its page tables?**

Solution:

i) 64 K= 2^16), so we need 16 out of 44 bit logical address for the offset. Then we have 44 - 16 = 28 bits left for the page number. There are, therefore, 2 ^28 = 268435456 pages in the virtual address space.

ii) Since we have 4 bytes per page table entry, one-page frame can fit 64 KB/4 B = 2^14 entries. We need 14 bits to index into a page of the page table. Then we have 44 - 14 - 16 = 14 bits left for the page number.

43 42 ... 30	29 28 ... 16	15 14 ... 0
|--outer--|	|--inner--|	|-offset-|

Two Level Paging Practice Problems

1. **Problem:** **Suppose we have a computer system with a 44-bit virtual address, page size of 64K, and 4 bytes per page table entry.**

(i) **How many pages are in the virtual address space?**

(ii) **Suppose we use two-level paging and arrange for all page tables to fit into a single page frame. How will the bits of the addresses be divided up?**

(iii) **Suppose we have a 4GB program such that the entire program and all necessary page tables are in memory. How much memory (in page frames) is used by the program, including its page tables?**

Solution:

iii) The address would be divided up as 14 | 14 | 16 since we want page table pages to fit into one page and we also want to divide the bits roughly equally.

Since the process' size is 4GB = 2^32 B, we assume what this means is that the total size of all the distinct pages that the process accesses is 2^32 B. Hence, this process accesses 2^32 / 2^16 = 2^16 pages. The bottom level of the page table then holds 2^16 references. We know the size of each bottom level chunk of the page table is 2^14 entries. So we need 2^16 / 2^14 = 2^2 of those bottom level chunks.

The total size of the page table is then:

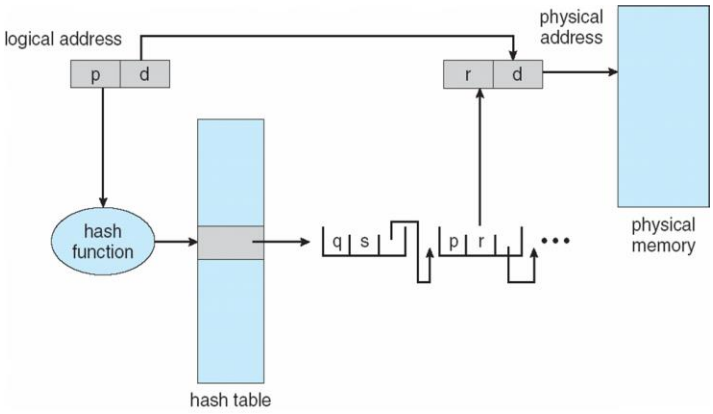
//size of the outer page table		//total size of the inner pages		
1 * 2^14 * 4	+	2^2 * 2^14 * 4	=	2^16 * (1 + 4) ~64 KB

So Total memory (in page frames) is used by the program, including its page tables = 2^16 +2^16 = 2^17 B

Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

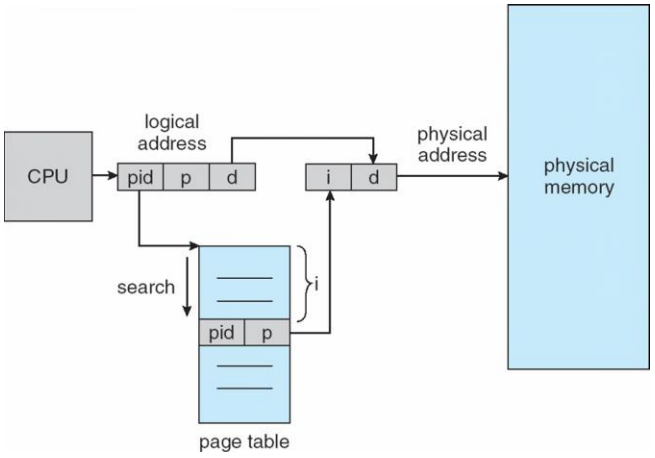
Hashed Page Table



Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

Inverted Page Table Architecture



End of Chapter 8