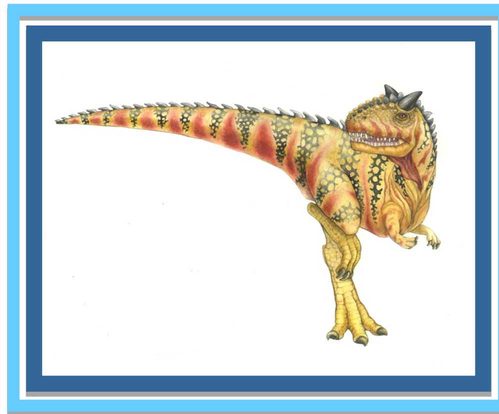
Chapter 8: Main Memory



Silberschatz, Galvin and Gagne ©2013

Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Paging

Objectives

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

Background

- The main purpose of a computer system is to execute programs.
- Program must be brought (from disk) into memory and placed within a process for it to be executed.

- To improve both the utilization of the CPU and the speed of its response to users → a system must keep several processes in memory.

Main memory Management

Background

- Memory consists of a large array of bytes, each with its own address.
- Main memory and registers(built into the processor itself) are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests.
- Registers that are built into the CPU are generally accessible within one cycle of the CPU clock
- Main memory can take many cycles, causing the processor to **stall**

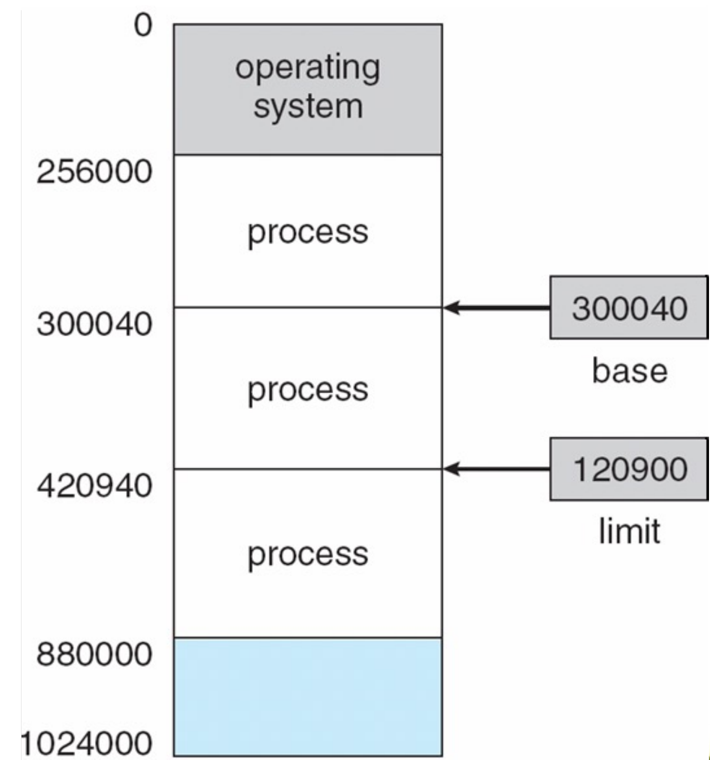
Remedy ➔ **Cache** sits between main memory and CPU registers

- Protection of memory required to ensure correct operation and controlled access.

Base and Limit Registers

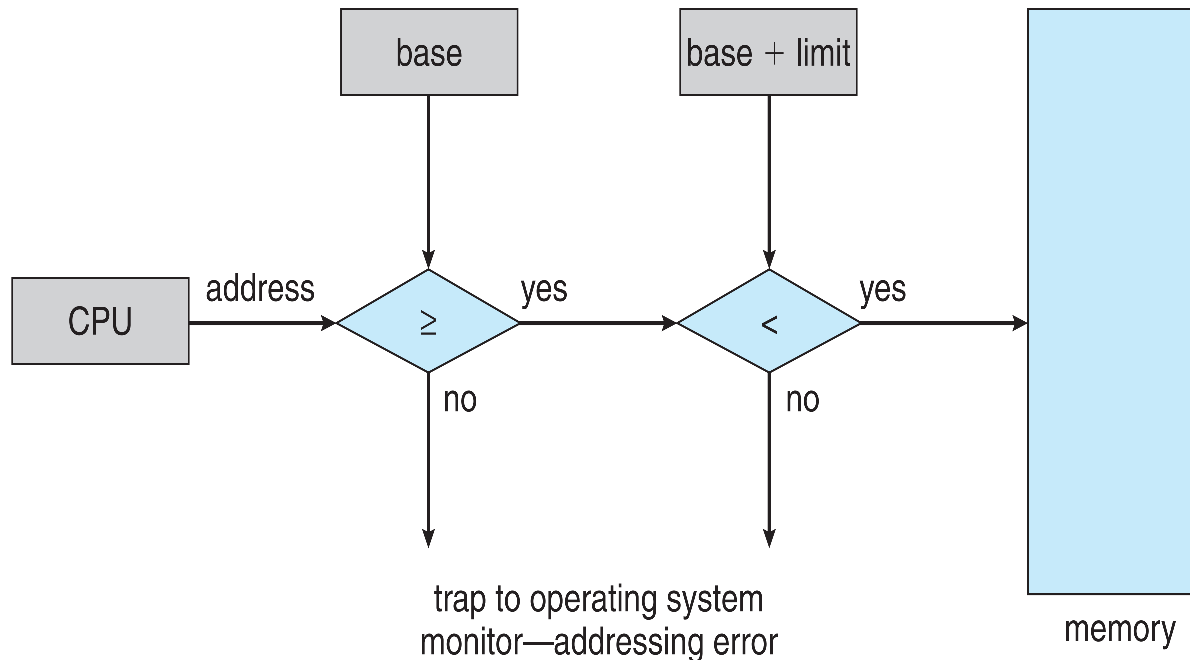
- Each process has a separate memory space.
- To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access.
- A pair of **base** and **limit registers** define the logical address space
  - The **base register** holds the smallest legal physical memory address.
  - the **limit register** specifies the size of the range.

For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420940 (inclusive).



Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user

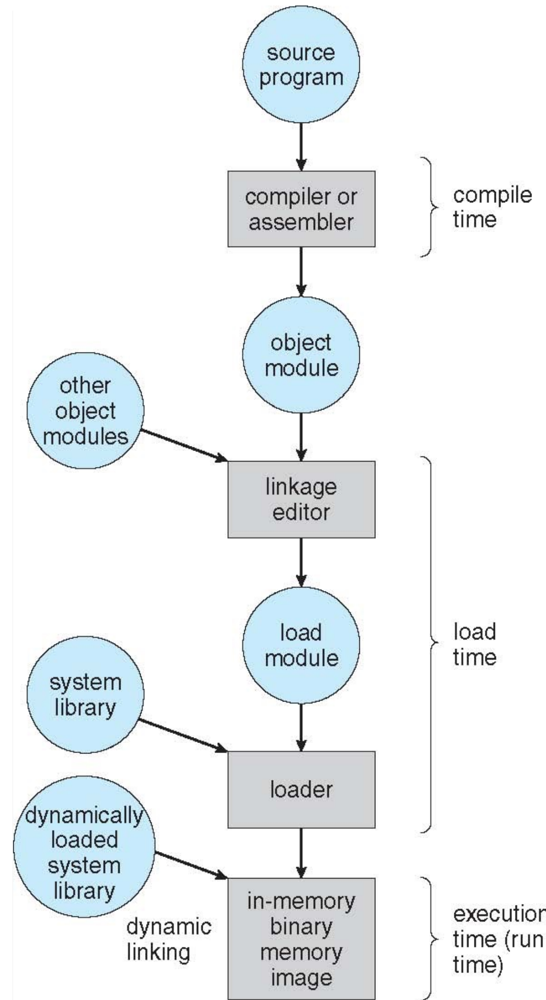


- This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

Address Binding

- A program resides on a disk as a binary executable file
- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding maps one address space to another

Multistep Processing of a User Program



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 memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address mappings (e.g., base and limit registers)

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**
  - **Physical address** – address seen by the memory unit
- 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 generated by a program

Physical address space is the set of all physical addresses generated by a program

- The run time mapping from virtual to physical address is done by a hardware device called the **Memory-Management Unit (MMU)**

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- 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**
  - MS-DOS on Intel 80x86 used 4 relocation registers
- 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 bound to physical addresses

Dynamic relocation using a relocation register

relocation register

14000

CPU

logical address

346

+

physical address

14346

memory

MMU

Dynamic Loading

- A 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

- With **dynamic loading**:
  - 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

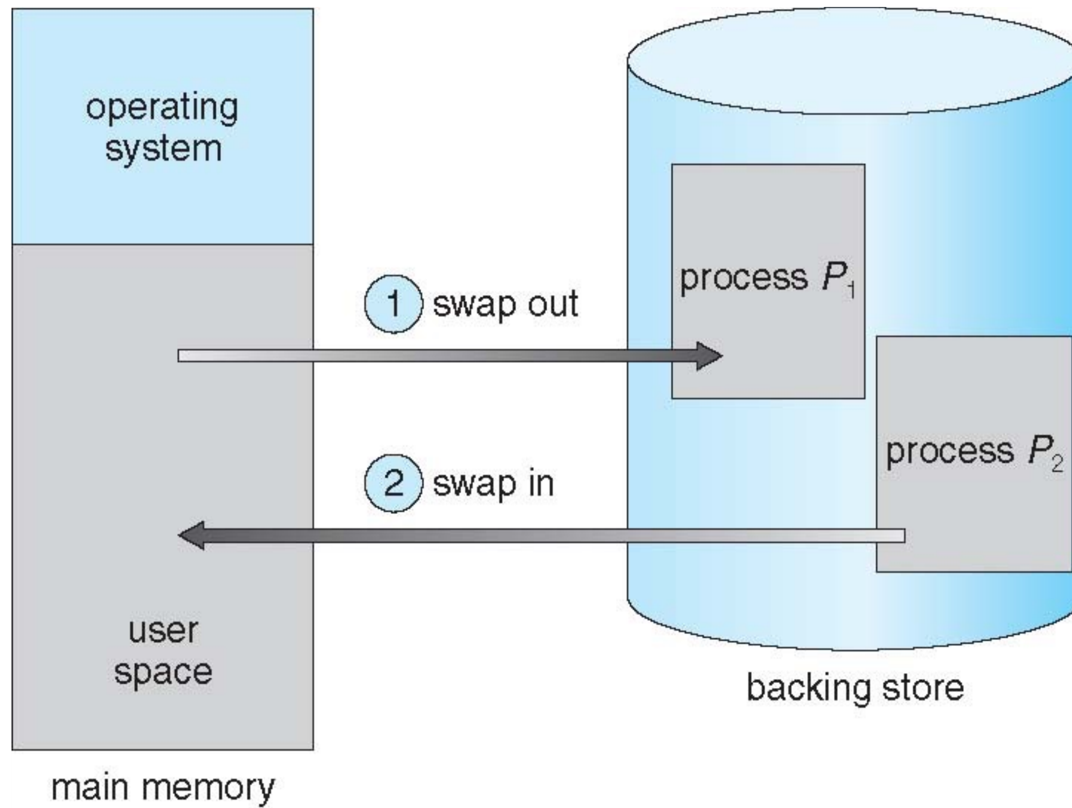
Dynamic Linking

- **Dynamically linked libraries** are **system libraries** that are linked to user programs when the programs are run
- **Static linking** – system libraries and program code combined by the loader into the binary program image ➔ waste disk space and main memory
- **Dynamic linking** –linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present.
- **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
- Under this scheme, all processes that use a language library execute only one copy of the library code.
- Dynamic linking is particularly useful for libraries ➔ **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 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

Schematic View of Swapping



Swapping (Cont.)

- **Does the swapped out process need to swap back in to same physical addresses?**
  - Depends on address binding method

- Swapping is constrained by other factors as well.
  - If we want to swap a process, we must be sure that it is completely idle
  - A process may be waiting for an I/O operation ➔this process cannot be swapped.

- 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

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)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`

Swapping on Mobile Systems

- Not typically supported
  - Flash-memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to the flash memory for fast restart
  - Both OSes support paging (memory management abilities)

Contiguous Allocation

- Main memory must support both OS and user processes
- Main memory is a limited resource→ must be allocated efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
    - each process is contained in a single section of memory that is contiguous to the section containing the next process.

Contiguous Allocation

Memory Allocation

How to allocate memory ?

Divide memory into several fixed-sized partitions.

Each partition may contain exactly one process.

When a partition is free, a process is selected from the input queue and is loaded into the free partition.

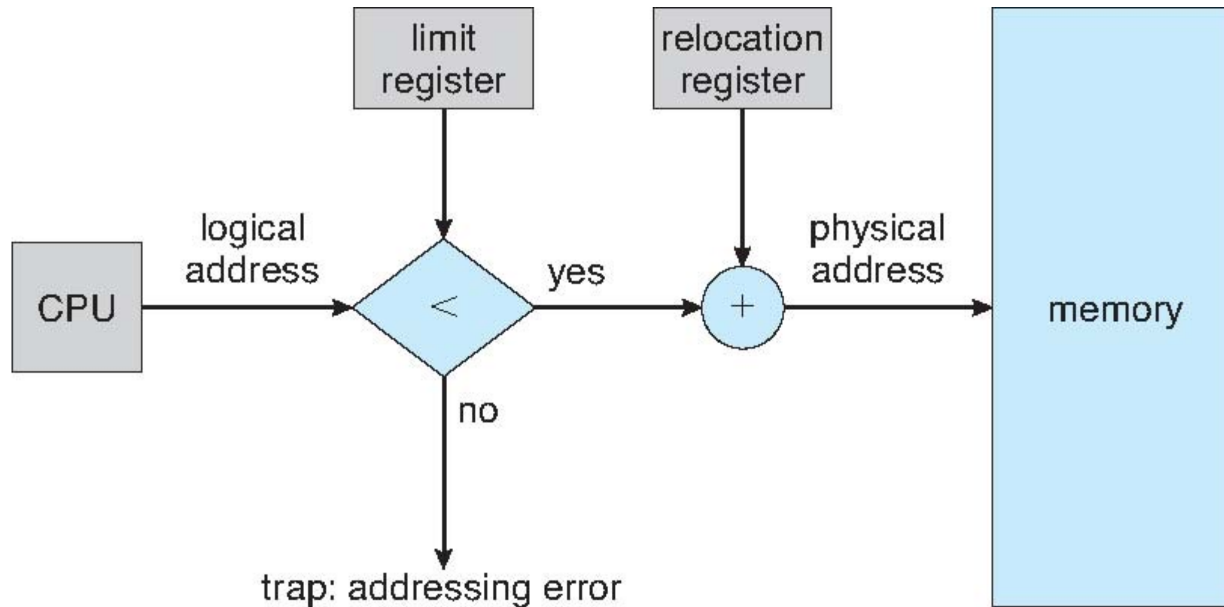
When the process terminates, the partition becomes available for another process.

This is one of the simplest methods for allocating memory

Contiguous Allocation (Cont.)

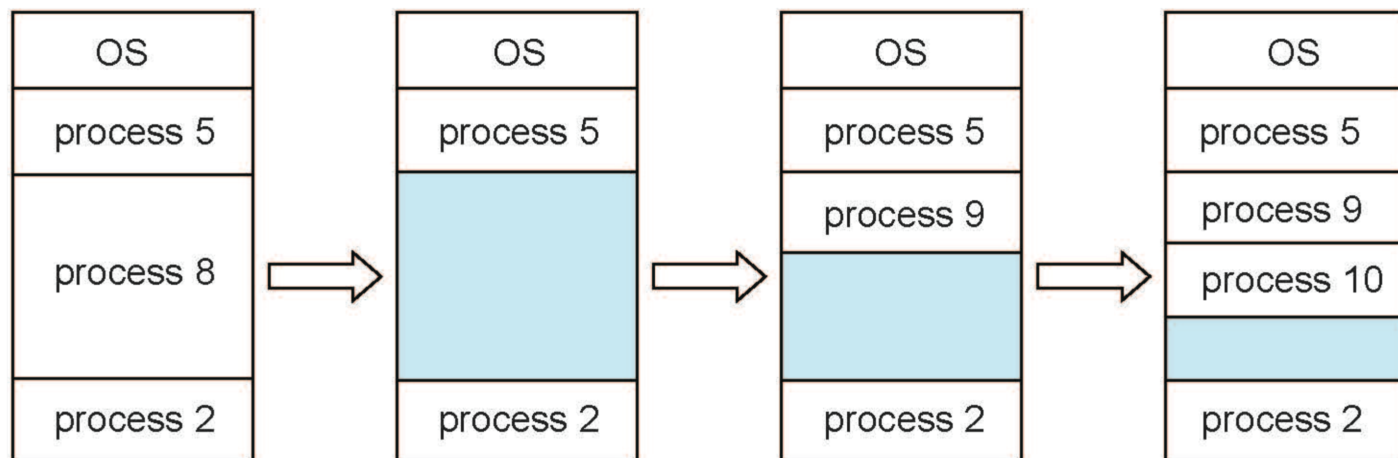
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - The relocation-register scheme provides a way to allow the operating system's size to change dynamically.
  - For example, the operating system contains code and buffer space for device drivers. 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,
  - Such code is sometimes called **transient** operating-system code (it comes and goes as needed)

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 partitions/holes in main memory?

- **First-fit**: Allocate the ***first*** hole that is big enough
  - Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended.
  - We can stop searching as soon as we find a free hole that is large enough.

- **Best-fit**: Allocate the ***smallest*** hole that is big enough
  - We must search the entire list, unless the list is ordered by size.
  - Produces the smallest leftover hole

- **Worst-fit**: Allocate the ***largest*** hole;
  - We must search the entire list, unless it is sorted by size.
  - Produces the largest leftover hole

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

Multiple-partition allocation: Examples

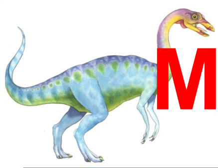
1. Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)? Which algorithm makes the most efficient use of memory?

First-fit:
212K is put in 500K partition
417K is put in 600K partition
112K is put in 288K partition (new partition 288K = 500K - 212K)
426K must wait

Best-fit:
212K is put in 300K partition
417K is put in 500K partition
112K is put in 200K partition
426K is put in 600K partition

Worst-fit:
212K is put in 600K partition
417K is put in 500K partition
112K is put in 388K partition
426K must wait

In this example, best-fit turns out to be the best.



Multiple-partition allocation: Examples

Example

Consider a swapping system in which memory consists of the following whole sizes in memory order: 10K, 4k, 20k, 18k, 7k, 9k, 12k, and 15k. Which hole is taken for successive segment request of i)12k, ii)10k, iii)9k for first fit? Now repeat the question for best fit and worst fit.

First Fit		
12k	→	20k
10k	→	10k
9k	→	18k

Best Fit		
12k	→	12k
10k	→	10k
9k	→	9k

Worst Fit		
12k	→	20k
10k	→	18k
9k	→	15k

Fragmentation

- As processes are loaded and removed from memory, the free memory space is broken into little pieces➔ fragmentation

- **Types of Fragmentation:**
  - **External Fragmentation** – total memory space exists to satisfy a request, but the available spaces are not contiguous (storage is fragmented into a large number of small holes)
  - **Internal Fragmentation** – allocated memory may be slightly larger than a process's requested memory (unused memory that is internal to a partition)
  - Solution: Best fit approach.

- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 of memeory may be unusable -> **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

Paging

- **Paging** is a memory-management scheme that permits the physical address space of a process to be non-contiguous.
- 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

Process P1

PP1
PP1
PP1
PP1

Pages of Process P1

Process P2

PP2
PP2
PP2
PP2

Pages of Process P2

Physical memory divided into frames

Process P3

PP3
PP3
PP3
PP3

Pages of Process P3

Process P4

PP4
PP4
PP4
PP4

Pages of Process P4

Paging

Process P1

PP1
PP1
PP1
PP1

Pages of Process P1

Process P2

PP2
PP2
PP2
PP2

Pages of Process P2

PP1
PP1
PP1
PP1
PP2
PP2
PP2
PP2

Physical memory divided into frames

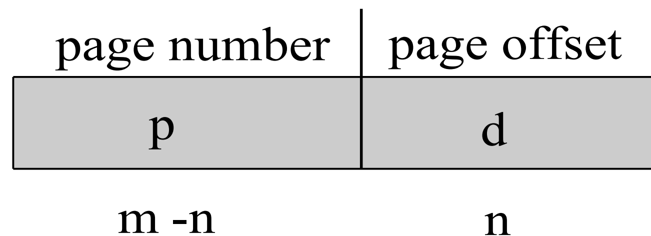
Process P5

PP5
PP5
PP5
PP5
PP5
PP5
PP5
PP5

Pages of Process P5

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 (the displacement within the page)



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

Paging Hardware

logical address

physical address

CPU

p d

f d

f0000 . . . 0000

f1111 . . . 1111

p

f

page table

f

physical memory

Paging Model of Logical and Physical Memory

logical memory

page
page 0
page 1
page 2
page 3

page table

page	frame
0	1
1	4
2	3
3	7

physical memory

frame number	
0	
1	page 0
2	
3	page 2
4	page 1
5	
6	
7	page 3

Paging Example

Address	Contents
0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

Page	Frame
0	5
1	6
2	1
3	2

page table

Address	Contents
0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

physical memory

n=2 and m=4   32-byte memory and 4-byte pages

Paging Example

Translation of Logical Address to Physical Address using Page Table

Logical Memory

Page	Address	Content
Page 0	0	a
	1	b
	2	c
	3	d
Page 1	4	e
	5	f
	6	g
	7	h
Page 2	8	i
	9	j
	10	k
	11	l
Page 3	12	m
	13	n
	14	o
	15	p

Page Table

Page	Frame
0	5
1	6
2	1
3	2

Physical Memory

Frame	Address	Content
Frame 0	0	
	1	
	2	
	3	
Frame 1	4	i
	5	j
	6	k
	7	l
Frame 2	8	m
	9	n
	10	o
	11	p
Frame 3	12	
	13	
	14	
	15	
Frame 4	16	
	17	
	18	
	19	
Frame 5	20	a
	21	b
	22	c
	23	d
Frame 6	24	e
	25	f
	26	g
	27	h
Frame 7	28	
	29	
	30	
	31	

Logical address x maps to physical address y

X= ((Frame X Page size)+offset)

Paging (Cont.)

- With paging scheme, we have no external fragmentation, but we may have some internal fragmentation

- **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 page + 1 byte ➔ n+1 frames
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?

    But each page table entry takes memory to track (overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases)
  - Page sizes growing over time (4 KB and 8 KB in size)
    - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory now very different
- By implementation process can only access its own memory

Free Frames

free-frame list
14
13
18
20
15

new process: page 0, page 1, page 2, page 3

13, 14, 15, 16, 17, 18, 19, 20, 21

(a)

free-frame list
15

new process: page 0, page 1, page 2, page 3

0	14
1	13
2	18
3	20

new-process page table

13 page 1
14 page 0
15
16
17
18 page 2
19
20 page 3
21

(b)

Before allocation

After allocation

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 by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)

Implementation of Page Table (Cont.)

- 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)
- If the page number is found, its frame number is available and is used to access memory → TLB Hit
- If the page number is not found, a memory reference to the page table must be made → TLB Miss

- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered (e.g., replacing the least used entries)
  - Some entries can be **wired down** (meaning that they cannot be removed from the TLB) 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

CPU
logical address
p
d
page number
frame number
TLB hit
physical address
f
d
TLB
p
TLB miss
f
page table
physical memory

Paging Hardware With TLB

START

CPU CHECKS TLB

PAGE TABLE ENTRY IN TLB?

YES

NO

ACCESS PAGE TABLE

PAGE IN MAIN MEMORY?

NO

YES

UPDATE TLB

CPU GENERATES PHYSICAL ADDRESS

CPU READS PAGE FROM DISK

ACTIVATE I/O HARDWARE

PAGE TRANSFERRED FROM DISK TO MEMORY

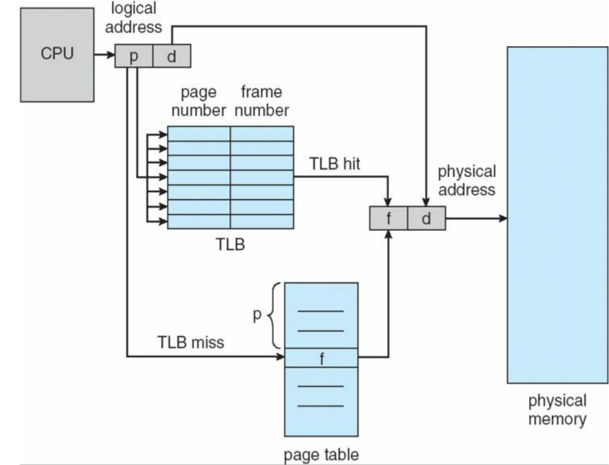
MEMORY FULL?

YES

NO

PERFORM PAGE REPLACEMENT

PAGE TABLES UPDATED



Effective Access Time

- **TLB_hit_time := TLB_search_time + memory_access_time**
- **TLB_miss_time := TLB_search_time + page_frame_table_access_time + memory_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

For example, an 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time.

- **Effective Access Time** (**EAT**)

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

  - **Hit: 1 memory access + 1 TLB access => $(1 + \varepsilon)$**
  - **Miss: 2 memory accesses + 1 TLB access => $(2 + \varepsilon)$**

  - **EAT := TLB_hit_time * hit_ratio + TLB_miss_time * (1- hit_ratio)**

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

- EAT = 0.80 x 120 + 0.20 x 220 = 140ns **s.t.120 = (100+20), 220=(2*100+20)**

- Consider slower memory but better hit ratio -> α = 98%, ε = 20ns for TLB search, 140ns for memory access
  - EAT = 0.99 x 160 + 0.02 x 300 = 162.8ns

Effective Access Time

Effective Access Time [EAT]=

[(Hit)(TLB access time + memory access time)

+ (1-Hit)(TLB access + PT access + memory access)]

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**) to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process.
- Any violations result in a trap to the kernel

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

00000

page 0
page 1
page 2
page 3
page 4
page 5

10,468
12,287

frame number
valid–invalid bit

	frame number	valid–invalid bit
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

page table

frame	contents
0	
1	
2	page 0
3	page 1
4	page 2
5	
6	
7	page 3
8	page 4
9	page 5
	• • •
	page *n*

Shared Pages

- An advantage of paging is the possibility of ***sharing*** common code. This is important in a time-sharing environment.

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)

  (**Reentrant** code is non-self-modifying code: it never changes during execution)

  - 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

Shared Pages Example

process P_1
ed 1
ed 2
ed 3
data 1

page table for P_1
3
4
6
1

process P_2
ed 1
ed 2
ed 3
data 2

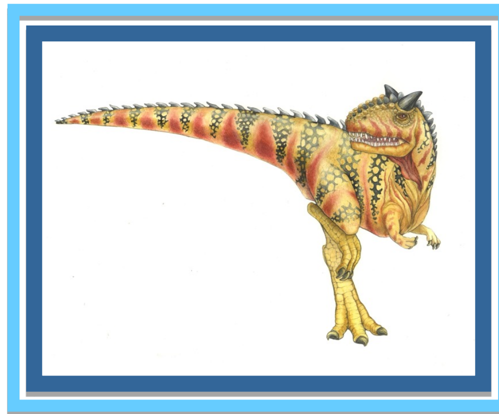
page table for P_2
3
4
6
7

process P_3
ed 1
ed 2
ed 3
data 3

page table for P_3
3
4
6
2

Frame	Contents
0	
1	data 1
2	data 3
3	ed 1
4	ed 2
5	
6	ed 3
7	data 2
8	
9	
10	
11	

End of Chapter 8



Silberschatz, Galvin and Gagne ©2013