CSE3221.3
Operating System Fundamentals

No.9

Memory Management (2)

Prof. Hui Jiang
Dept of Computer Science and Engineering
York University


Memory Management Approaches

· **Contiguous Memory Allocation**

· **Paging**

· **Segmentation**

· **Segmentation with paging**

Contiguous Memory Allocation suffers serious external fragmentation


Paging(1)

- **Logical space is contiguous and consists of pages**
- **Physical space is broken into frames**
- **Page size = Frame size**

· Each page is independently mapped to (or physically supported by) one frame.

· User program sees a contiguous logical space.

· But the supporting frames are scattered in physical memory.

· The mapping is automatically done by hardware or OS based on a **page table**.
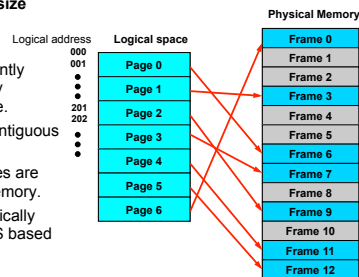
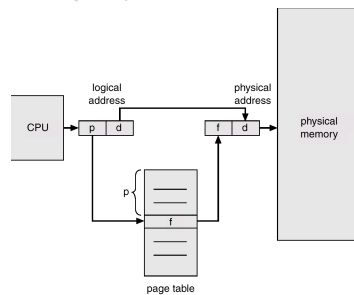## Paging Example(1)
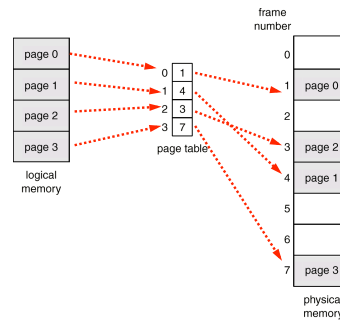
Address Translation Architecture

- **Convert logical address into page # and offset :**
  Logical address (X) = page number (*p*) + page offset (*d*).

- **Assume page size k:**
 $p = X/k$ (quotient).
  $d = X\%k$ (remainder).

· *p* is used to index page table to find frame number or base physical address of this page.

· *d* is the offset in the mapped frame.

- The physical address Y:
  $Y = f * k + d$
  (*f* is frame number).

Translation of logical address (for binary address)

· Page size (frame size) is typical a power of 2. (4k – 16M).
· Logical address is a concatenated bit stream of page number and page offset.
· **An example:** 1) logical space is 2**m: logical address is m bits.
   2) page size is 2**n: page offset is n bits.
   3) a logical space needs at most 2**(m-n) pages:
      page table contains at most 2**(m-n) elements
      page number needs (m-n) bits to index page table

page number	page offset
p	*d*
m-n bits	n bits
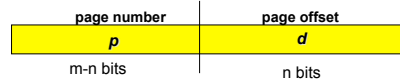
Given a binary logical address, the last n bits is page offset and the first m-n bits is page number.

Paging Example (2)

- Physical memory: 32-byte ($2^{**}5$).
- Logical memory: 16-byte ($2^{**}4$).
- Page size: 4-byte ($2^{**}2$).
- Logical memory needs up to 4 pages: 4 entries in page table.
- m=4, n=2.

Logical address 9 : [1 | 0 | 0 | 1]

Paging Example (2)

- Physical memory: 32-byte ($2^{**}5$).
- Logical memory: 16-byte ($2^{**}4$).
- Page size: 4-byte ($2^{**}2$).
- Logical memory needs up to 4 pages: 4 entries in page table.
- m=4, n=2.

Logical address 9 : [1 | 0 | 0 | 1]

Physical address 5 : [0 | 0 | 1 | 0 | 1]

Paging Hardware

- OS maintains a page table for every process.
- All page tables are kept in physical memory.
- The currently active page table is page table of the currently running process.
- For small active page-table (<256 entries): using registers
- For large page-table: using two indexing registers
  - *page-table base register (*PTBR) points to the active page table.
  - *page-table length register* (PTLR) indicates size of the active page table.
  - In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.

Paging Hardware: TLB

- **Caching**: using of a special fast-lookup hardware cache called *associative registers* or ***translation look-aside buffers (TLBs)***
  - Associative registers (expensive) – parallel search
  - speedup translation from page # $\rightarrow$ frame # :
    Assume page number is P:
    -- If P is in associative register, get frame # out. (hit)
    -- Otherwise get frame # from page table in memory (miss)
    Save to TLB for next reference, replace an old one if full

Page #	Frame #
P1	F1
P2	F2
P3	F3
......	......

P → → Fx

Paging Hardware with TLB: MMU in Paging

Need to flush TLB's in context switch

Effective Access Time of paging after TLB

- Assume memory cycle time is **a** time unit.
- One TLB Lookup = **b** time unit.
- Hit ratio – percentage of times that a page number is found in the associative registers; ration related to number of associative registers.
- Hit ratio = λ.
- Effective Access Time (EAT):

$$EAT = (a + b) \lambda + (2a + b)(1 - \lambda)$$
$$= (2 - \lambda)a + b$$

Example: a = 100 nanoseconds, **b** = 20 nanosecond.

If λ = 0.80, EAT = 140 nanoseconds (40% slower).
If λ = 0.98, EAT = 122 nanoseconds (22% slower).

Paging (2)

- **No external fragmentation in paging.**
- **Internal fragmentation: process size does not happen to fall on page boundaries.**
  - **Average one-half page per process.**
- **How to choose page size:**
  - **Smaller page size:**
    - **less internal fragmentation.**
    - **large page table (more overhead).**
  - **Typical 4K—8KB**
- **If each page table entry is 4 bytes long, it can point to one of 2^{32} frames**
  - **Maximal physical address: frame size * (2^{32})**
  - *(from this we can deduce bit number in physical address)*

Paging (3): Memory Allocation

- OS keeps track of all free frames.
- To run a program of size n pages, OS needs to find *n* free frames and load program.
- OS sets up a page table to translate logical to physical addresses.
- Each process has its page table and saved in memory pointed by its PCB.
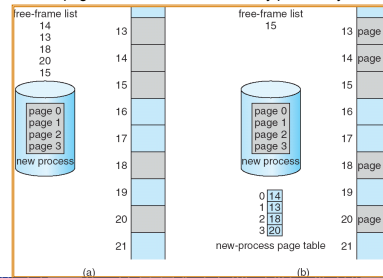

OS data structure for Paging

- OS maintain a page table for each process in memory, pointed by PCB of this process.
  - Used to translate logical address in a process' address space into physical address.
  - Example: one process make an I/O system call and provide an address as parameter (logical address in user space). OS must use its page-table to produce the correct physical address.

- OS maintains a global **frame table:**
  - One entry for each physical frame in memory.
  - To indicate the frame is free or allocated, if allocated, to which page of which process.

- In context switch, the saved page-table is loaded by CPU dispatch to MMU for every memory reference and flush TLB. (This increases context switch time)
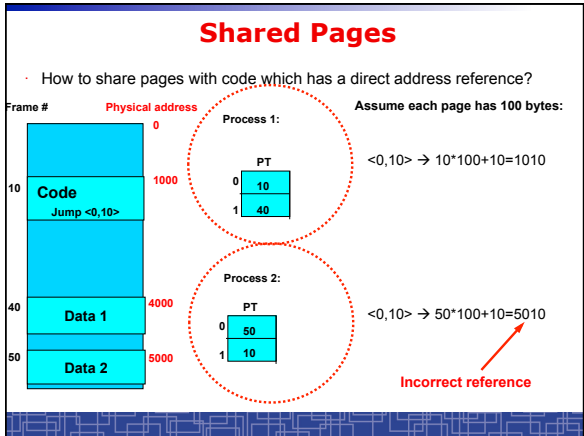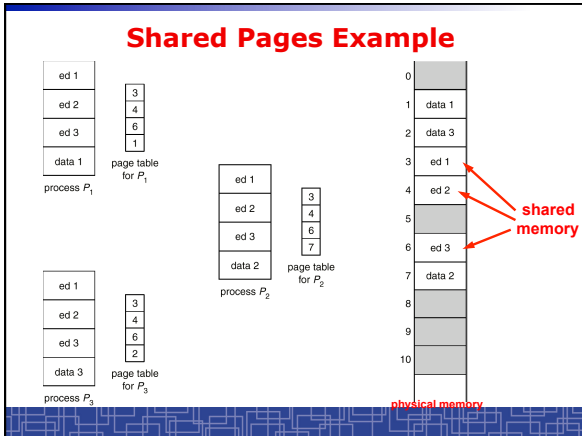
Memory Protection in paging

- How is memory protected from different processes?
  - In paging, other process memory space is protected automatically.
- Memory protection can be implemented by associating protection bits with each frame in page table
  - One bit for read-only or read-write
  - One bit for execute-only
  - One *Valid-invalid* bit
    - "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.
    - Use page-table length register (PTLR): to indicate the size of page table
    - *Valid-invalid* bit is mainly used for virtual memory
- In every memory reference, the protection bits are checked. Any invalid access will cause a trap into OS.

Example:

--14-bit address
-- page size 2KB
-- valid space
0-16383 (2^{14})
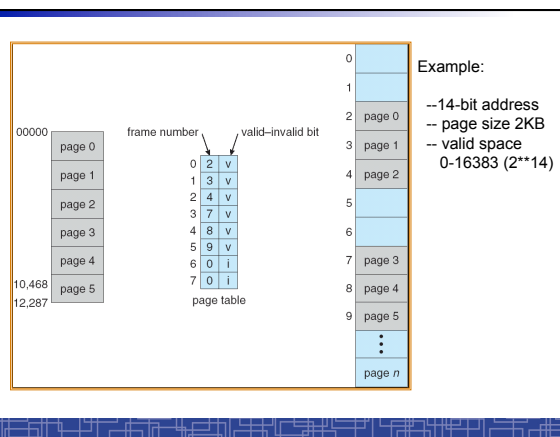
Sharing Memory in Paging

- Different pages of several processes can be mapped to the same frame to let them share memory.
- Shared-memory for inter-process communication.
- 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 code:
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes (i.e. same locations in the page tables).

Shared Pages Example

ed 1 / ed 2 / ed 3 / data 1 — page table for P_1 (3, 4, 6, 1) — process P_1

ed 1 / ed 2 / ed 3 / data 2 — page table for P_2 (3, 4, 6, 7) — process P_2

ed 1 / ed 2 / ed 3 / data 3 — page table for P_3 (3, 4, 6, 2) — process P_3

physical memory: 0 / 1 data 1 / 2 data 3 / 3 ed 1 / 4 ed 2 / 5 / 6 ed 3 / 7 data 2 / 8 / 9 / 10

shared memory

Shared Pages

- How to share pages with code which has a direct address reference?

Frame # Physical address Assume each page has 100 bytes:

Process 1:
PT: 0 → 10, 1 → 40

$<0,10> \rightarrow 10*100+10=1010$

Process 2:
PT: 0 → 50, 1 → 10

$<0,10> \rightarrow 50*100+10=5010$

Incorrect reference

10 Code Jump <0,10> 1000
40 Data 1 4000
50 Data 2 5000

Copy-on-Write

- **For quick process Creation: *fork()***
- **Traditionally, fork() copies parent's address space for the child.**
- ***Copy-on-Write*: without copying, the parent and child process initially share the same pages, and these pages are marked as copy-on-write.**
  - **If either process needs to write to a shared page, a copy of the shared page is created and stop sharing this page.**
- **Advantages of copy-on-write:**
  - **Quick process creation (no copying, just modify page table for page sharing)**
  - **Eventually, only modified pages are copied. All non-modified pages are still shared by the parent and child processes.**
    - **Better memory utilization**

Copy-on-Write

process₁ — physical memory — process₂
page A
page B
page C

process₁ — physical memory — process₂
page A
page B
page C
Copy of C

Hierarchical Paging (multilevel paging)

- **In modern computer, we require a large logical-address space, which results in some huge page table.**

- **No contiguous memory space for the large page table.**

- **Hierarchical paging: using paging technique to divide the large page table into smaller pieces**

outer page table — page of page table — page table — memory

Address-Translation in two-level paging

- Logical address 32-bit, page size 4K, maximal physical address 2**32 frames
- A logical address is divided into 20 bits page number and 12 bits page offset.
- Since page-table is paged, the logical address is as follows:

page number		page offset
p_i	p_2	d
10	10	12

where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table.

logical address: p₁ p₂ d

4-byte — outer-page table
4-byte — page of page table
1-byte — Physical address

Multilevel Paging and Performance

- 64-bit logical address may require 7-level paging.
- Since each level is stored as a separate table in memory, converting a logical address to a physical one may take seven memory accesses.
- TBL-based caching permits performance to remain reasonable.
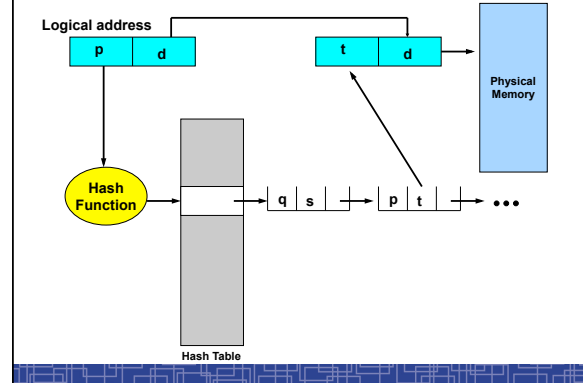- Cache hit rate of 98 percent yields:

  effective access time = 0.98 x 120 + 0.02 x 820

  $\qquad\qquad\qquad$ = 134 nanoseconds.

  which is only 34 percent slowdown in memory access time.
- But the overhead is too high to maintain many page-tables
- In 64-bit Linux, it uses 4-level paging to page 48-bit address.

## Hashed Page Tables

---

Inverted Page Table

- One entry for each real frame of memory.

- Each entry consists of the virtual page number stored in this frame, with information about the process that owns that page.

- Only one table in the system: decreases memory needed to store page tables.

- 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.
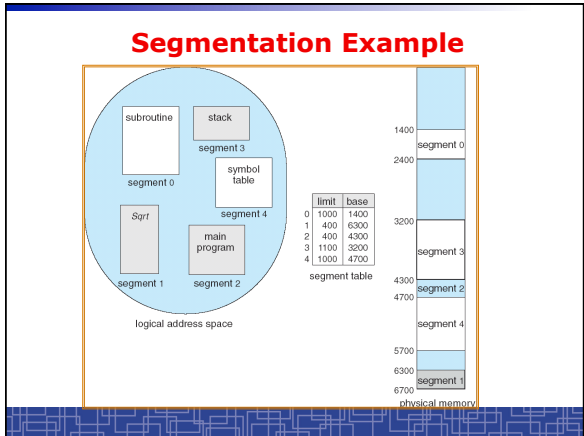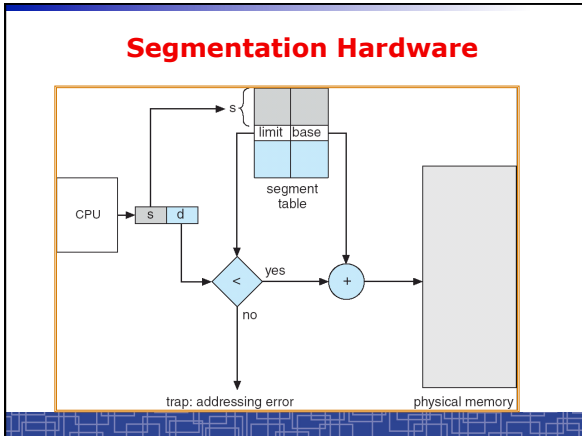 – To speedup further, TLB is used.

Inverted Page Table Architecture
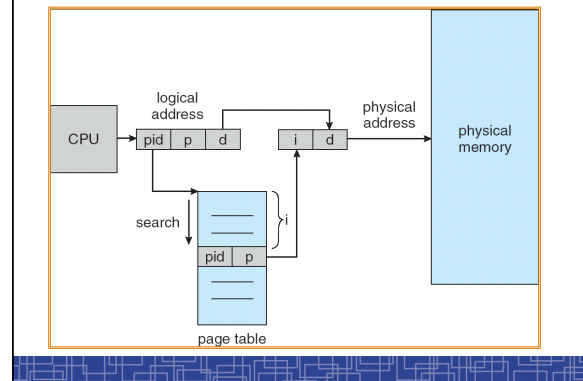
Another view of logical space

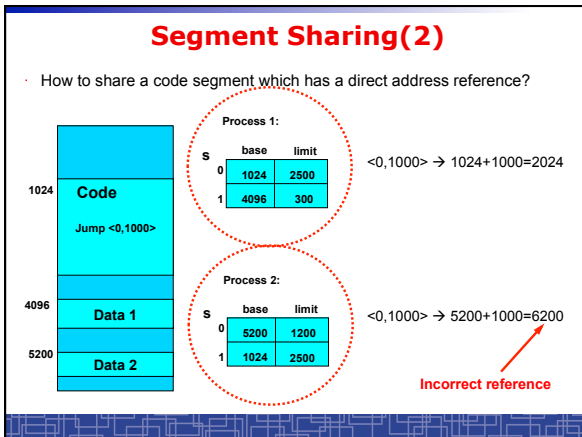logical memory space   physical memory space

---
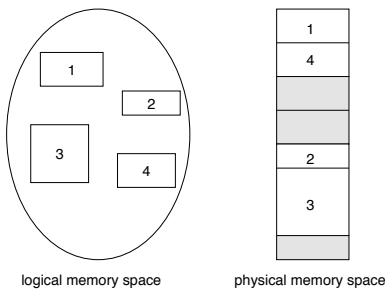
Segmentation

- A logical-address space is a collection of segments.
- Each segment has a name (segment number) and a length.
- A logical address consists of: a segment number and an offset.

  <segment-number (**s**), offset (**d**)>
- Physical memory address is still one-dimension linear array.
- MMU Translates logical address (2-D) into physical address (1-D) based on a **segment table**.
  - Including all segments in the program.
  - Each entry has a segment base and a segment limit.
  - An array of base-limit pairs.
- The active segment table is pointed by two CPU registers for MMU:
  – *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; *segment number s is legal if s < STLR*.

Segmentation Hardware

CPU → s | d

segment table
limit | base

< yes +

no

trap: addressing error

physical memory

Segmentation Example

subroutine | stack

segment 3

segment 0

symbol table

Sqrt | segment 4

main program

segment 1 | segment 2

logical address space

	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

1400 segment 0
2400
3200 segment 3
4300 segment 2
4700 segment 4
5700
6300 segment 1
6700

physical memory

Memory Protection in Segmentation

- Limit check for every memory reference.

- Each segment is a semantically defined portion of data. They tend to be used in the same way. We can define protection bits for every segment:
  - Read-only, read-write, and so on.
  - The segment hardware will check protection bits for each memory reference.

Segment Sharing(1)

- Every process has a segment table. Segments are shared when the entries point to the same physical location.

- Sharing has to be at the segment level.

editor
segment 0
data 1
segment 1
logical memory process P_1

	limit	base
0	25286	43062
1	4425	68348

segment table process P_1

editor
segment 0
data 2
segment 1
logical memory process P_2

	limit	base
0	25286	43062
1	8850	90003

segment table process P_2

43062 editor
68348 data 1
72773
90003 data 2
98553
physical memory

Segment Sharing(2)

- How to share a code segment which has a direct address reference?

1024 Code
Jump <0,1000>

4096 Data 1

5200 Data 2

Process 1:

s	base	limit
0	1024	2500
1	4096	300

<0,1000> → 1024+1000=2024

Process 2:

s	base	limit
0	5200	1200
1	1024	2500

<0,1000> → 5200+1000=6200

Incorrect reference

Segment Sharing(3)

- If processes share a code segment with the direct address reference, all processes should have the same segment number for this segment.

- The following segments can be shared freely:
  - Read-only data segment.
  - Code segment with only indirect address reference (by offset from the current position or segment beginning).
  - Code segment with address relative to a register which contains the current segment number.

6

Fragmentation in Segmentation

· No internal fragmentation.

· External fragmentation:
  – Since segments have various size.
  – Dynamic storage-allocation problem.
  – Best-fit, first-fit, worst-fit.
  – External fragmentation depends on average segment size.
  – If the average segment size is small, external fragmentation will also be small.
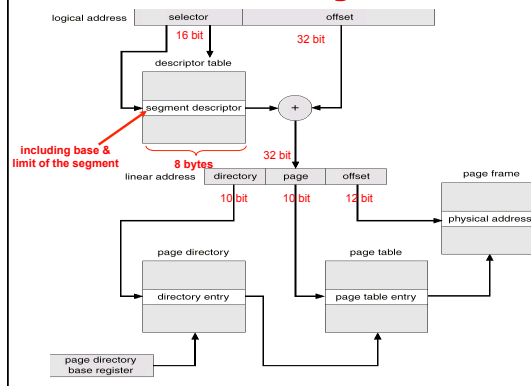
Segmentation with Paging

· Both segmentation and paging have advantages and disadvantages. We can combine them to improve on each.

· Two most popular CPU's, Motorola 68000 line and Intel 80x86 and Pentium uses a mixture of paging and segmentation.

· Example: Intel Pentium uses segmentation with paging for memory management.
  – Based on segmentation primarily.
 – The varying-length segments are paged into a set of fixed-sized pages.

Intel Pentium addressing

· A process can have up to 16K (2**14) segments, divided into two segment tables:
  – Local descriptor table (LDT)
  – Global descriptor table (GDT)
  – Each entry in the tables is 8 bytes (base+length+others).
· Each segment can be 4GB (2**32) in maximum.
· A logical address is 48 bits, consists of:
  – 16 bits selector: 13-bit segment number, 1-bit indicate LDT or GDT, 2-bit for protection.
  – 32 bits segment offset: a segment can be up to 2**32 bytes
  – Each segment is paged: page size 4KB & 2-level paging: 10-bit page directory # +10-bit page # + 12-bit page offset
· CPU has six segment registers (caches), allowing 6 segments to be addressed at any time (avoid reading descriptor for each memory reference).
· In Pentium, physical address is 32-bit (max 4GB).

Pentium Addressing Architecture

Pentium Addressing Architecture
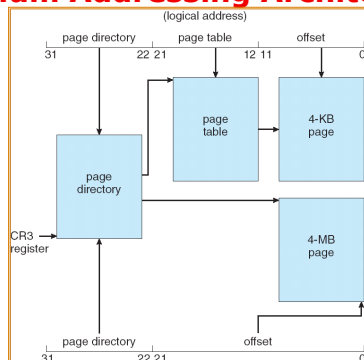
Comparing Memory-Management Strategies

(1)Contiguous allocation, (2)paging, (3)segmentation, (4)Segmentation with paging

· Hardware support
· Performance
· Fragmentation
· Relocation
· Swapping
· Sharing
· Protection