SNS COLLEGE OF TECHNOLOGY

Coimbatore-35
An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with ‘A++’ Grade

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

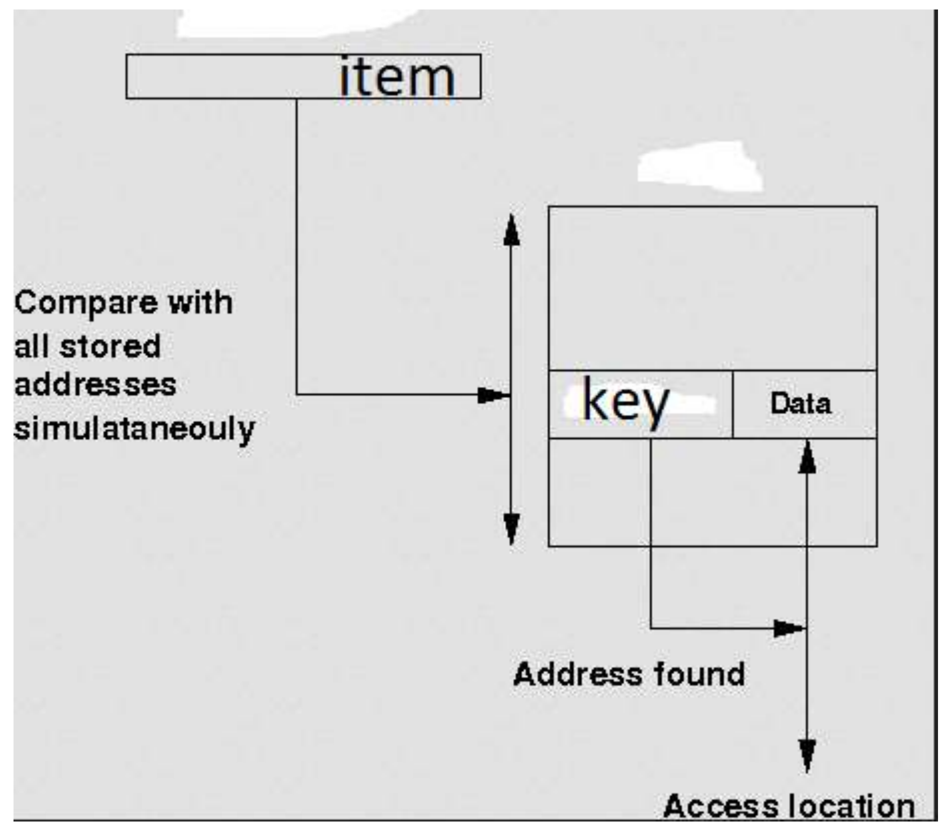
19ECT312 – EMBEDDED SYSTEM DESIGN

III YEAR/ VI SEMESTER

UNIT 4 : EMBEDDED OPERATING SYSTEM AND MODELING

TOPIC 4.4 : MEMORY MANAGEMENT

Associative memory



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

Implementation of Page Table

- For each process, 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)**
- 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 (LRU)
  - Some entries can be **wired down** for permanent fast access
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process (PID) to provide address-space protection for that process
  - Otherwise need to flush at every context switch

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

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 size of TLB

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

Effective Access Time (EAT)

$$EAT = (100 + \varepsilon)\,\alpha + (200 + \varepsilon)(1 - \alpha)$$

Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 120 + 0.20 x 220 = 140ns

Consider better hit ratio -> α = 98%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.98 x 120 + 0.02 x 220 = 122ns

Memory Protection

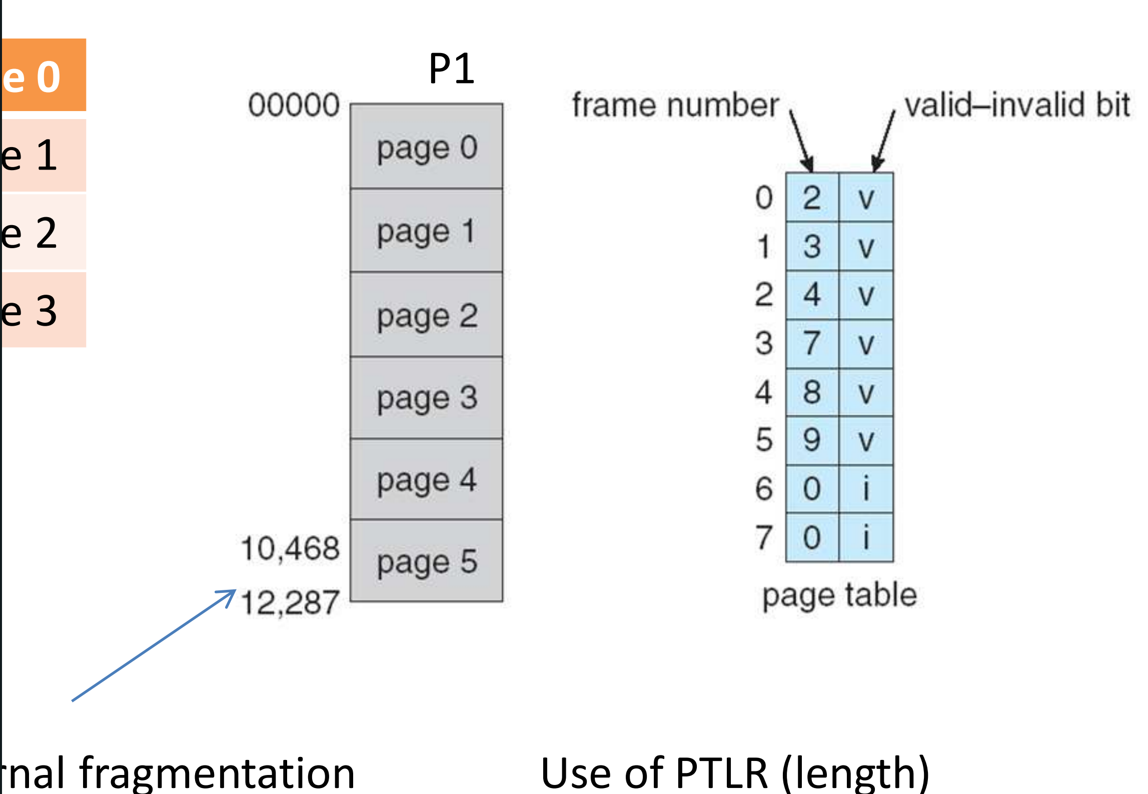
- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated 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 PTLR
- Any violations result in a trap to the kernel

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

14 bit address space (0 to 16383)

Page size 2KB

Process P1 uses only 0 to 10468



Use of PTLR (length)

Shared Pages Example

- System with 40 users
  - Use common text editor
- Text editor contains 150KB code 50KB data (page size 50KB)
  - 8000KB!
- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
    - Code never changes during execution
- Only one copy of the editor in the memory
- Total memory consumption
  - 40*50+150=2150KB

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	

Data share: example

writer.c

```
int main()
{

      int shmid,f,key=3,i,pid;
      char *ptr;


shmid=shmget((key_t)key,100,IPC_CREAT|066
6);
      ptr=shmat(shmid,NULL,0);
      printf("shmid=%d ptr=%u\n",shmid, ptr);
      strcpy(ptr,"hello");
     i=shmdt((char*)ptr);

}
```

reader .c

```
int main()
{

      int shmid,f,key=3,i,pid;
      char *ptr;

      shmid=shmget((key_t)key,100,IPC_CREAT|0666);
      ptr=shmat(shmid,NULL,0);
      printf("shmid=%d ptr=%u\n",shmid, ptr);
      printf("\nstr %s\n",ptr);
}
```

ptr

Shared memory

Structure of the Page Table

- Memory requirement for page table 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^{20} (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
    - Don't want to allocate that contiguously in main memory

- Hierarchical Paging

- Hashed Page Tables

- Inverted Page Tables

Hierarchical Page Tables

- Break up the page table into multiple pages
- We then page the page table
- A simple technique is a two-level page table

Two-Level Page-Table Scheme

outer page table

page of page table

page table

memory

1
500
100
708
929
900

0
1
100
500
708
900
929

Two-Level Paging Example

A logical address (on 32-bit machine with 4KB page size) is divided into:

– a page number consisting of 20 bits

– a page offset consisting of 12 bits

Since the page table is paged, the page number is further divided into:

– a 10-bit page number

– a 10-bit page offset

Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

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

Two-Level Page-Table Scheme

Each divided page table size=2^{10} *4bytes=4KB =Page size

p_1

outer page table

p_2

1
500
100
708
929
900

page of page table

page table

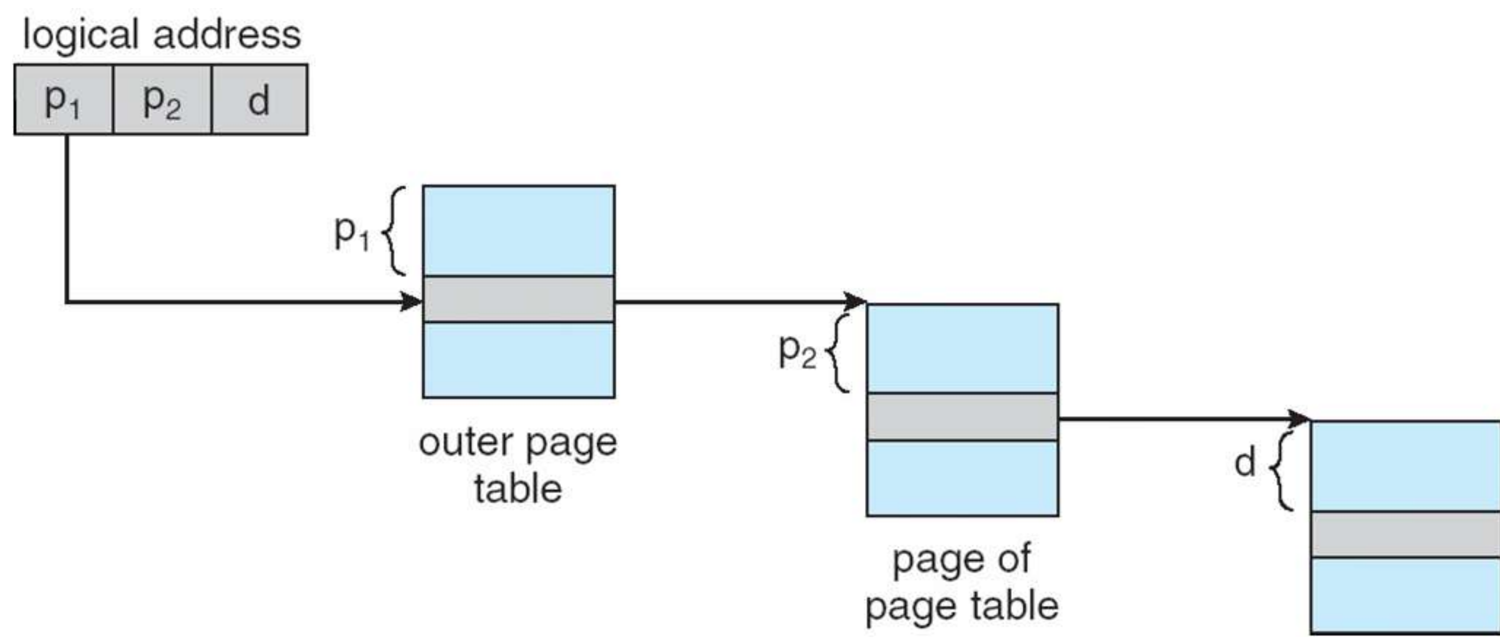
0
1
100
500
708
900
929

memory

$\downarrow$ d

ntium II

Address-Translation Scheme



ntium II

64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- 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	inner page	page offset
p_1	p_2	d
42	10	12

  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2nd outer page table
  - But in the following example the 2nd outer page table is still 2^{34} bytes in size
    - And possibly 4 memory access to get to one physical memory location

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

PARC (32 bits), Motorola 68030 support three and four level paging respectively

Hashed Page Tables

- Common in virtual address spaces > 32 bits
- The 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 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

Hashed Page Table

logical address
p | d
physical address
r | d
hash function
q | s
p | r
• • •
hash table
physical memory

Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all frames
- One entry for each frame
- Entry consists the page number stored in that frame, 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

Inverted Page Table Architecture

64 bit UltraSPARC, PowerPC,

CPU

logical address

pid | p | d

i | d

physical address

physical memory

search

i

pid | p

Address space ID

page table

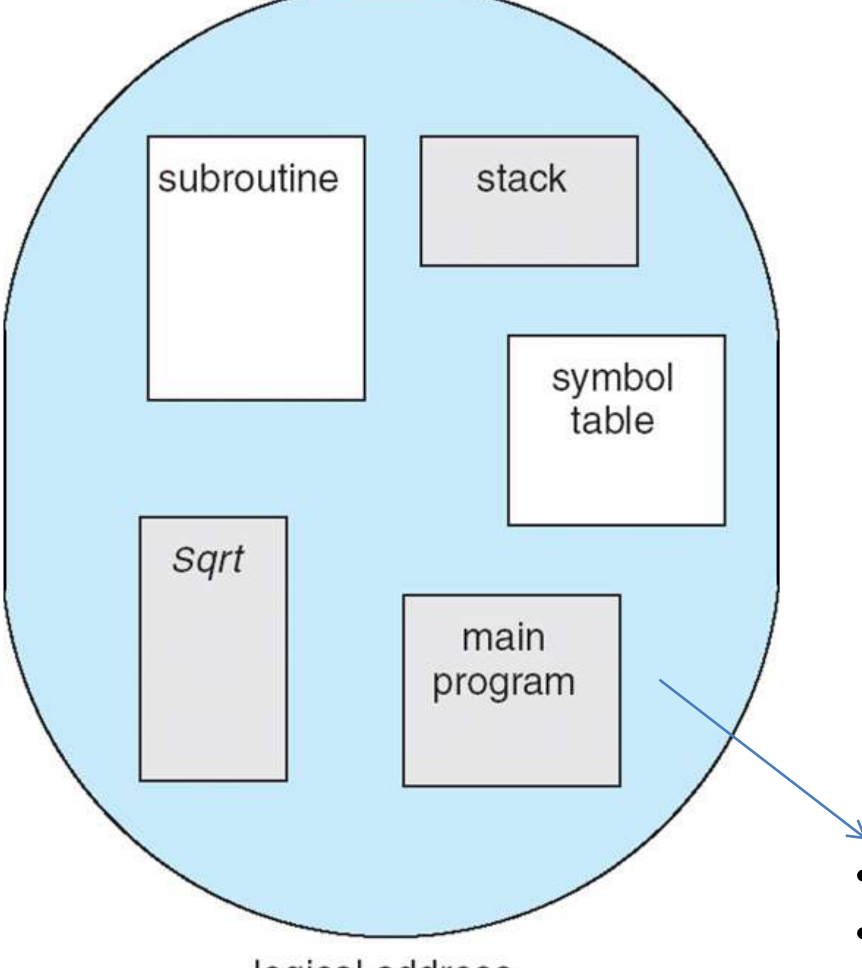
Segmentation

- 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

Compiler generates the segments
Loader assign the seg#

User’s View of a Program



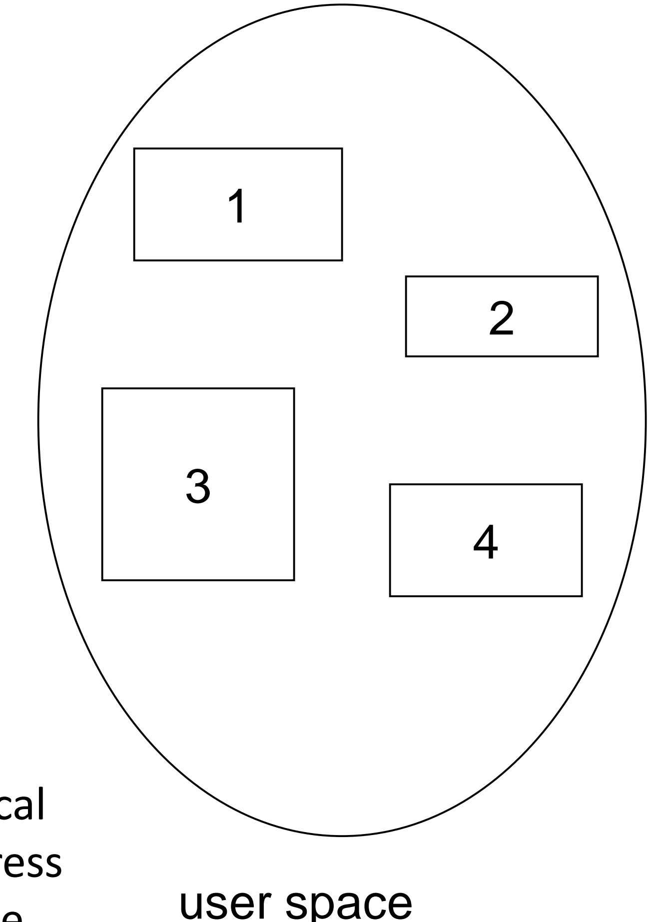
User specifies each address by two quantities
(a) Segment name
(b) Segment offset

Logical address contains the tuple
<segment#, offset>

- Variable size segments without order
- Length=> purpose of the program
- Elements are identified by offset

Logical View of Segmentation

Logical Address <segment-number, offset>



Logical address space

1
4
2
3

physical memory space

Long term scheduler finds and allocates memory for all segments of a program
Variable size partition scheme

Windows XP Memory Usage

Segment	First Address	Last Address	Size
Code	401000x	403000x	002000x ~ 8 Kbytes
Static (Global) Data	403000x	703000x	300000x ~ 3 megabytes
Heap	760000x	3A261000x	39800000x ~ 950 megabytes
Stack	22EF00x	16EF00x	1C0000x ~ 2 megabyte

LINUX Memory Usage			
Segment	**First Address**	**Last Address**	**Size**
Code	8048400x	8049900x	001500x ~ 6 Kbytes
Static (Global) Data	8049A00x	8349A00	300000x ~ 3 megabytes
Heap	B7EE,B000x	01CE,4000x	B6000000x ~ 3 gigabytes
Stack	BFFB,7334x	29BA,91E0x	9640,0000x ~ 2.5 gigabyte

Memory image

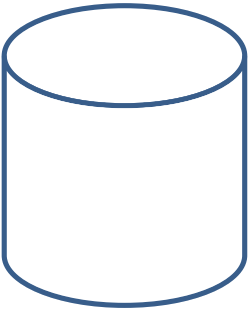
```
0x08048368 <main+0>:    55                  push   %ebp
0x08048369 <main+1>:    89 e5               mov    %esp,%ebp
0x0804836b <main+3>:    83 ec 08            sub    $0x8,%esp
0x0804836e <main+6>:    83 e4 f0            and    $0xfffffff0,%esp
0x08048371 <main+9>:    b8 00 00 00 00      mov    $0x0,%eax
0x08048376 <main+14>:   83 c0 0f            add    $0xf,%eax
0x08048379 <main+17>:   83 c0 0f            add    $0xf,%eax
0x0804837c <main+20>:   c1 e8 04            shr    $0x4,%eax
0x0804837f <main+23>:   c1 e0 04            shl    $0x4,%eax
0x08048382 <main+26>:   29 c4               sub    %eax,%esp
0x08048384 <main+28>:   83 ec 0c            sub    $0xc,%esp
0x08048387 <main+31>:   68 c0 84 04 08      push   $0x80484c0
0x0804838c <main+36>:   e8 1f ff ff ff      call   0x80482b0
0x08048391 <main+41>:   83 c4 10            add    $0x10,%esp
0x08048394 <main+44>:   e8 02 00 00 00      call   0x804839b <b>
```

```
1   void b();
2   void c();
3   int  main( )
4   {
5       printf( "Hello from main\n");
6       b();
7   }
8   // This routine reads the opcodes from memory and prints them out.
9   void  b()
10  {
11      char *moving;
12
13      for ( moving = (char *)(&main); moving < (char *)(&c); moving++ )
14          printf( "Addr = 0x%x, Value = %2x\n", (int)(moving), 255 & (int)*moving )
15  }
16  void  c()
17  {
18  }
```

```
0x0804839b <b+0>:     55                    push    %ebp
0x0804839c <b+1>:     89 e5                 mov     %esp,%ebp
0x0804839e <b+3>:     83 ec 08              sub     $0x8,%esp
0x080483a1 <b+6>:     c7 45 fc 68 83 04 08  movl    $0x8048368,0xfffffffc(%ebp)
0x080483a8 <b+13>:    81 7d fc d9 83 04 08  cmpl    $0x80483d9,0xfffffffc(%ebp)
0x080483af <b+20>:    73 26                 jae     0x80483d7 <b+60>
0x080483b1 <b+22>:    83 ec 04              sub     $0x4,%esp
0x080483b4 <b+25>:    8b 45 fc              mov     0xfffffffc(%ebp),%eax
0x080483b7 <b+28>:    0f be 00              movsbl  (%eax),%eax
0x080483ba <b+31>:    25 ff 00 00 00        and     $0xff,%eax
```

ecutable file and virtual addr

out



Symbol table	
Name	address
SQR	0
SUM	4

Paging view

0	Load	0
4	ADD	4

Segmentation view

<CODE, 0>	Load	<ST,0>
<CODE, 2>	ADD	<ST,4>

Virtual address space

Segmentation Architecture

- Logical address consists of a two tuple:

  $<$segment-number, offset$>$

- **Segment table** – maps two-dimensional logical address to physical address;
- 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;

  segment number ***s*** is legal if ***s*** < **STLR**

Example of Segmentation

subroutine
segment 0
stack
segment 3
symbol table
segment 4
Sqrt
segment 1
main program
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

Segmentation Hardware

s
limit
base
segment table
CPU
s
d
<
yes
+
no
trap: addressing error
physical memory

Example of Segmentation

subroutine

segment 0

stack

segment 3

symbol table

segment 4

Sqrt

segment 1

main program

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

Segmentation Architecture

- Protection
- Protection bits associated with segments
  - With each entry in segment table associate:
    - validation bit = 0 $\Rightarrow$ illegal segment
    - read/write/execute privileges
- Code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
  - Long term scheduler
  - First fit, best fit etc
- Fragmentation

Segmentation with Paging

Key idea:

Segments are splitted into multiple pages

Each page is loaded into frames in the memory

Segmentation with Paging

- Supports segmentation with paging
  - Each segment can be 4 GB
  - Up to 16 K segments per process
  - <selector(16), offset (32)>
  - Divided into two partitions
    - First partition of up to 8 K segments are private to process (kept in **local descriptor table LDT**)
    - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table GDT**)

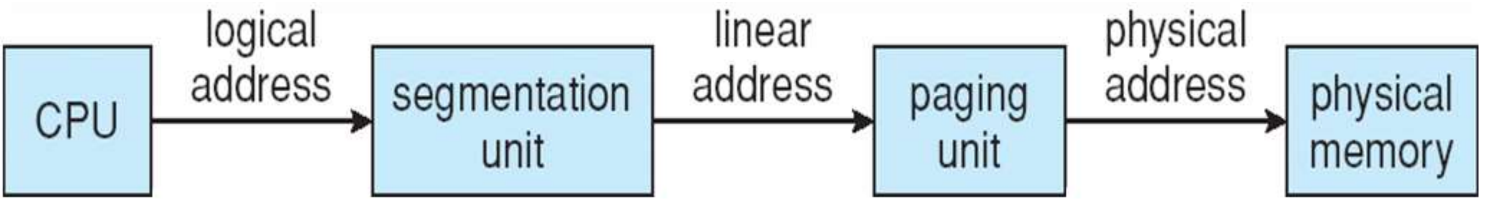
S(13)	G(1)	P(2)

- CPU generates logical address (six Segment Reg.)
  - Given to segmentation unit
    - Which produces linear addresses
  - Physical address 32 bits
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB

Intel 80386

IBM OS/2

Logical to Physical Address Translation in Pentium



Page table=2^{20} entries

page number		page offset
p_1	p_2	d
10	10	12

Example: The Intel Pentium

logical address
selector
offset

Segment register

descriptor table
← 8 bytes →
segment descriptor
+

linear address
directory
page
offset

page frame
physical address

page directory
directory entry

page table
page table entry

page directory
base register

Intel Pentium Segmentation

logical address | selector | offset

descriptor table

segment descriptor

+

32-bit linear address

Pentium Paging Architecture

(logical address)

page directory | page table | offset

31 22 21 12 11 0

page table

4-KB page

page directory

4-MB page

CR3 register

page directory | offset

31 22 21 0