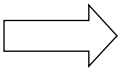
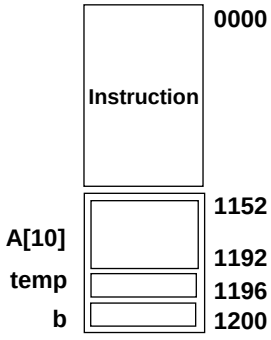
Memory Management Concepts

- **Address Binding**
  - ♦ **symbolic address -> relocatable address -> absolute address**
  - ♦ **Compile time, loadingtime vs. execution time.**
- **Dynamic Loading**
  - ♦ **Subroutine is linked during compilation but loaded during execution.**
- **Dynamic Linking**
  - ♦ **Subroutine is linked during execution**
- **Overlays**
  - ♦ **Example: A Two Pass Compiler:**
    - » **Symbol Table 20K, Common Routines 30K, Overlay Driver 10 K, Pass 1 70 K, pass 2 80K**

Memory Image of a Program

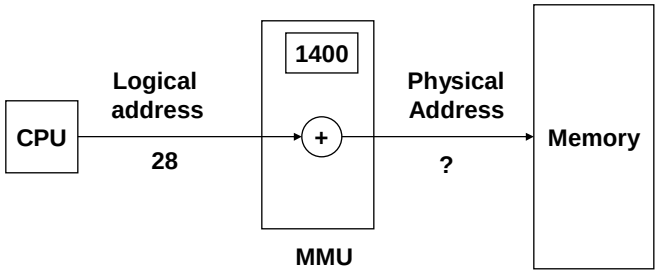
```
Main()
{
    int A[10];
    int temp;
    int b;
    temp=A[0];
    for (b=0;b<9;b++) {
            a[b]=a[b+1]
    }
    a[9]=temp;
}
```





More Concepts

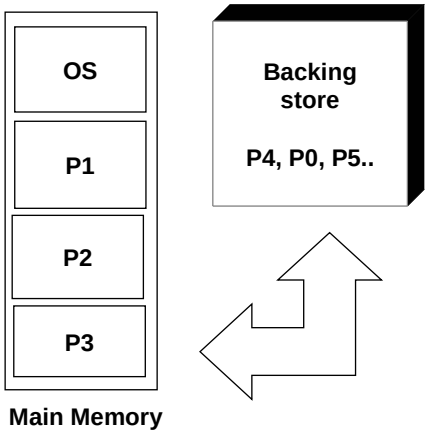
- **Logical vs. Physical Address Space**
  - ♦ For compile time and load time binding: logical=physical
  - ♦ For execution time binding: logical /= physical
  - ♦ 80x86 processors have 4 relocation registers



•**Security by adding limit register**

More Concepts

- **Swapping**
  - ♦ process roll-in and roll-out
  - ♦ backing store
  - ♦ cost factor: 100K process backed on 1MBps backing store?
  - ♦ MMU should know the exact memory usage to save (system calls)
  - ♦ What if there is pending I/O



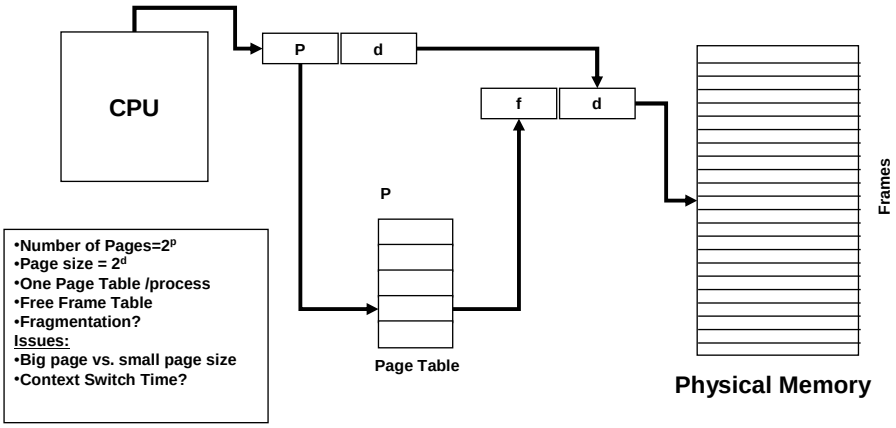
Memory Allocation

- **OS Resident+User process**
- **OS Resident Placement**
  - ♦ **Top or low?**
  - ♦ **What if some part of OS (drivers) are added or deleted?**
- **Partition among Processes**
  - ♦ **Allocation algorithms**
    - » **First-fit**
    - » **best-fit**
    - » **Worst-Fit**
  - ♦ **Problems**
    - » **External Fragmentation**
    - » **Internal Fragmentation**
  - ♦ **Relocation**

OS	400K	resident
Process	Memory	Time
P1	600K	10
P2	1000K	5
P3	300K	20
P4	700K	8
P5	500K	15

What if we use FCFS and total memory is 2560K?

Page Table



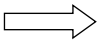
- Number of Pages=2^p
- Page size = 2^d
- One Page Table /process
- Free Frame Table
- Fragmentation?

Issues:

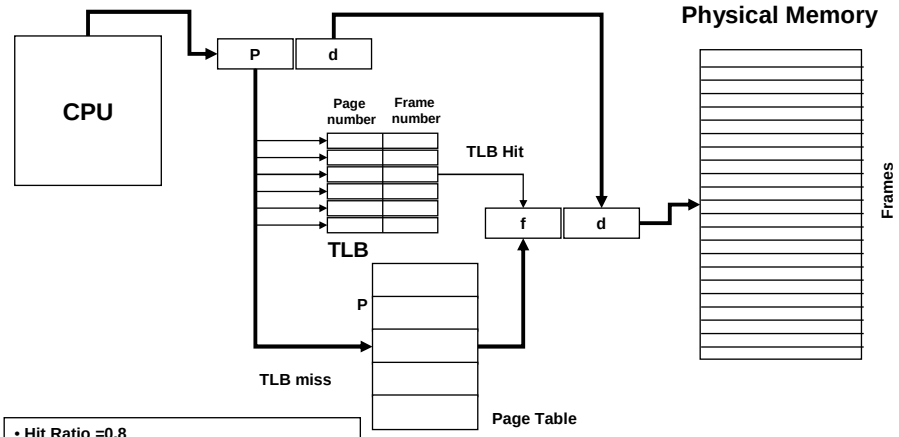
- Big page vs. small page size
- Context Switch Time?

Implementation of Page Table

- **Hardware Register Based:**
  - ♦ **Keeps everything in high speed registers.**
  - ♦ **Context switch = load/unload the entire table.**
  - ♦ **Good for small table size (~256 entry).**
- **Memory Based:**
  - ♦ **Keeps everything in memory**
  - ♦ **Use a page-table-base-register (PTBS) to point in memory**
  - ♦ **Large tables can be kept.**
  - ♦ **Context switch = load the PTBS.**
  - ♦ **Memory reference= 2 memory access.**
- **Cache Mediated:**



Translation-Look ahead Buffer Page Table

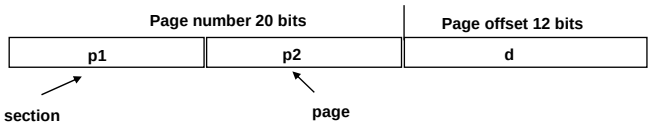


- **Hit Ratio =0.8**
- **Memory access time=100 ns**
- **Associative Search Time=20ns**
- **Avg access time = .8x120+.2x220=140ns**

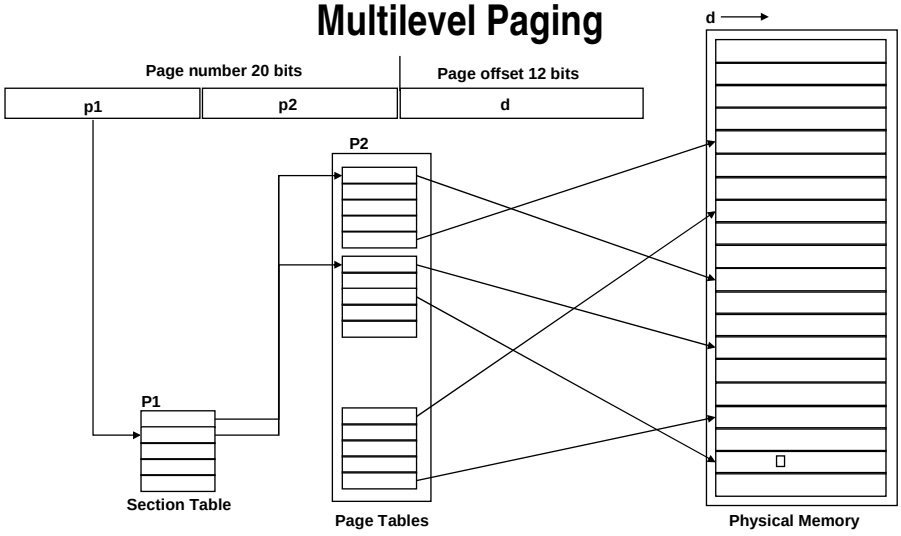
Size of Page Table

Page Table Size Computation:

Logical Address Space = 32 bits i.e. memory size =2^{32} (it is typically 2^{32} - 2^{64})
If page size = 4K = 2^{12}
# of entries in page table = $2^{32}/2^{12}$ = 1 million entry = 2^{20}
Each entry = 4 bytes
Page Table size= 4 MB

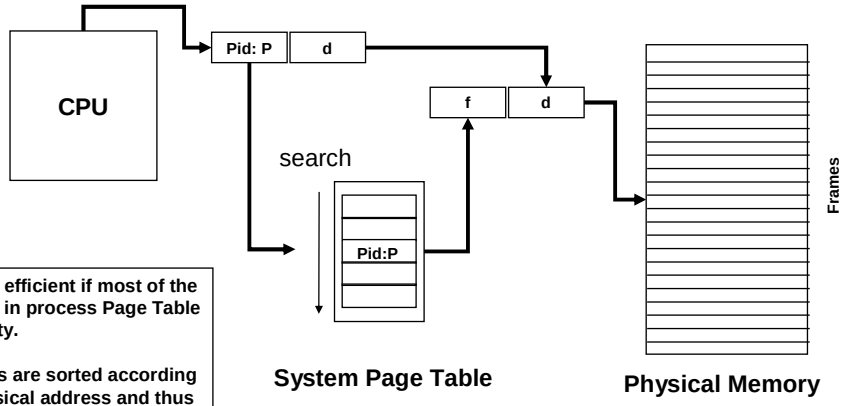


Multilevel Paging



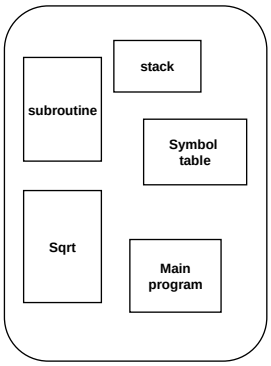
- **Hit Ratio =0.98**
- **Memory access time=100 ns**
- **Associative Search Time=20ns**
- **Number of Levels 2 Avg access time = .98x120+.02x320=124 ns**
- **32-bit Motorola 68030 has 4 level paging. Avg access time?**

Inverted Page Table



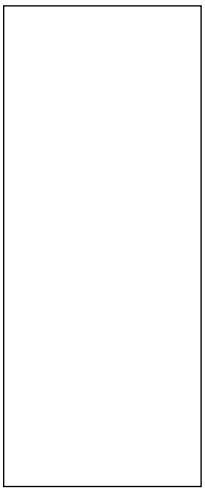
- Space efficient if most of the entries in process Page Table is empty.
- Entries are sorted according to physical address and thus may need searching the entire table.
- Hash Table implementation can help.

Program View: Memory Segments



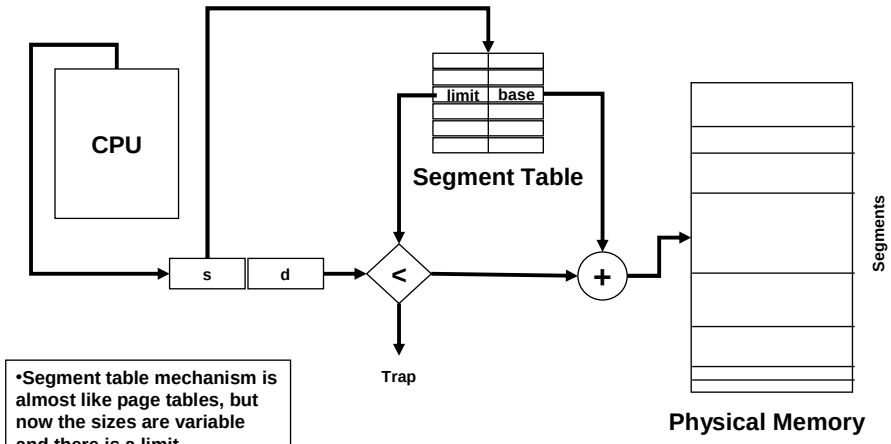
	limit	base
0	1000	
1	400	
2	400	
3	1100	
4	1000	

Segment Table



Physical Memory

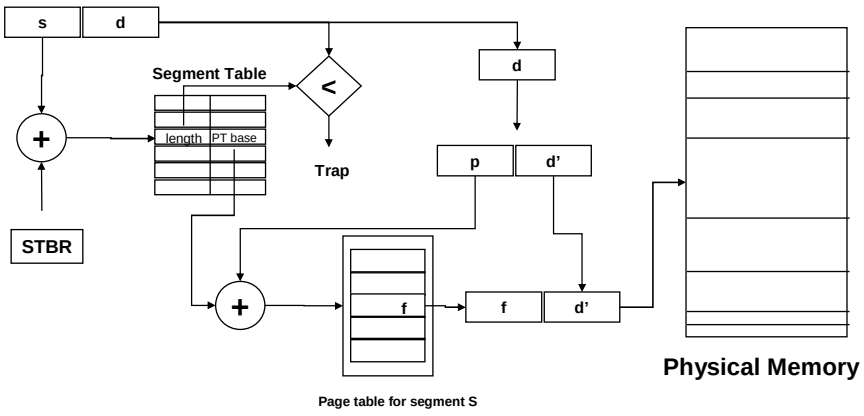
Segment Table

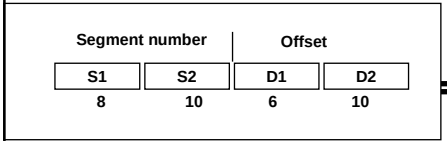


- Segment table mechanism is almost like page tables, but now the sizes are variable and there is a limit.
- One single process may have multiple segments

Os-slide#13

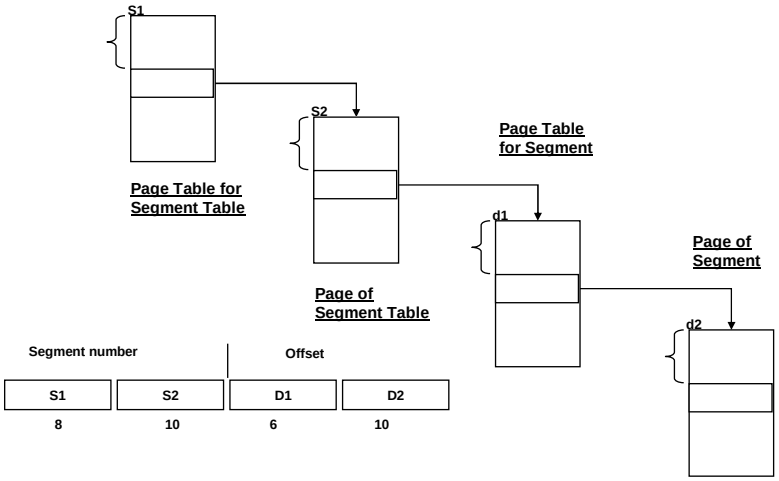
Example: MULTICS Pages Segmentation





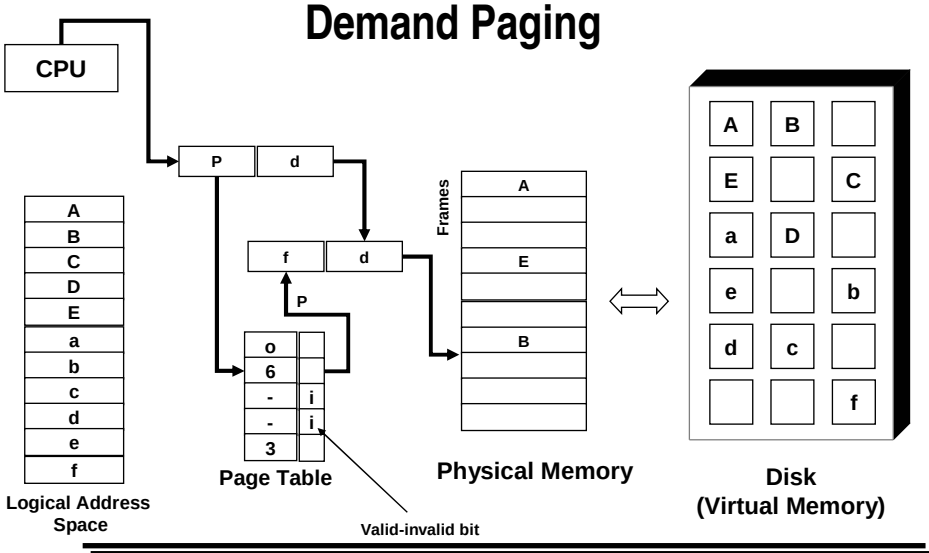
Os-slide#14

Example: MULTICS Addressing Scheme



Os-slide#15

Demand Paging



Os-slide#16

Performance of Demand Paging

Steps on Page fault:

- **Service Page Fault Interrupts**
  - ♦ trap->save registers->determine victim ->..
- **Write back the Victim**
  - ♦ wait in queue->device seek->begin transfer
- **Read in the Page**
  - ♦ wait in queue->device seek->begin transfer
- **Restart the Process**
  - ♦ wait for CPU perhaps allocated to others->restore registers, process table, page table..

Steps 1 and 4 = 1-100 microsec
Steps 2 and 3 = 25 milliseconds
What is the effective access time if hit ratio = p?

Page Replacement Algorithms

Memory Access Sequence:

732 090 103 211 024 343 031 452 201 364 023 321 201 132 248 094 108 795 022 156

Page Access Sequence (assuming page size=100):

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

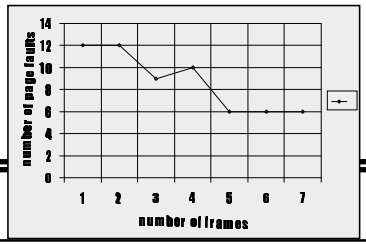
LRU Replacement Policy:

7	7	7	2		2	2	4	4	4	0		0	0			7	7	7
	0	0	0		3	3	3	2	2	2		1	1			1	0	0
		1	1		1	0	0	0	3	3		3	2			2	2	1

What if we have page sequence:

1,2,3,4,1,2,5,1,2,3,4,5

Number of Frames = 3, page faults?
Number of Frames = 4, page faults?
Belady's Anamoly!



Page Replacement Algorithms (cont.)

- Optimal Algorithm
  - ♦ victim=which will not be used for the longest period of time.
- LRU Algorithm
  - ♦ victim=least recently used
  - ♦ needs counter implementation
- LRU Approximation Algorithm
  - ♦ uses one reference bit. Initial value=0, set to1 when referenced.
  - ♦ victim=one whose reference bit=0.
- Additional Reference Bit Algorithm
  - ♦ 8 bit right shifting counter
  - ♦ In each epoch set rightmost bit if referenced, and rightshift
  - ♦ victim=page with minimum byte
- Second Chance Algorithm
  - ♦ FIFO with reference bit
  - ♦ victim=First one in line with reference bit =0, if it 1 set it 0.
- Enhanced Second Chance Algorithm
  - ♦ uses reference bit + modify bit

•(0,0) neighter recently used nor modified.

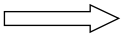
•(0,1) not recently user, but modified.

•(1,0) recently used but clean, but probably will be used soon.

•(1,1) rcently used and modified.

Other Issues

- **Page Buffering:**
  - ♦ **Keep a pool of free frames helps delayed writeout.**
- **Frame Allocation:**
  - ♦ **Minimum required frames per process**
  - ♦ **Equal allocation vs. proportionate allocation**
  - ♦ **Global vs. local replacement**
- **Thrashing (too much page fault):**
  - ♦ **cause: too much multiprogramming.**
  - ♦ **Solution: monitor working set**
  - ♦ **Solution: monitor frequency of page fault**
- **Effect of Program Structure**
- **I/O Interlocking**



What if we run the following program on a ssytem with page size 128 words?

```
Int A[128][128]
for j=0 to 127
     for I=0 to 127
         A[I][j]=0;
```