Virtual memory

Program and data use virtual addresses

Virtual address maps to

Physical address in RAM

OR

Disk address

Process called memory mapping

or address translation

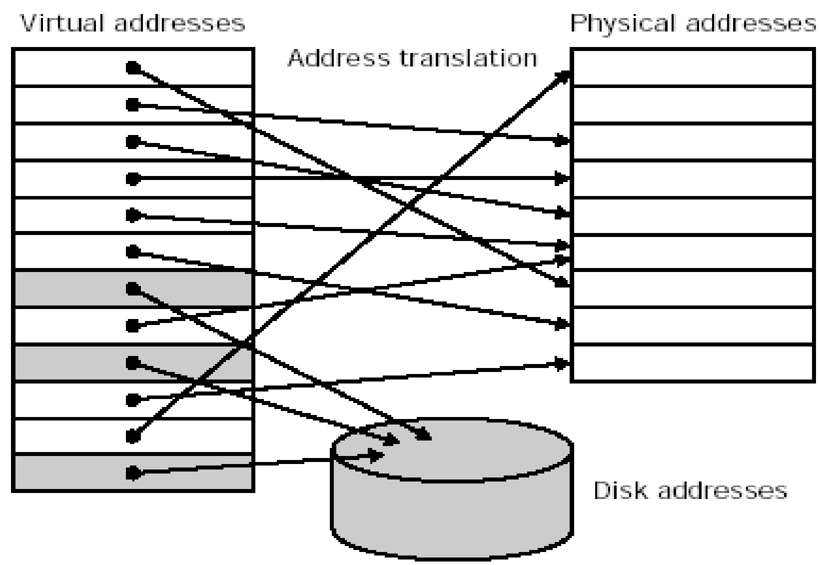


Fig 7.20

Principles same as cache, but different historical roots mean different terminology

Virtual memory block ---> page

Virtual memory miss ---> page fault

Virtual memory also simplifies the process of loading a program in memory

relocation: program can be loaded anywhere

Virtual memory: addressing

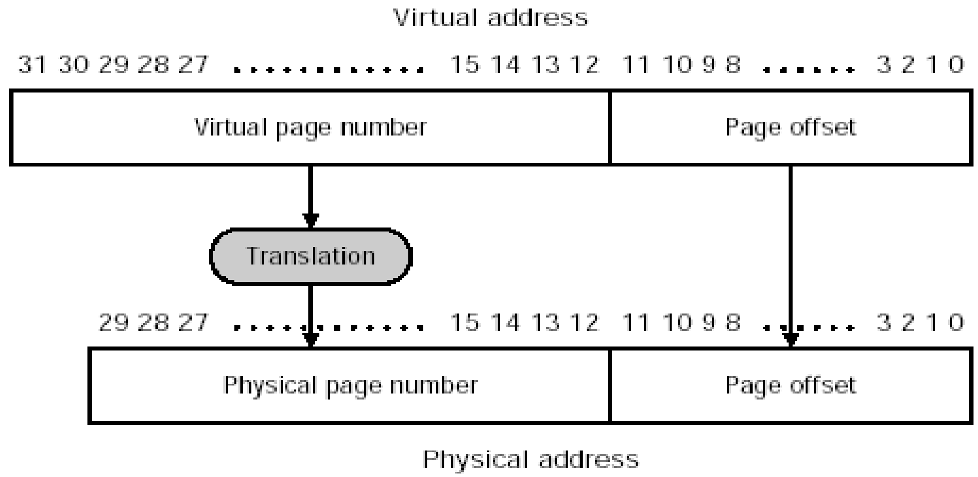


Fig 7.21

Instructions use virtual address

Virtual page number (upper 20 bits) translated to physical page number for memory

Note that the physical address has fewer bits (30 in the example)

- **Different number of virtual and physical pages**
- **How big is physical memory? As big as we can afford.**
- **How big is virtual memory? As big as the architecture can support.**
- **One of biggest system design mistakes: too few address bits**
  - **"Nobody will ever need more than 640K" (W. Gates)**

Page offset (lower 12 bits) remains same

- **Location within page**
- **Number of bits: determined by page size**
  - **12 bits for 4K page**

Virtual memory: design issues

Design issues for VM are related to HUGE cost of a miss (page fault)

Accessing disk may take MILLIONS of clock cycles

SRAM: 5-25ns

DRAM: 60-120ns

Disk: 10-20 million ns

- Pages should be large enough to cover the cost of page fault
  - transfer time is much less than access time
  - 4KB to 16KB common
  - newer systems: 32KB - 64KB
- Reducing page fault rate has high priority
  - fully-associative page placement
- Page faults can be handled in software
  - overhead is small compared to cost of disk access
  - use clever algorithms to minimize page faults
- Write-through is too slow
  - use write-back to store modified data

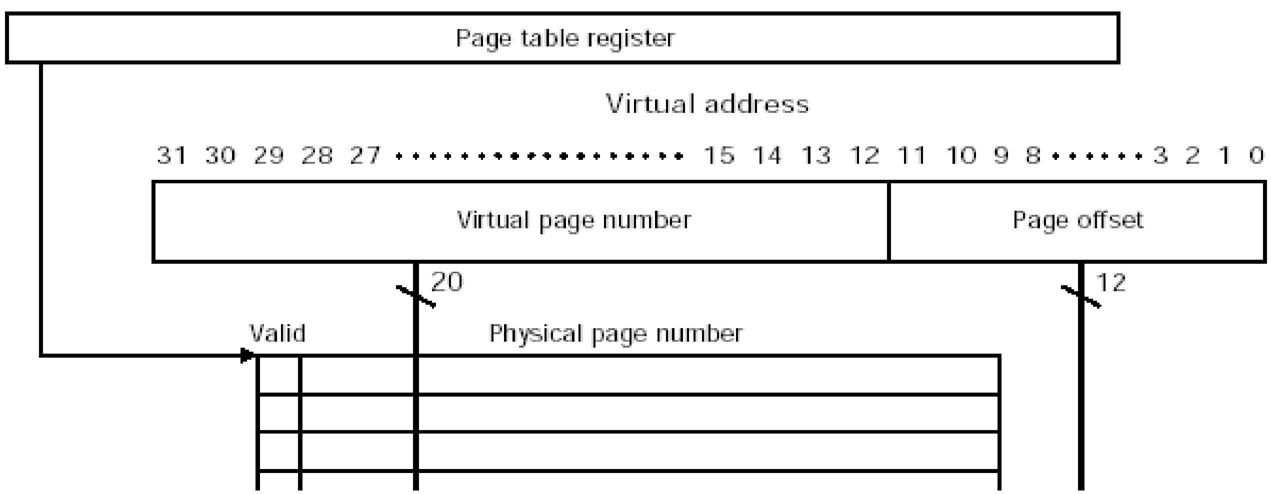
Virtual memory: page management

Optimizing page placement: reduces page faults

- **Fully-associative: map any virtual page to any physical page**
- **Operating system can afford to use sophisticated algorithms and data structures**
  - **to keep track of page usage**

Problem with fully-associative mapping: need to search entire set of pages

- **Full searching is impractical: use page table**
  - **Stored in memory**
  - **Contains the physical page number for every virtual page number**
  - **Each program has its own page table**
  - **To assist accessing page table, hardware has page table register**
    - **Points to start of page table**
  - **Valid bit is used as in cache**
  - **Note that page table is indexed by virtual page number: no tags needed**



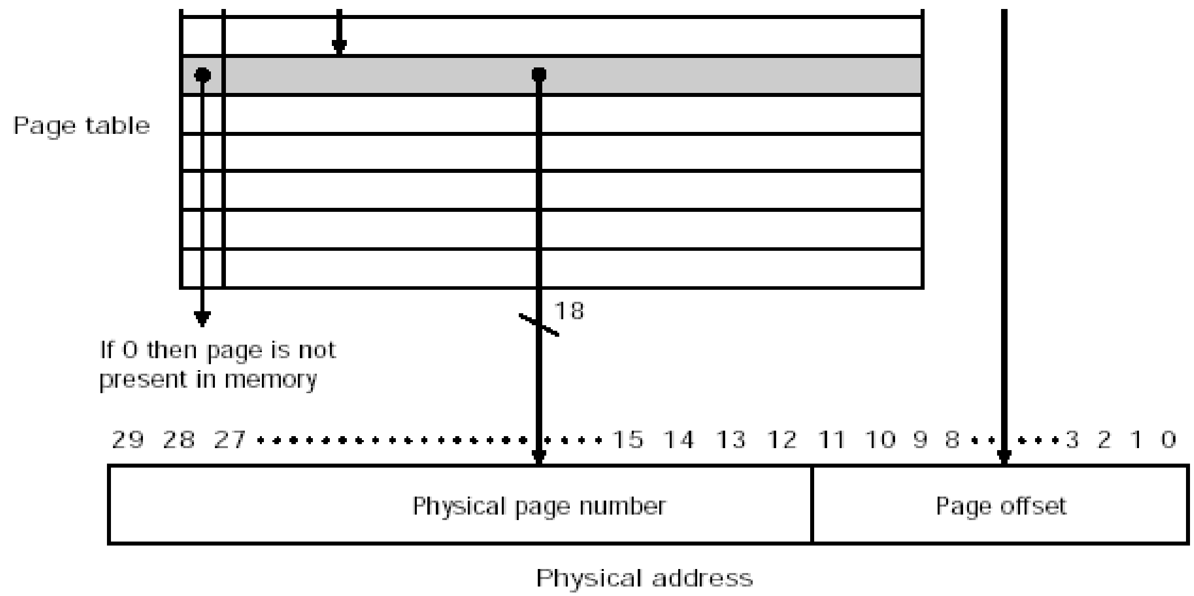

Page table
18
If 0 then page is not
present in memory
29 28 27 15 14 13 12 11 10 9 8 3 2 1 0
Physical page number
Page offset
Physical address

Fig 7.22

Virtual memory: page fault

Page fault: valid bit for virtual page is 0

Operating system gets control, finds required page and puts it in memory

Virtual address does not directly tell where page is on disk

Operating system has data structure to keep track of where pages are located

May be part of page table or separate

Example of single table:

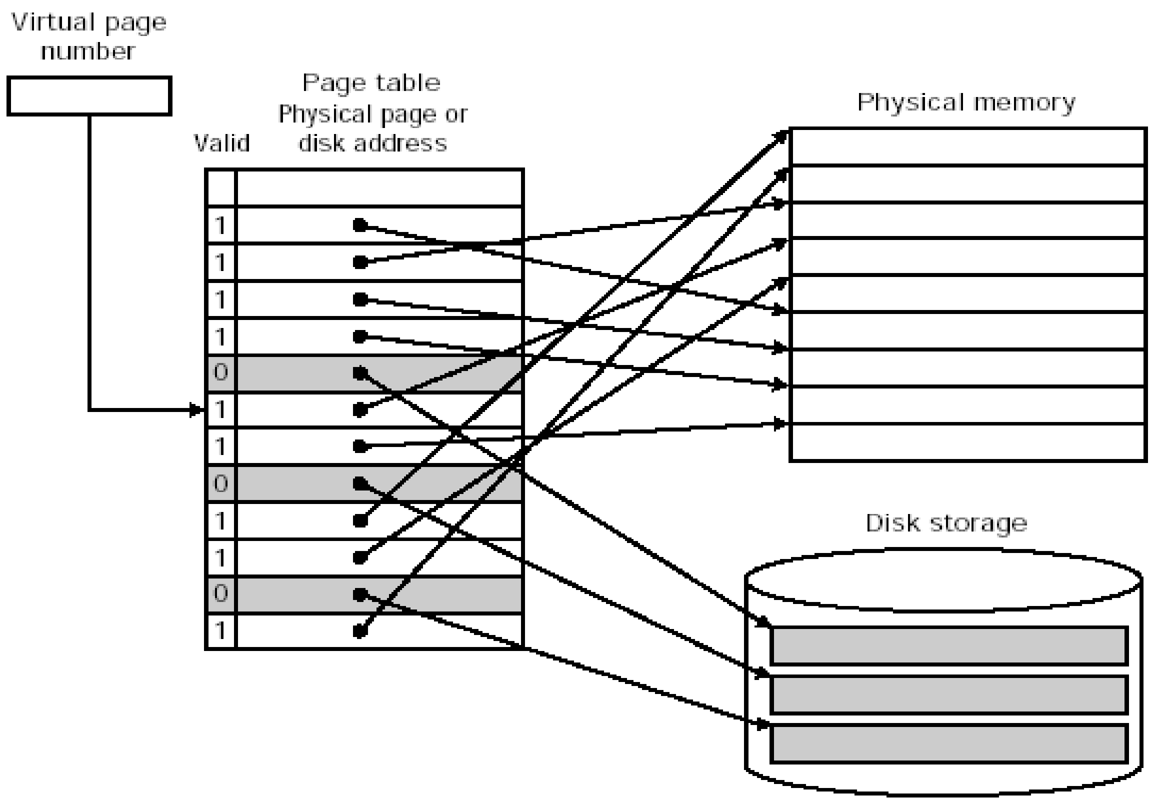


Fig 7.23

Virtual memory: page table

How large is the page table for 32-bit address range?

Assume 4K block size

Number of table entries:

2^{32} addresses / 2^{12} addresses per block = 2^{20} blocks

Each table entry:

20 bits of address + valid bit: use 4 bytes

Total bytes

$4 * 2^{20}$ = 4MB

Each program (process) has its own page table

Suppose 50 processes running on a system:

200MB of memory used for page tables!

Alternatives to storing entire page table

- **Let page table grow as memory usage grows**
- **Stack and heap grow from opposite directions:**
  Have 2 tables which each grow with memory usage
- **Use a hash function on the virtual address**
  Only need as many entries as number of physical pages
  Called inverted page table
  Lookup process more complicated
- **Allow page table to be paged**
  Could result in paging loop, but can keep page tables in OS address space
- **Multiple levels of page tables**
  Top level: blocks of pages (64-256), called segments

Virtual memory: TLB

How much does a page fault cost?
- Time to determine page is not in memory plus time to read the page from disk

How much does a page hit cost?
- 1 memory access to find page physical address + 1 memory access for data
- Cost of memory access has doubled!

Solution: cache the page table
- Translation lookaside buffer
  - Contains most recently used page table entries
  - Small: 128-256 entries
  - Fully-associative
- Idea: Locality in memory references means super-locality in page references
- Table entries
  - Tag holds portion of virtual page number
  - Data entry holds physical page address
  - Also valid and dirty bits
- Look up virtual page number in TLB
  - Hit: use physical page address to locate page
  - Miss: may be missing entry in TLB or page fault
    - Load page table entry in TLB and try again
    - If miss, then page fault

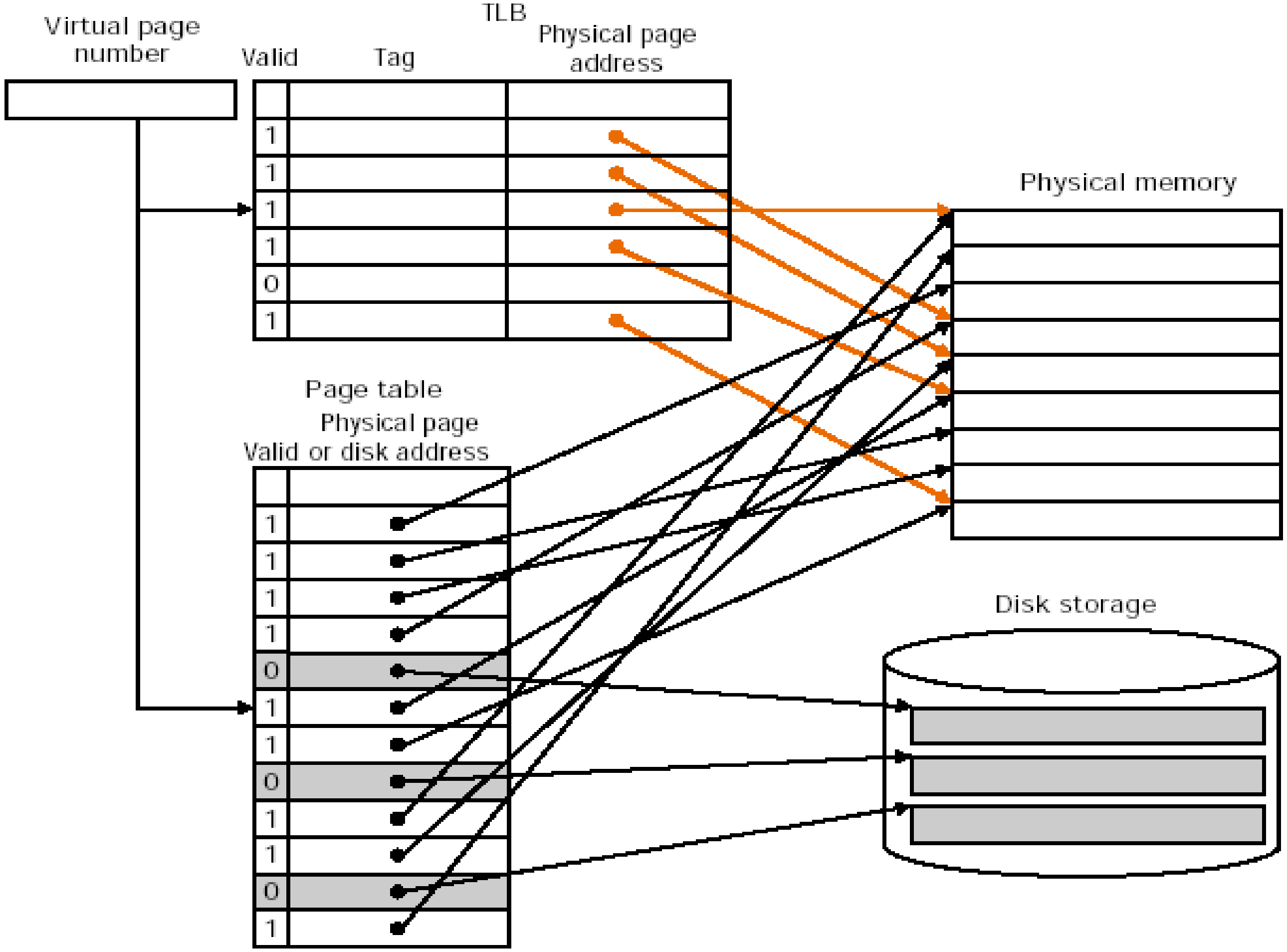

Virtual page number
TLB
Valid
Tag
Physical page address
1
1
1
1
0
1
Physical memory
Page table
Physical page
Valid or disk address
1
1
1
1
0
1
1
0
1
1
0
1
Disk storage

Fig 7.24

Memory: performance

Memory hierarchy can have important effect on performance

Inner loop of matrix multiply:

```
for (i = 0; i < 500; i++)
      for (j = 0; j < 500; j++)
            for (k = 0; k < 500; k++)
                  x[i][j] = x[i][j] + y[i][k] * z[k][j];
```

Running time on Silicon Graphics system with MIPS R4000 processor
and 1MB secondary cache: 77.2 seconds

If loop order reversed so i is innermost: 44.2 seconds

Only difference: order of accessing data

Other compiler optimizations: less than 10 seconds!

Memory: performance

Why memory will be even more important to performance in the future:

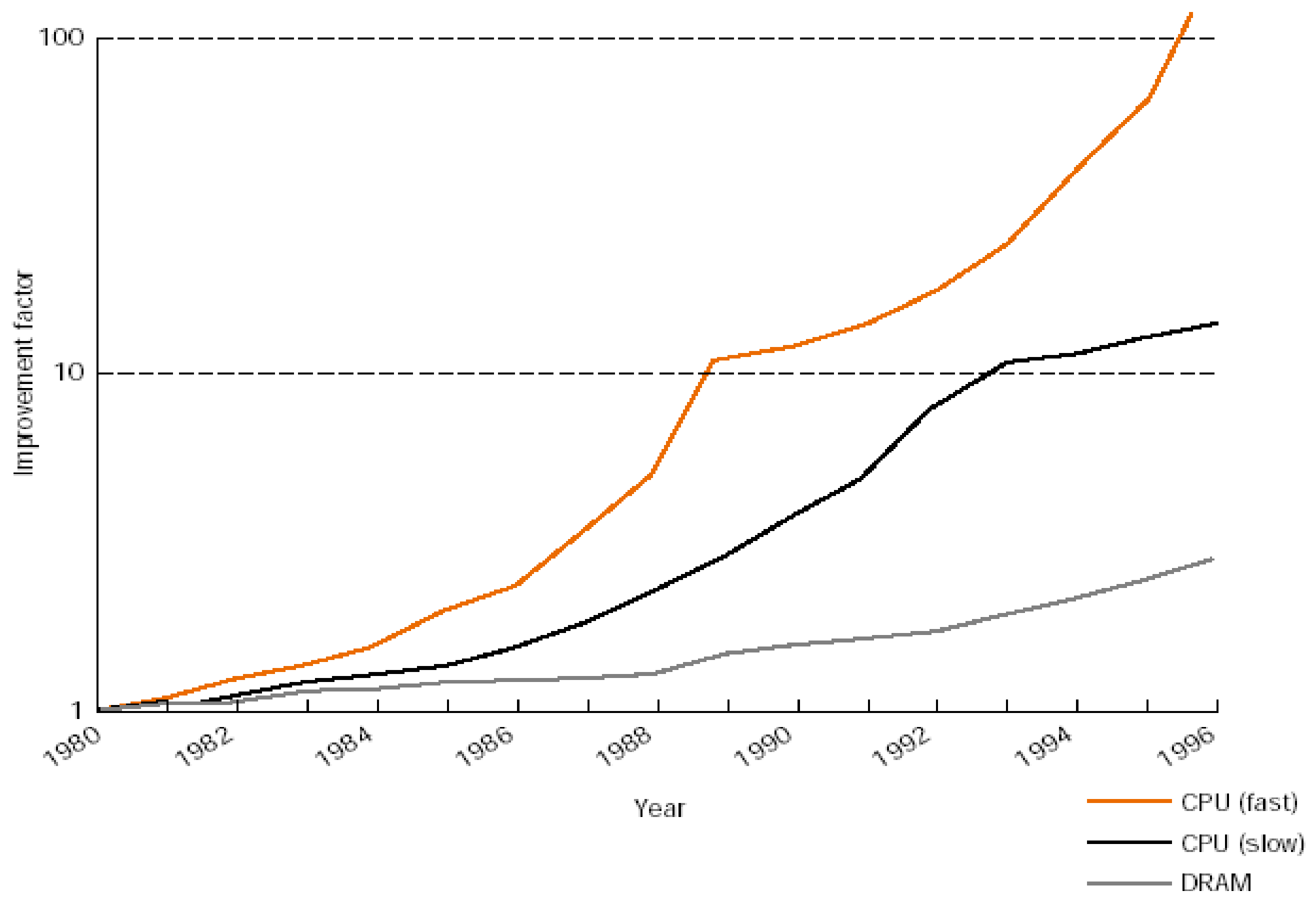


Fig 7.35

Improvement in access time relative to 1980:

DRAM: 9% per year

Slow CPU: 15% per year to 1985, then 25% per year

Fast CPU: 25% per year to 1985, then 40% per year