

# Chapter

# Virtual Memory



### Memory

Baby had an instruction set where the programmer wrote absolute addresses.

If you wanted to add a line all the subsequent lines would have the wrong number.

Any jump backs – say for loops would also need modification.

Wrote the code – created the opcodes and keyed them in.

Corrections were time consuming.

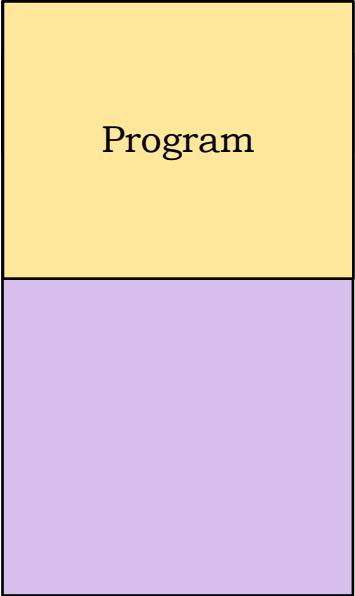
Any subroutine was also a problem. How do you write a subroutine which can be linked in with an arbitrary programme?

Assembler to automate recalculating the offsets etc.

Relative addressing.

Start at 0, all commands at an offset.

On loading merely add start address to all offsets



Program

### 3 Relocation

#### **Relocatable**

Such code is called re-locatable.  
Makes code development much easier (possible?)

Can think of it as a virtual address space for the subroutine 0-173<sub>8</sub> which is mapped onto real memory from 326<sub>8</sub>-531<sub>8</sub>  
code is called re-locatable.

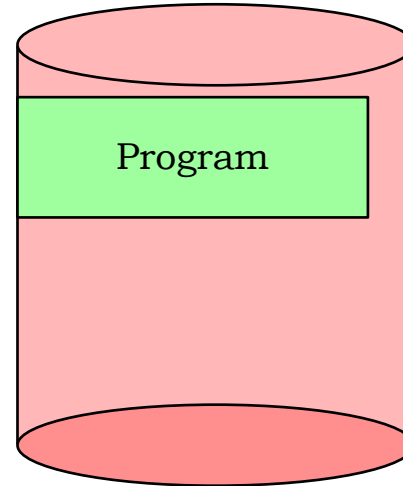
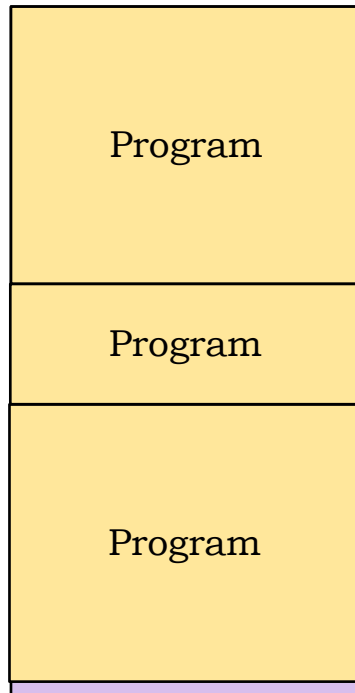
#### **Memory Limit**

Early machines had a small address space.  
An even smaller amount of memory.  
You wrote your programs to fit in the available *physical* memory.  
If it wouldn't fit you simplified the programme.  
Found clever ways of saving an instruction.  
But the limit was a real hard limit.

That is **not** how it would have been described.

### Address mapping

People tried to write self modifying code, but it was never more than a (very small) minority.  
The real solution to create overlays.



The programme is split into sections. The sections must (*clearly*) contain code which is temporally separate.

The programmer must identify when different sections of code are required and arrange for the sections to be loaded at the appropriate time.

That is **not** how it would have been described.

## 5 Overlays

### **Larger than physical memory**

It was not seen as a single virtual address space which was mapped into a smaller physical space.

There were a number of constraints:

a) Variables required outside the overlay had to be defined elsewhere, in a permanently resident section

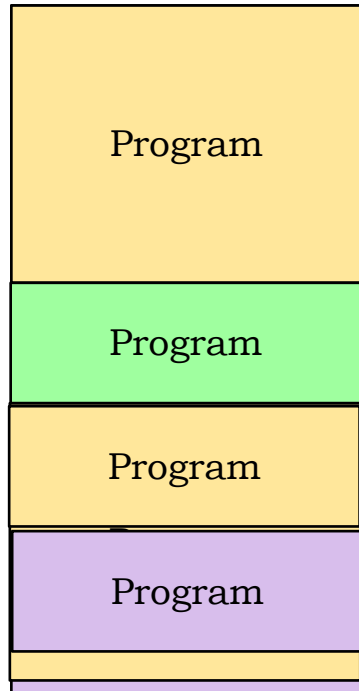
b) With multiple overlays you need to make sure that all overlays are resident at the appropriate times. (Including the call into the section)

c) You needed a root section which was always present.

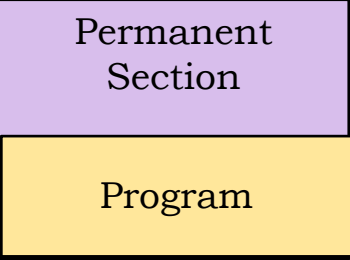
d) The values in the overlay would not be present if it was overwritten and reloaded.

e) You needed to be careful about the state of registers.

Experience did not translate between different machines.



### Example



Permanent  
Section

Program

Real time system to control an experiment.

0.0-0.7s beam of particles interact to create events.

State of detectors need to be monitored.

0.7-2.3s gap between beam. Study of output from the detectors during the last **burst**.

repeat for 2 hours.

10 minute gap

repeat for three weeks

A slight advantage in that there is an external signal which indicates the start and end of beam.

Hard to write, hard to test and verify correct operation.

It would hang approximately once every 24 hours.

Simple in that temporal localisation was enforced by environment.

Taking data.

ensuring data  
quality

## Operating System Overlays

In the mid 70's overlaying moved into the operating system. Improved reliability and programmer productivity.

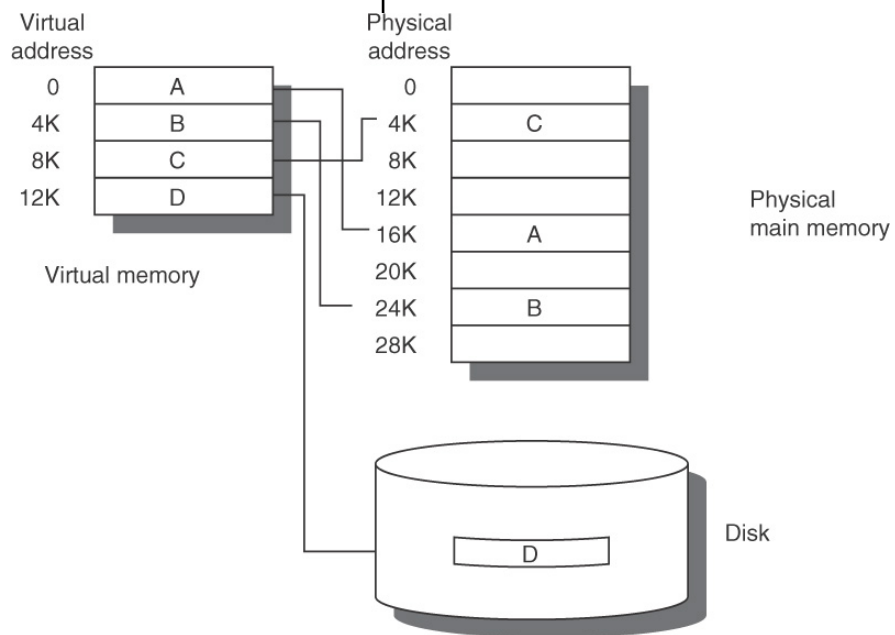
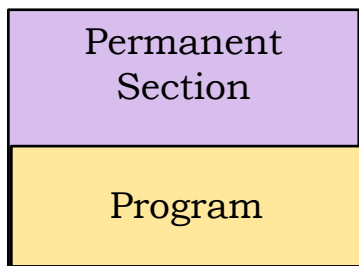
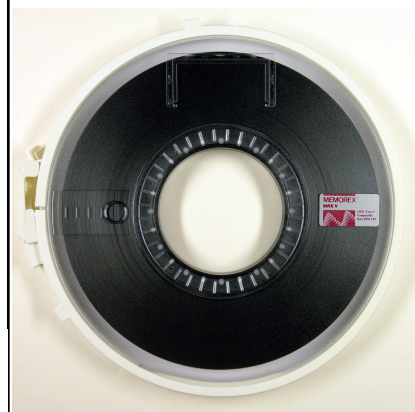
Commonplace for programmers to use big data sets.

Rather than multiple passes and clever algorithms, read all the data into "memory" and let the OS sort it out.

Allowed more than one process to be apparently

running at the same time.

The "overlays" became dynamic and "relocatable", they could be swapped back into physical memory in different locations



## Virtual memory terminology

Virtual memory shares many concepts with cache, but different terms.

The unit which is swapped in and out of memory is called a **page** or **block**

A **cache miss** becomes a **page fault** or (**address fault**)

The processor produces **virtual addresses** translated to **physical addresses** in memory  
**Memory mapping** or **address translation**

Special place on disk – *the page file*

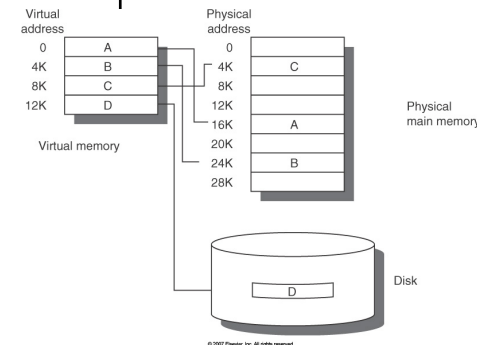
Excessive *faulting* leads to repeated replacement of the pages in memory **disk thrashing**

May lead to very little work being undertaken.

Cache replacement is implemented in hardware

Virtual memory strategies are OS determined.

Miss penalty is higher so the algorithm can be more complex.





**VM v. cache**

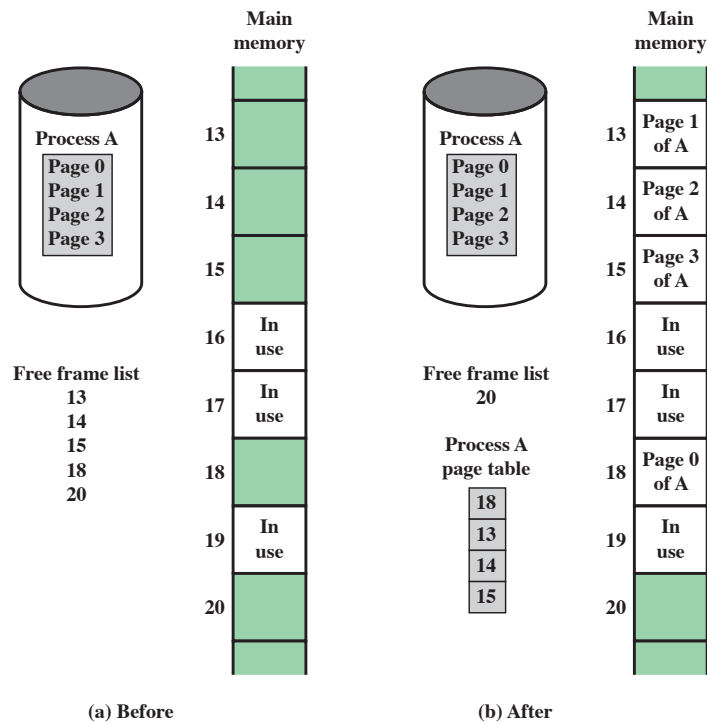
<b>Parameter</b>	<b>1<sup>st</sup> Level cache</b>	<b>Virtual Memory</b>
Unit size	16-128 bytes	Up to 8TB
Hit Time	1-3 cycles	100-200 cycles
Miss penalty	8-200 cycles	1-10 million cycles
Access time	6-60 cycles	0.8-8M cycles
Transfer time	2-40 cycles	0.2-2M cycles
Miss Rate	0.1-10%	$10^{-6}$ - $10^{-30}$
Miss Time	0.1%-22%	0.5%-50%
Mapping	25-45 physical to 14-20 cache	32-64 bit virtual-25-45 physical

Processor address space determines the size of virtual memory – hence importance of 64 bit. VM can be fixed size **pages** or variable size **segments**. Pages lead to inefficient memory usage as the pages may contain many empty cells. Modern machines base their system on pages although they may allow multi-page transfer, which are also called segments.

	<b>Page</b>	<b>Segment</b>
Words per address	1	2
Programmer visible	Invisible	Possibly visible
Replace a block	Trivial	Computationally hard
Memory efficiency	Internal gaps	Gaps between segments too small to fill
Disk traffic	Yes. Choice of page size	No -small segments are inefficient. Access time > transfer time

High page fault penalty, so optimise for low page faulting.

# 11 details



The memory divided into **frames** (or page frames)  
The compiled programmes are divided into **pages**

They are of course the same size.

A problem is now apparent.

Compile code contains memory locations:  
location of data words, targets of jumps and  
subroutine calls.

The programme with addresses starting at some arbitrary location. This is the logical address of the instruction or data word.

When a page is copied into a frame, the locations are all at a physical address in memory.

Need to translate from logical to physical.  
Add the base address of the physical location to the logical location.

Like the overlay when we bring a page in from disk it may go into a new place.

Modern OS use **demand paging**

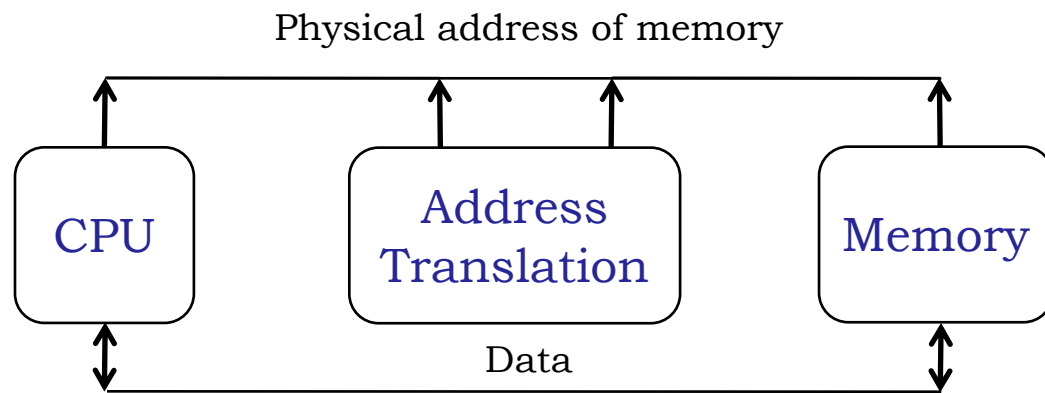
A page is only brought in off disk when it is required.

Access to a page not in memory generates a **page fault** (cf cache miss)

It will overwrite a page already in memory **page replacement** for which we need an algorithm analogous to cache replacement.

Dynamic version of the relocatable loader

## 13 Translation\*



The size of the logical address space may be larger than the physical memory  
**Virtual memory**

All programs share the same physical address space  
Machine Language programs must be aware of machine organisation. (compilers for HLL)  
Program can access any machine resource.

Programs run in a standard virtual address space.  
Address translation managed by hardware which maps virtual address to physical.

**Address translation** supports:  
multi-threaded programming.

Stacks may be allowed to grow in disjoint physical memory, but in contiguous virtual memory

### **Protection**

protection of sensitive areas of memory  
protection of threads from each other  
Kernel data protected from user programs **security**

### **Sharing**

Sharing of memory between processes (threads)  
Speedy interchange between programs

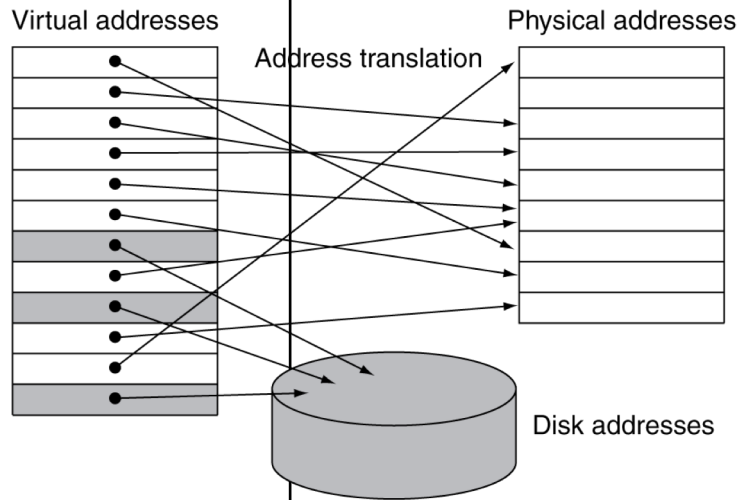
Memory acts as a cache for disk

Security is difficult to enforce

Address translation adds complexity, but with numerous benefits

Virtual Mem

# 14 Translation



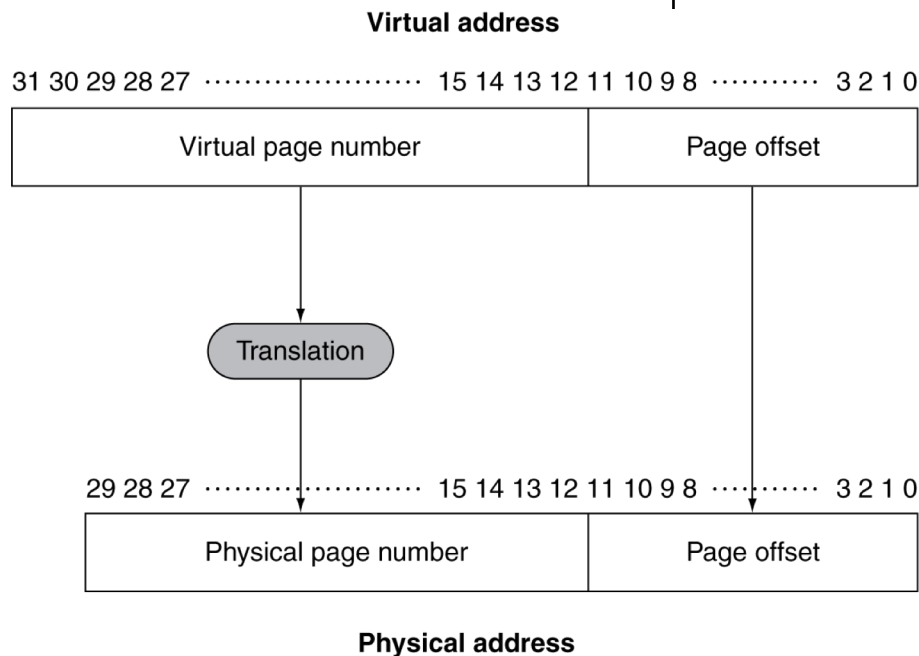
Both real and virtual memory is addressed in pages.

Processor generates virtual addresses, memory addressed are translated into physical address – which may be in memory or on disk

Address in page same physical and virtual. In this case  $2^{12}$

Size of virtual memory is not the same as physical.

VAX 11/780 had typically 1-4Mbyte of physical memory, but 4.3Gbyte virtual. Reading a large amount of data into “memory” (say a large array) and then process it without data read.



## Finding the page

Page fault is very expensive, transferring data from disk is thousands of times slower than even from main memory – fractions of a percent reductions in rate are worthwhile.

Complex algorithms can be used to track page usage.

we don't want conflict misses  
so Fully associative placement.

But search to find which frame has the required page is too time consuming

4Gbytes memory – which is not large  
4Kbyte pages – which is standard

1 million comparisons – even 1 comparison per clock operation gives  $\frac{1}{4}$  millisecond for a location which is **in** memory

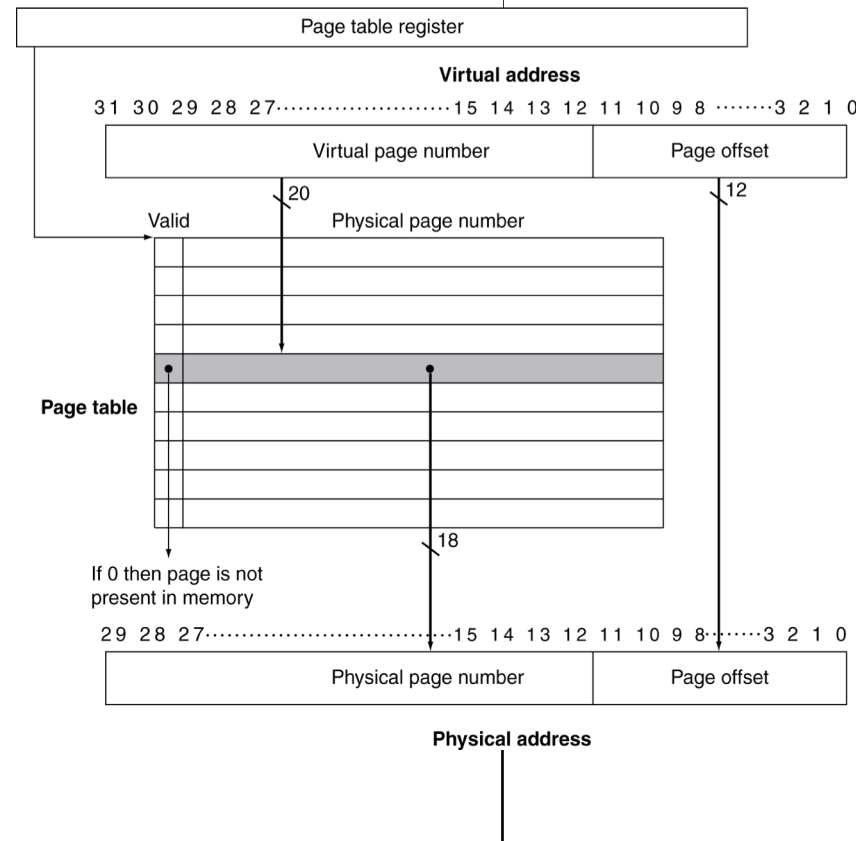
## Finding the page

Array which indexes the memory – kept in memory  
**Page Table** translates virtual address to physical

Each program has own page table

Special hardware register

**Page table register** points to page table in physical memory



Page table contains an entry for every page.  
 Tag bit to say if it is in memory



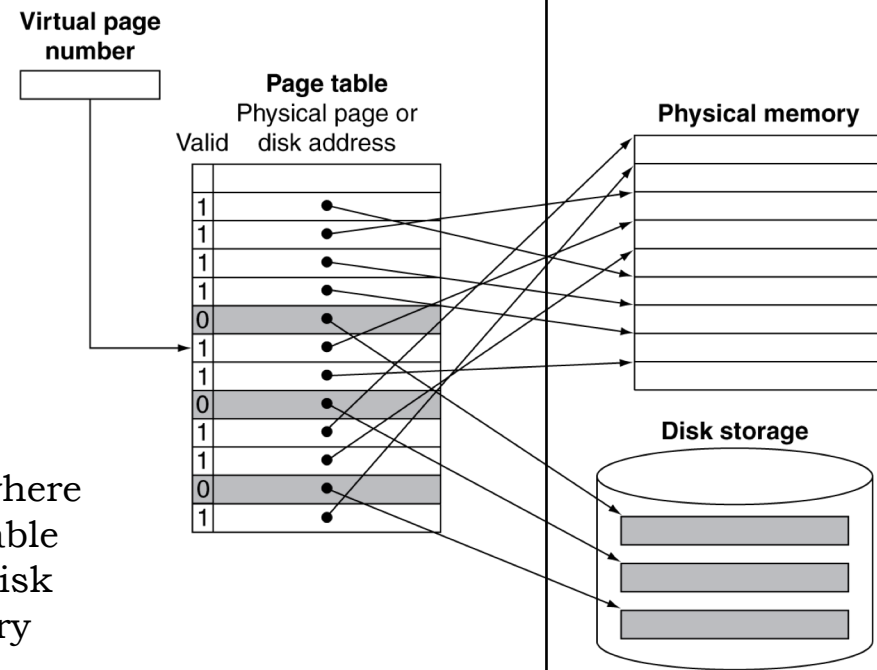
## Using the table

Valid bit = 0. generates a page fault  $\Rightarrow$  OS  
 OS has virtual address – needs to get it from an address on disk.

Could get page from source file – update values?  
 Creates an area **Swap Space**.

Needs a map from virtual memory to disk address.  
 Logically another table – indexed as the *page table*  
 May be the same (as shown).

When Physical memory is full replacement algorithm to decide on page to replace



Example where the page table points to disk and memory

**Replacing pages**

LRU is the algorithm of choice.

Full LRU means structure update on each access

Approximate LRU: reference or use bit set on access.

But cleared periodically. Page with bit=0 can be replaced.

**Writes**

Write through is impractical – as is write buffer.

For disks access time > transfer time.

Copy pages not items (spatial locality).

Update page in memory – set **dirty bit**

If page is replaced then only write out before replacing.

If possible choose a page without the dirty bit set.

**Problems with the page table**

32 virtual address – 4KByte pages  $\Rightarrow 2^{20}$  entries

4bytes per entry

4 Mbyte – page table.

1 page table per process

Currently my machine has 77 processes. – 300 Mbytes

1 Gbyte – memory 30% on the page table.

**Solution**

Page the page tables.

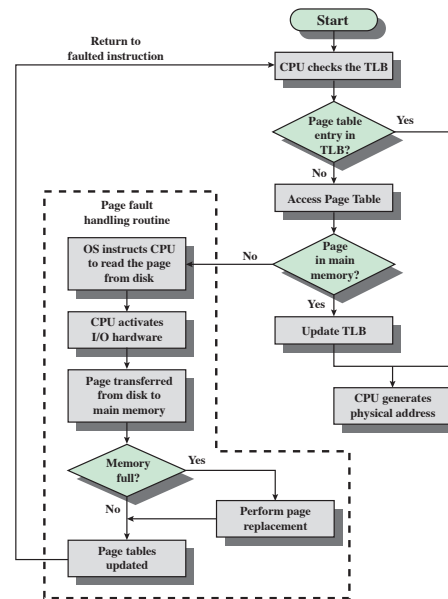
Tables sit in the OS virtual space (not process)

A page in memory must have its part of the page table in memory.

There are parts of memory which are non-paged.  
Data in these areas cannot be swapped out or overwritten.

Some OS pages are in non-paged memory – don't want to have to wait for some parts of the OS

## Flow diagram for address translation



This is very good, but it has caused a problem.

When a physical address is required, it needs one memory access to get the instruction/data.

With a virtual address you need to access the page table to get the physical address and then access the physical address.

The memory access time has just doubled.

To get round this problem a special cache is provided which holds part of the page table.

## Translation lookaside buffer TLB

## Replacing pages

Address translation is likely to be used again very shortly; both temporal locality and spatial locality.

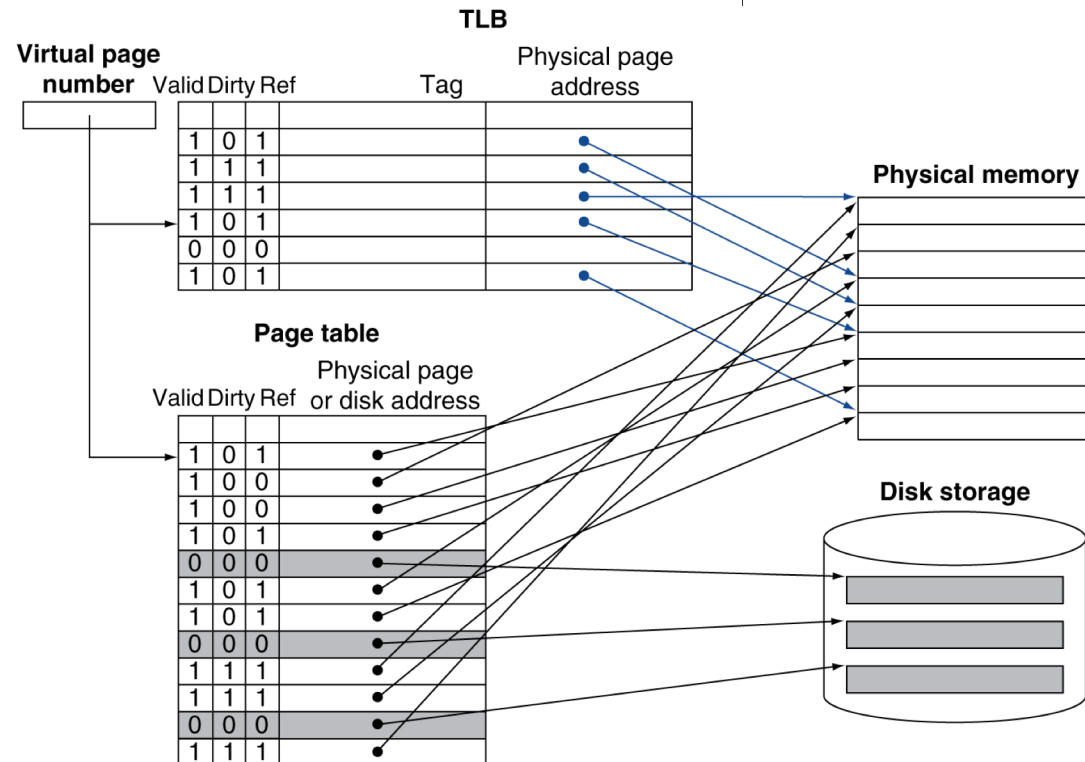
Keep recently used translations in a special cache

### Translation-lookaside buffer

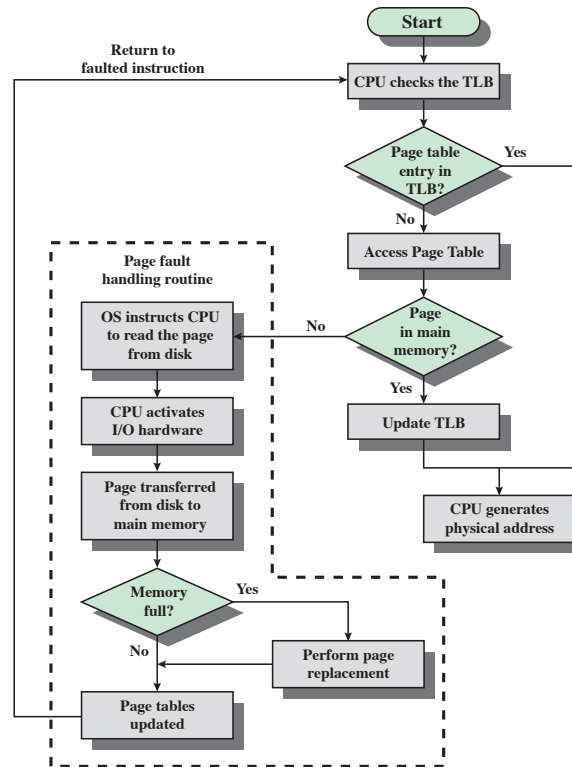
dirty bit, valid bit must be available here.

Miss on the TLB

Translation not  
in the buffer



## Flow diagram for address translation



A diagram of what happens when the cpu wants to translate a logical address into a physical address.

First check the TLB.

Then need to access the page pointed to by the TLB.

**TLB Miss**

TLB Hit – pass address and set dirty bit for write

TLB miss:

page in memory but not in TLB. Load address into TLB, copy dirty bit back to the page table for the replaced page into and restart

page nor in memory. Throw an exception, handle in software or hardware.

Typical values

TLB size	16-512 entries
Block size	1-2 page table entries
Hit time	0.5-1 clock cycles
Miss penalty	10-100 cycles
Miss rate	0.01-1%

Storage and replacement strategy has to be defined for the TLB.

**Problems with the page table**

32 virtual address – 4KByte pages  $\Rightarrow 2^{20}$  entries

4bytes per entry

4 Mbyte – page table.

1 page table per process

Currently my machine has 77 processes. – 300 Mbytes

1 Gbyte – memory 30% on the page table.

Many techniques  
have other  
implications

**Solutions**

Limit register restricts the size of the table. Grow page table as required. Addresses can only grow in one direction.

Systems want to grow from high memory down and low memory up. So two pages tables and two registers. If memory is filled sparsely page table will grow large.

Make the page table the size of memory – placing the virtual address via a hashing function. **Inverted page table.**

**Multi-Level pages tables**

1 level is at segment (multi-page level) pointing to page table if segment is used. Useful for sparse allocation (matrix problems) – decode more complex.



Size  
Access time  
cost

# Memory hierarchy Revis

*faster*

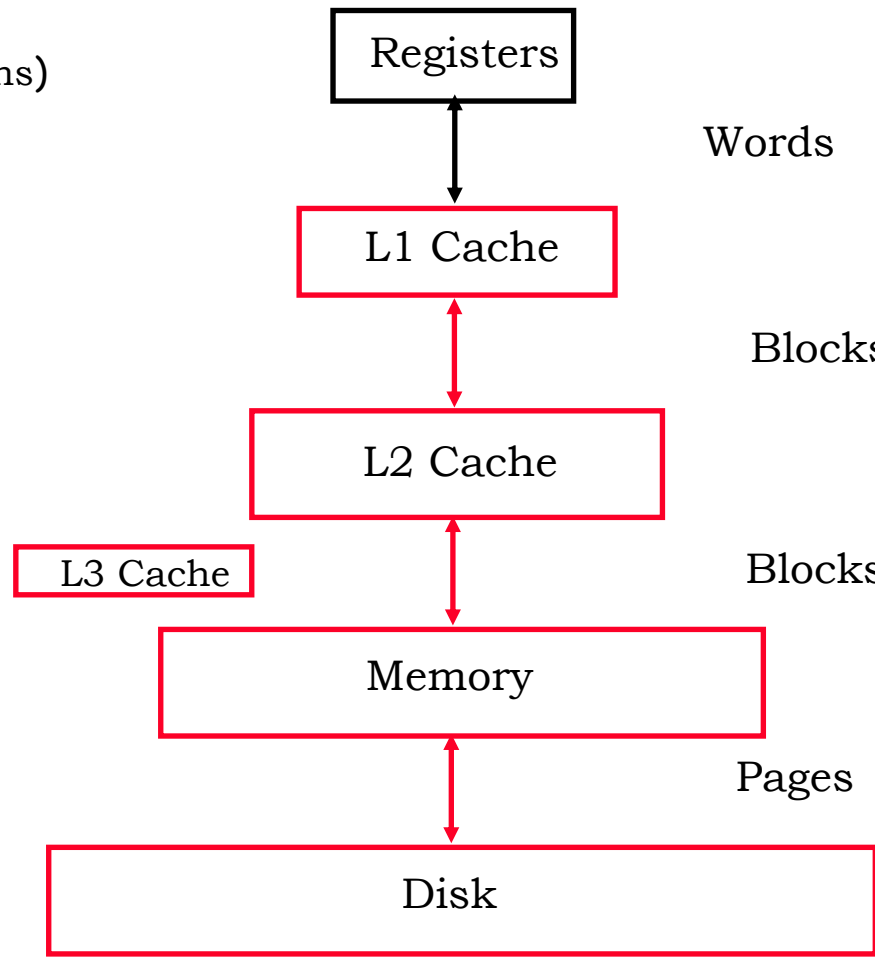


100s Bytes  
300 – 500 ps (0.3-0.5 ns)

10s-100s K Bytes  
~1 ns - ~10 ns  
\$1000s/ GByte

G Bytes  
80ns- 200ns  
~ \$100/ GByte

10s T Bytes, 10 ms  
(10,000,000 ns)  
~ \$1 / GByte



Words

Blocks

Blocks

Pages

Files

Staged by  
transfer Unit

prog./compiler  
1-8 bytes

cache cntl  
32-64 bytes

cache cntl  
64-128 bytes

OS  
4K-8K bytes

user/operator  
Gbytes+

Upper Level

Lower Level

*Larger*

The ideas are similar at every level. The solutions depend on the details at that level

**Review**

Placing a block : associativity: Direct; n-way; full  
 Finding a block

Associativity	Location method	Tag comparisons
Direct mapped	Index	1
n-way set associative	Set index, then search entries within the set	n
Fully associative	Search all entries	#entries
Full lookup table	0	

Replacing a block: LRU - random

Writing to a block: write through: simpler needs buffer  
 write back

Misses: Compulsory; increase block size  
 Capacity misses; increase size –  
 Conflict miss: increase assoc -  
 (collision)

Miss amelioration  
 requested word first  
 hit under miss and miss under miss  
 prefetch

Increase miss penalty  
 Increase access time  
 Increase access time

Large memories are slow:technology and size  
 Cache: apparently large  
 fast memory  
 Depends on locality