Memory

If we define memory as a place where data is stored there are many levels of memory:

- Processor registers
- Primary (or main) memory
  - RAM
- Secondary memory
  - slower and more permanent
  - disks
- Tertiary memory
  - archival
  - removable
- Cache memory

one level of memory pretending to be another

- different levels of cache
  - at the processor
  - at devices
  - and at hosts in distributed systems

Main Memory

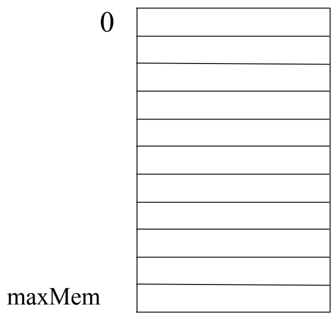
All processes need main memory.

Instructions and data being worked on are brought into the processor from memory and sent back to memory.

Traditional view:

Addresses are numbers: 0 – maxMem

An address goes out of the processor to the address bus of memory.



Address binding

We can make the connection between code and data and their addresses at different times:

- Compile time

Need to know exactly where the value will be stored.
Fixed addresses.
Not very flexible.

- Load time

Object modules need tables of offsets.
e.g. variable x is at 24 bytes from the start of the module
The mapping is done as the module is loaded.
Can also reference other modules – linking.
More flexible but can't be changed after loading.

- Dynamic loading – don't load unless called
- Dynamic linking similar (useful with shared libraries) – may already be in memory – if shared between processes needs OS help

- Run time

Mapping to final address maintained on the go by hardware.

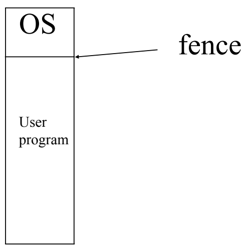
Memory spaces

Even in simple systems we would like a way of protecting OS memory from our program and enabling programs larger than memory to run.

Split memory

Can protect with a single fence register.

Alternatively if the OS is in ROM it is safe from overwriting.



Overlays

Load needed instructions over the top of ones not needed any more.

Dividing memory

With linking loaders we can have multiple programs in memory simultaneously.

We also want protection.

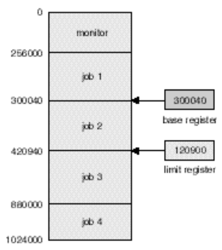
In the history section we saw separate partitions and base and limit registers.

Both require contiguous memory.

The algorithm for base and limit registers is simple.
If the address is less than the base or greater than the base + limit – 1 then we have an access violation.

The base and limit registers are loaded for each process.

We need a lot more memory, maybe more than we have.
So we could use overlays in each allocated area.



Two different addresses

If we change our base and limit system to produce the address by adding the base register (now called a relocation register) to each address we can make all processes start at address 0.

This is a huge conceptual change.

We now have two types of addresses.

- The logical (virtual) address coming out of the process
- and the physical (real) address used on the address bus to memory.

We still have contiguous memory for each process but a process can now be positioned anywhere in physical memory without its addresses needing to be changed.

We can even move a process around, just copy the memory to the new place and change the relocation register.

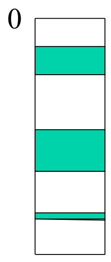
This is useful if we want to defragment memory to give one large free area.
Have to be careful if moving data (e.g. from a device) into the process' memory space.

Split memory into smaller chunks

Rather than moving memory around to make big enough spaces for processes we could have more than one section of physical memory accessed by the same process.

We need either a number of base-limit registers or a table of the information.

Chunk	base	limit
0	1024	1024
1	8192	512
2	4096	2048



The process sees 3.5K of contiguous memory.

Two approaches

This technique evolved in two directions.

1. Same sized chunks – pages
2. Variable sized chunks – segments

Both have advantages and disadvantages.

Both use Memory Management Units (MMUs) in hardware to do the translation between the logical and the physical address.

Rather than doing a tedious calculation (where is logical address 2000 on the previous slide?) to find what chunk an address is in, we just split the address into two parts.

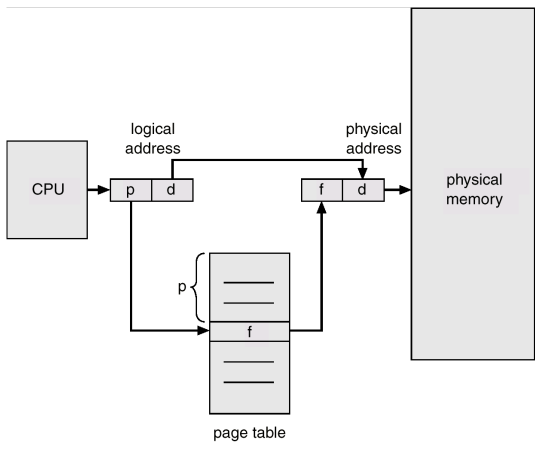
Then the translation is much simpler and looks very similar in both paged and segmented systems.

Paged system address translation

Logical address is divided into:

Page number – index into a *page table* which contains base address of each page in physical memory

Page offset (displacement) – added to base address to get the physical address.



In this case there is a constant number of bits for the offset. This means that pages are always powers of 2 in size – usually from 2048(11 bits) to 8192(13 bits).

Some systems allow variable sizes of pages.

x86 variable sized pages

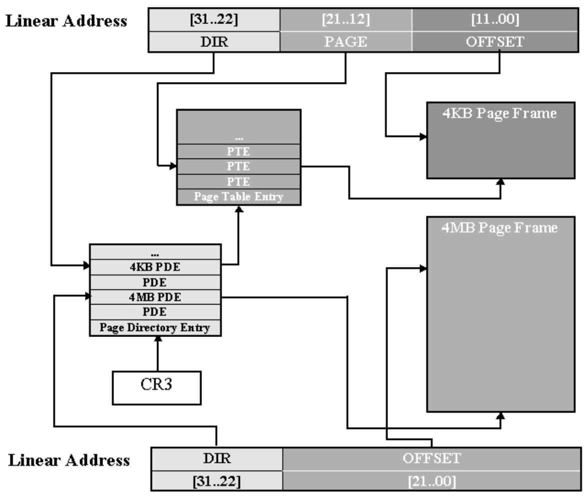


Diagram from http://www.x86.org/articles/4mpages/4moverview.htm

Frames and pages

The textbook makes a distinction between pages and frames.

A frame is a page sized chunk of physical memory that can be allocated to a page from a process.

A page is a frame sized chunk of logical memory that fits in a frame.

It is common to refer to both simply as pages (physical and logical).

<u>Fragmentation</u>

No external fragmentation between pages.

Internal fragmentation in any pages that are not completely used.

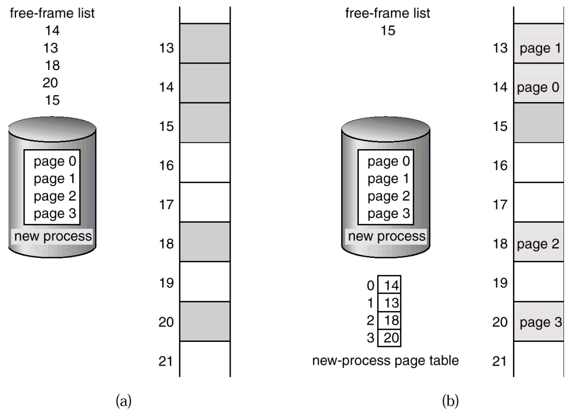
Could be an average of ½ a page per process area (or ½ a page per thread, ½ a page for heap space, ½ a page for code, ½ a page for initialized data, etc).

So small pages save space but require larger page tables

Tables

Each process has its own page table.

And commonly the OS has a frame table with one entry for each frame showing whether it is free or not and which process is currently using it.



Different sized chunks

Rather than constant sized pages we could design our hardware to work with variable sized chunks – these are known as segments. (Not to be confused with variable sized pages.)

<u>Memory model</u>

How memory appears to be organised to the program (and programmer) is sometimes referred to as the memory model.

A segmented memory model is when we look at memory as a group of logical units all with addresses starting at zero.

Processes can have lots of segments
- functions
- global variables
- activation records
- large data structures (arrays)
- objects

Segments

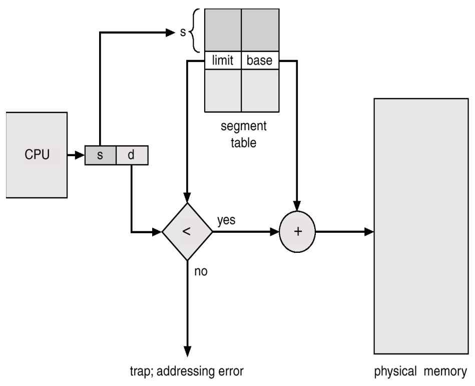
These logical units fit nicely into hardware specified segments where different amounts of memory can be allocated in one chunk.

Segments are contiguous blocks of memory.
We will get problems of external fragmentation.

Our memory addresses become
<segment name, displacement>.

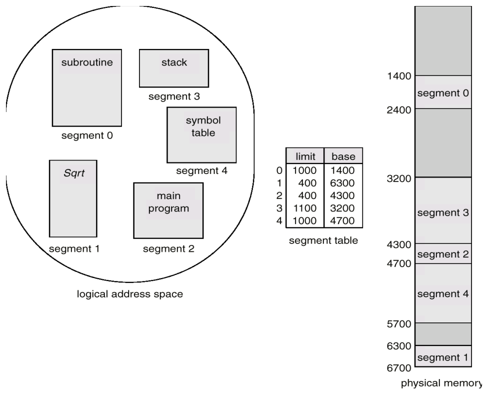
And the translation process looks very much like paging, except there is a length associated with each segment. We have a segment descriptor table rather than a page table.

The physical address is the result of an addition rather than a concatenation as it is in a paged system.



Example

Ideally segments should be able to cover all of the addressable memory – this could mean that logical addresses might have more bits than physical addresses.
Some segmented memory systems restrict segment sizes to prevent this.



Allocation strategies

Segments have no internal fragmentation – we only allocate the amount of space we want.
What is one obvious problem with this?

But we get external fragmentation.

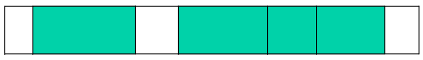
We have seen the allocation strategies before:
- first fit
- next fit (first fit but starting from where we were up to)
- best fit
- worst fit

We can defragment memory if we want to find large enough chunks. Faster than defragmenting disks.

How much space in the holes

Knuth's 50% rule
If there are n segments there are n/2 holes.

Each segment is either below another segment or below a hole.
We always combine adjacent holes.



Each segment is released independently – so in a steady state system the space above each segment will alternate between being used and being free. 50% of the time there will be a hole above each segment.

If the average size of a hole is the same as the average size of a segment we need about 1/3 of the memory free to keep this system running.

Before next time

Read from the textbook
8.2 – Swapping
8.5 – Paging
8.6 – Structure of the Page Table
8.7.1 – Segmentation with Paging
9.2 – Demand Paging
18.6.2 – Linux virtual memory