Half speed memory

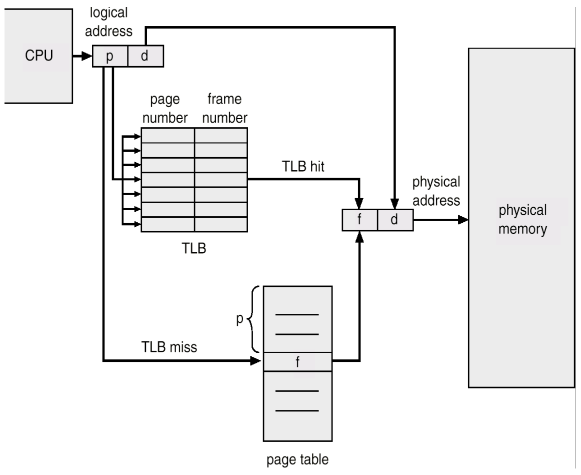
In both paged and segmented memories every logical memory access requires (at least) two memory accesses. One for the page/ segment table and one for the actual data.

Actually the number of segments may be quite small and there may be registers for them.

So the MMUs cache recent page table information in a special fast-lookup hardware cache called *associative registers* or *translation look-aside buffers (TLBs)*

Page #	Frame #

TLB use



Average access times

TLB Lookup = ε time unit

Assume memory cycle time is β

Hit ratio – percentage of times that a page number is found in the associative registers; the ratio is related to the number of pages cached in the TLB.

Hit ratio = α

Effective Access Time (EAT)

$$EAT = (\varepsilon + \beta)\,\alpha + (\varepsilon + 2\,\beta)(1 - \alpha)$$

$$= 2\,\beta - \alpha\,\beta + \varepsilon$$

e.g. $\alpha = 0.98$, $\beta = 1$, $\varepsilon = 0.1$

EAT = 1.12 (compared to 2 for no TLB)

TLB coverage

TLB coverage (or reach) is the amount of the address space included in the TLB entries.

Typical TLB caches hold about 128 entries.

With 8K pages this is only a megabyte of memory.

As working sets (more on those later) increase this means lots of processes have a real performance hit, memory wise.

The solution is larger page sizes. This means more internal fragmentation. More IO (in virtual memory systems).

Variable page sizes can be used but they need clever allocation algorithms to be worthwhile.

Page table size

Another problem with page tables is their potential size.

e.g. 32bit address space and 4Kbyte pages (offset of 12 bits).

So 20 bits to index into the page table ≈ a million entries (4Mbytes for each process)

You can work out the equivalent for 64bit address spaces.

Most processes do not use all memory in the CPUs logical address space.

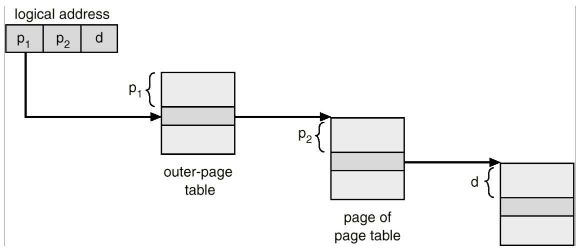
We would like to limit the page table to values that are valid.

Can do this with a page table length register.

Can flag page table entries with a valid bit.

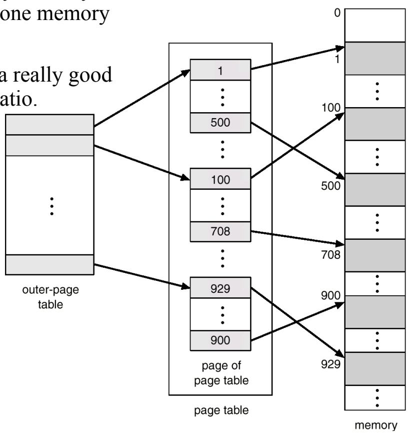
- Only allocate the parts of the page table we actually need.
- Page the page table (see virtual memory)

Multi-level page tables



How many memory reads for one memory access?

We need a really good TLB hit ratio.



Inverted page tables

As the number of address bits increases to 64 we need even more levels of page tables.

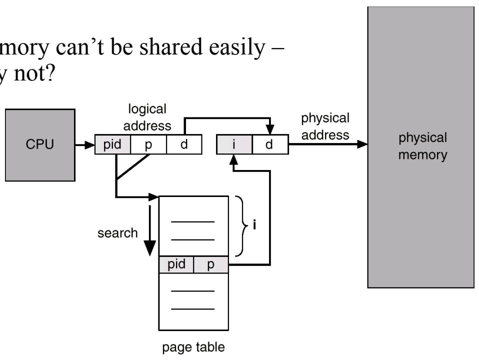
Another approach is to keep information about the physical pages (or frames) rather than all of the logical pages. This is known as inverted page tables.

Only need one page table for all processes. Each entry needs to refer to the process that is using it and the logical address in that process.

A logical address is <pid, page number, displacement>

Have to search the page table for <pid, page number>. Use hashtable for the page table and rely on TLBs.

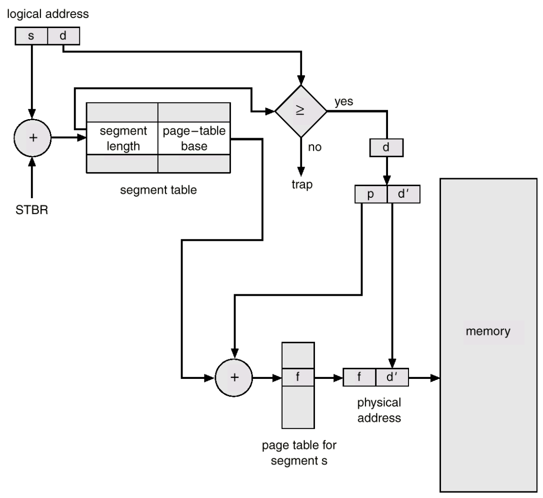
Memory can't be shared easily – why not?



Paging and Segmentation

The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments.

Different from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a *page table* for this segment.



Programs larger than memory

It has always been the case that no matter how much memory a computer system has there are programs that need more.

This was handled early on by overlays.

But these required care on the part of the programmer to split the program up into distinct sections.

Also any connection between the sections had to be carefully worked through.

It became even more of a problem with multiprogramming and several programs occupying memory.

Interestingly for personal computers at home there is almost no problem anymore – memory is so cheap.

With multiprogramming we can swap entire processes out to disk to provide space for others (and swap back in to run).

The disk is known as backing store.

Must be able to hold all memory for all processes.

Swapping is slow – especially if done at every process context switch.

Does the process have to be swapped back in to the same memory space?

Early UNIX used to swap. We still use variants of swapping.

Does it all have to be there?

Overlays provide the hint.

We can execute programs without the entire program being in memory at once.

Can keep either pages or segments on disk when not needed.

The logical address space can be larger than the physical. We call these virtual and real address spaces when we have virtual memory.

This has many advantages:

- unused code doesn't waste physical memory
- we have more memory for multiple processes
- we don't need to load the whole program into memory at once – hence speeding up responsiveness to commands

Why does it work?

Locality of reference

In almost all programs if we look at their memory access over a short period of time (a window) we see that only a small amount of the programs address space is being used.

Each memory access is very probably going to be near another recent memory reference.

True for both code and data.

But it is possible to write programs that don't do this. e.g. arrays stored as rows accessed by columns.

This is known as the principle of *locality of reference*.

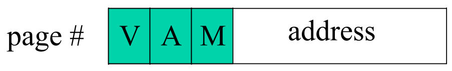
Programs do not reference memory with a random distribution.

See Figure 9.19 for a graphical snapshot of program memory accesses.

Paging

Virtual memory is commonly provided with paged memory.

There are extra bits stored in each page table entry (and some of them in corresponding TLB entries) e.g.



V – valid bit, is the page currently in real memory?

A – access bits – how can this page be accessed, read/write/execute?

M – mode bits – which mode does the processor have to be in when it uses this page

Other bits could be there too (see later)

address – either the frame number or the address on the disk device where this page is currently stored

Work for the OS

So when a page is accessed the page table entry indicates whether the page is currently in real memory or whether it is in a paging file (or swap space) on disk.

The MMU happily takes care of the translation between logical addresses and physical addresses when all pages are in real memory.

If a page is not in real memory it is up to the memory management system to

- allocate real memory for the page
- move pages from disk into memory
- indicate when the page is now ready

To do this several design decisions need to be made.

Moving pages into swap space

Different systems move pages into swap space at different times:

- allocate space for the entire process in swap space (this is usually allocated continuously)
  - this slows down the startup time for processes
  - but it can speed up later operation
  - possible complications as processes grow
- allocate when the page is accessed the first time
  - quick startup
  - all accessed pages have a copy in swap space (even if in real memory as well)
  - new accesses are slowed down
- only allocate when a page is swapped out
  - don’t use swap space at all unless necessary
- only allocate space for changed data
  - code, libraries and read-only data can have their virtual memory in their normal files (requires cooperation between paging system and file system – uniform storage structures)

Demand paging

Demand paging is concerned with when a page gets loaded into real memory.

When a process starts all of its memory can be allocated (and loaded).

- if there is not enough real memory available it has to be taken away from pages currently used
- if there is still not enough some has to go into swap space
- loading a large program can have a severe penalty on other processes in the system (and the overall amount of work done)

Demand paging only brings a page into real memory when the page is used by the process.

- when a process runs it is allocated memory space but it all points to the swap space (or somewhere else)
- actually most demand paging systems do load in the first few pages so that the program can start without lots of page faults (not pure demand paging)

Page faults

If ever a memory access finds the valid bit of the page table entry not set we get a page fault.

- The processor jumps to the page fault handling routine.
- Checks if the page is allocated (if not we have a memory violation).

If allocated (but not in a frame)

- find a free frame (possibly create one)
- read the page from the swap space into the frame
- fix the page table entry to point to the frame

if the page is shared then multiple entries must be fixed

- restart the instruction that caused the fault

instructions must be restartable

See Figure 9.6

The problems of one instruction

Different architectures make the task of paging easier or harder.

Some architectures can access many pages with one instruction.

e.g. add @x, @y

This instruction could cause 6 memory accesses and in the worst case 10 page faults.

The requirement to restart is also a problem

Autoincrement and autodecrement instruction operands need to be restored to their original values before restarting the instruction.

This can be done with extra registers holding initial values or changes.

Block moves are also a problem - part way through the move there is a page fault.

We don’t want to restart the instruction.

Can solve this by checking pages for validity beforehand or maintaining extra registers tracking progress.

Question

I have 8GB of memory in my laptop.

The boot drive (which includes any swap space) is 256GB.

The swap space when I looked was 1.5GB.

The amount of virtual memory reported by the system is more than 300GB.

How is this possible?

Before next time

Read from the textbook

9.5 – Allocation of Frames
9.4 – Page Replacement
9.6 – Thrashing
9.10 – Operating-System Examples
19.3.3.2 – Windows Virtual-Memory Manager