Stop thinking a process image should be stored in a single contiguos block of memory -> paging, segmentation

Not all pages of a process have to be in memory at the same time.

Just a limited num. of pages needed over short period of time.

Store part of the pages in secondary memory (swap space).

When page swapped, use corresponding page table entry to indicate where it can be found.

We can load more processes, even more than physical memory.

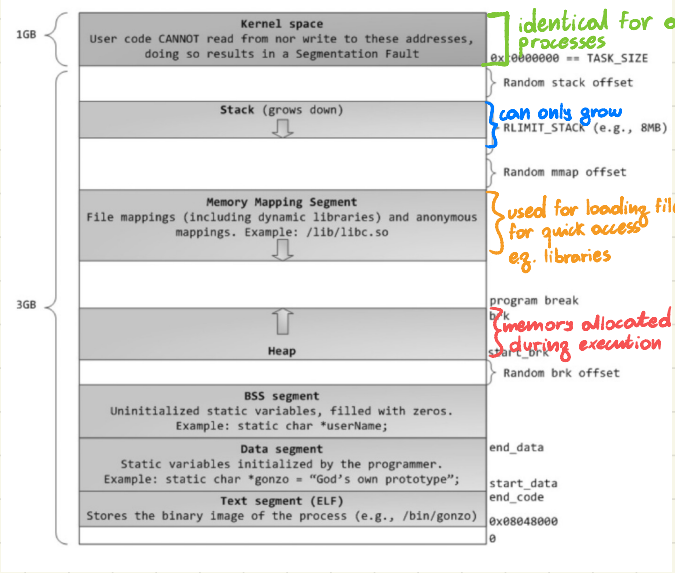
logical addresses created by compiler -> virtual addresses.

When page in memory can immediately access it, otherwise page fault and must load it (done by OS)

To know which page caused it paging HW stores this page number in a register.

Virtual memory layout

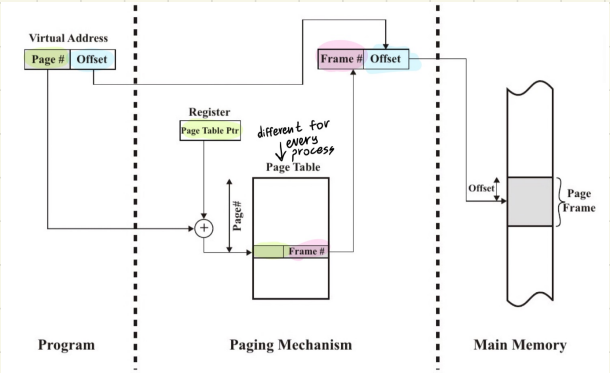
virtual memory space of a process divided in segments



Page table organization

Page table of a process always in memory when it's active

Process must frequently perform address translation, so important that frame number can be found quickly.



start of page table stored in register

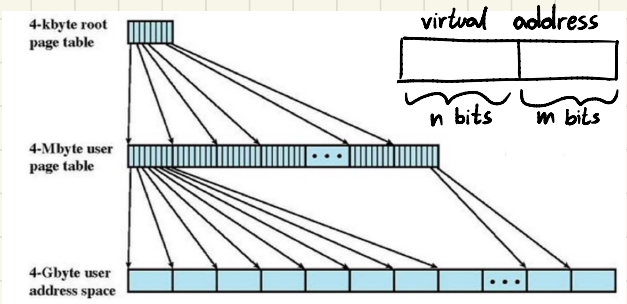
summed page number

obtained frame number, must be concatenated to offset.

Even small processes need big tables

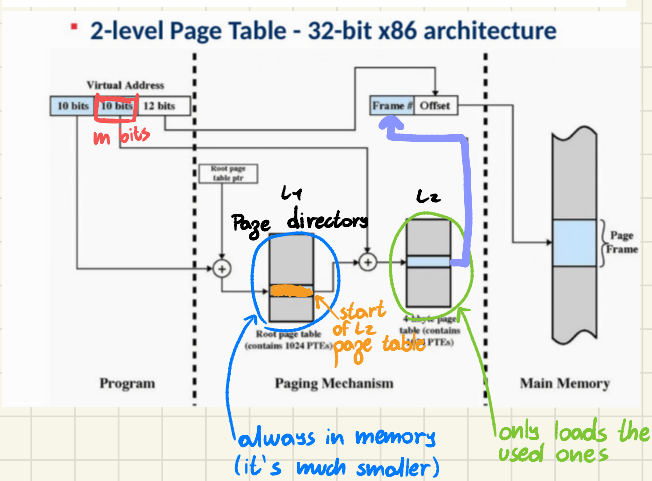
Multilevel page table

reduce size of table



two possible scenarios

1) Required L2 page table present in memory
   we will find a frame number in L1 page entry
   use the m bits as index to L2 page table
   and find frame number, combine offset and form physical address.
   If requested mem. word not in memory page fault and virtual page is loaded into memory, L2 page table modified
2) L2 page not in memory, page fault and load it in memory.
   A second page fault may occur if mem. word not in physical memory.



Disadvantage: single lookup of a frame # requires multiple memory calls

Inverted page table

look up frame number via direct access. We only keep a table with an entry for each physical page (or frame)
Go through all frames to see if one of them corresponds with the requested page of a certain process.
To do it efficiently we use hashing. The page/frame table is extended with an hash anchor table where we want to find the frame number of a certain page of a process.
first run hash f. on this number (and process id). The result will point to a table entry, whose index (can) contains the frame number if the page is in memory.
Could assign to different pairs same entry, they form a linked list (chain)

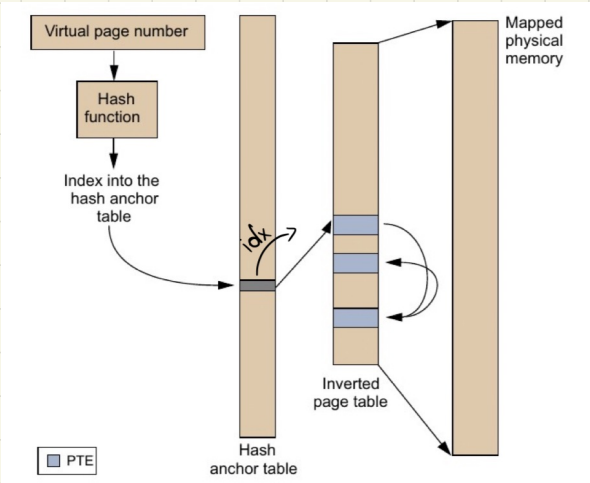
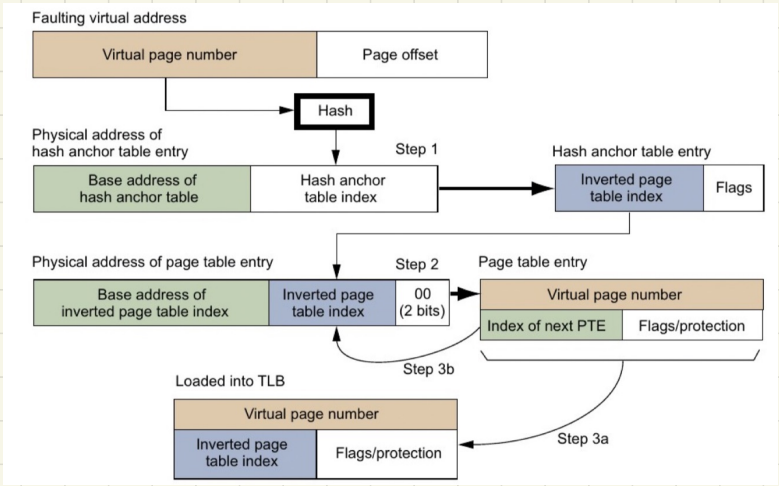


Fig. 31: Inverted page tables [Jacob 98].

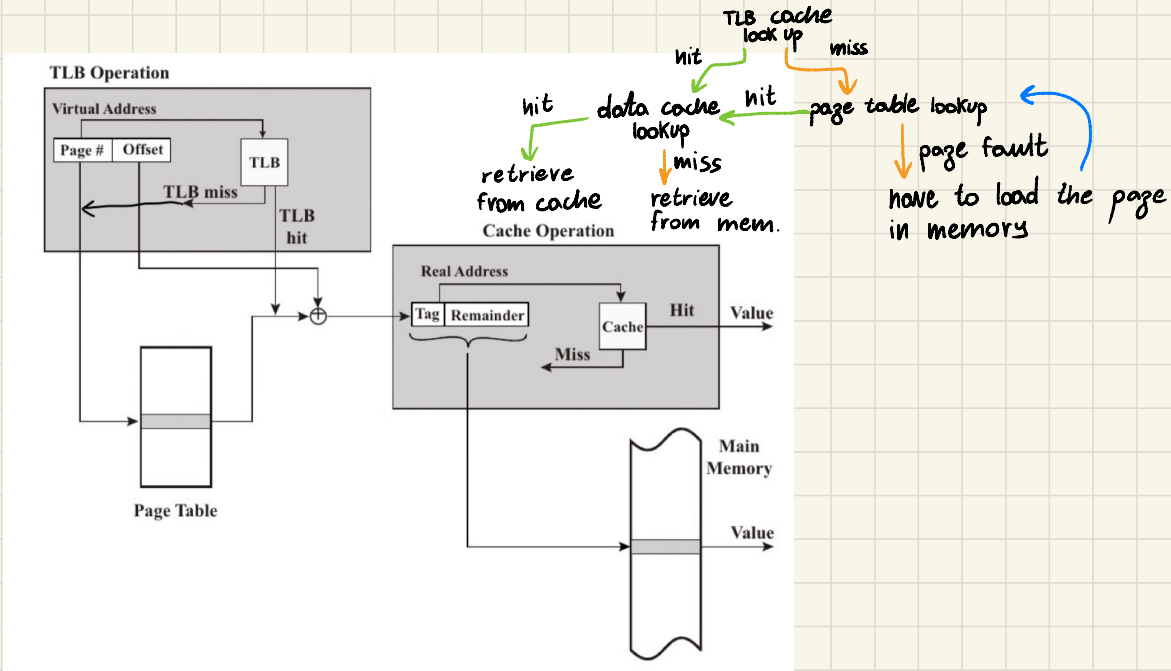


Translation lookaside buffer

Translation is still slow, don't want to do it every time. Use TLB cache

It only contains a small part of the page table entries. Uses reference of locality principle, make available most recently encountered page table entries to processor.

First translation made using page table, result stored in TLB cache.



the TLB cache can be organized like normal cache.

It always contains page table entries of the process currently in execution.

⇓

every process change TLB flush

Software support

Everything is done by hardware, as long as no page fault.

In this case page must be loaded in memory.

Load only that one (demand paging) or some extra (prepaging)?

In which frame do I put the page?

1) How to determine # of pages per process?
2) Can pages occupy frames of other processes?
3) How to choose page to replace?

Page replacement algorithms

reduce page faults

Optimal algorithm (MIN algorithm)

guaranteed to result in minimum # of page faults. Requires perfect knowledge of future events so not implementable. Used as reference point

LRU algorithm

Replace the least recently used page. We can use a simple stack

drawbacks:

- when page hit occurs that page must be moved to MRU position. Have to manage mutual exclusion when have multiple threads/processes
- lot of process power to adjust MRU pos. bcz of virtual memory
- LRU easily replaces pages frequently requested over short period of time and then unused for a while
- Sensitive to scan (long sequence used once)

FIFO algorithm

replace what has been in memory the longest

Implementation easy, can cause Belady's anomaly

Page Requests	3	2	1	0	3	2	4	3	2	1	0	4
Frame 1	3	3	3	0	0	0	4	4	4	4	4	4
Frame 2		2	2	2	3	3	3	3	3	1	1	1
Frame 3			1	1	1	2	2	2	2	2	0	0
Page Requests	3	2	1	0	3	2	4	3	2	1	0	4
Frame 1	3	3	3	3	3	3	4	4	4	4	0	0
Frame 2		2	2	2	2	2	2	3	3	3	3	4
Frame 3			1	1	1	1	1	1	2	2	2	2
Frame 4				0	0	0	0	0	0	1	1	1

(red indicates page fault)

Fig. 34: Belady's anomaly: with FIFO, more memory can lead to more page faults

clock algorithm

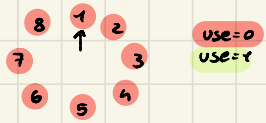
next frame pointer and used bit per page frame

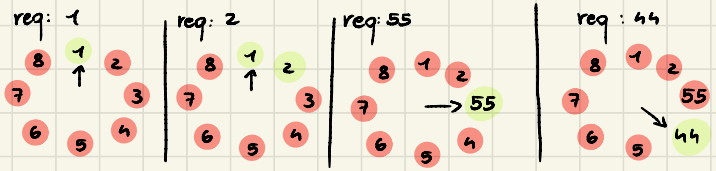
All frames in a circular buffer. Use bit set to 1 when loading new page and when page hit.

When there is a page fault, next frame pointer will loop through the buffer until it finds a frame with use bit 0 and place there the newly requested page (set its use bit to 1)

For each pointer I set use bit to 0 and go to the next one.

Pointer does not reset with each miss, keep going from where previous left off





frequently requested pages remain in memory for a long time

A variant also uses an updated bit.

first traverse buffer to find frame with both updated and use 0, choose it. Otherwise rotate again and search for use=0 and updated=1, switch use bits from 1 to 0.

If no frame found repeat.

Often this is more efficient

Page buffering and segmented FIFO

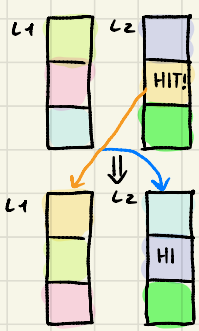
two lists L_1 and L_2 of frames.

L_1 FIFO, L_2 LRU

frame falls out of L_1, goes on top of L_2

frame falls out of L_2, memory released

L_1 hit nothing happens, L_2 hit: move it to L_1



Resident set managment

Which frames are eligible to be replaced?

Some never are (locked), belong to kernel process

A ssstem can have fixed or variable allocation (# of page frames per process).

Some ssstems allow processes to ocupy other processes' page frames, this is called global policy otherwise local replacement policy.

When many processes are active we have