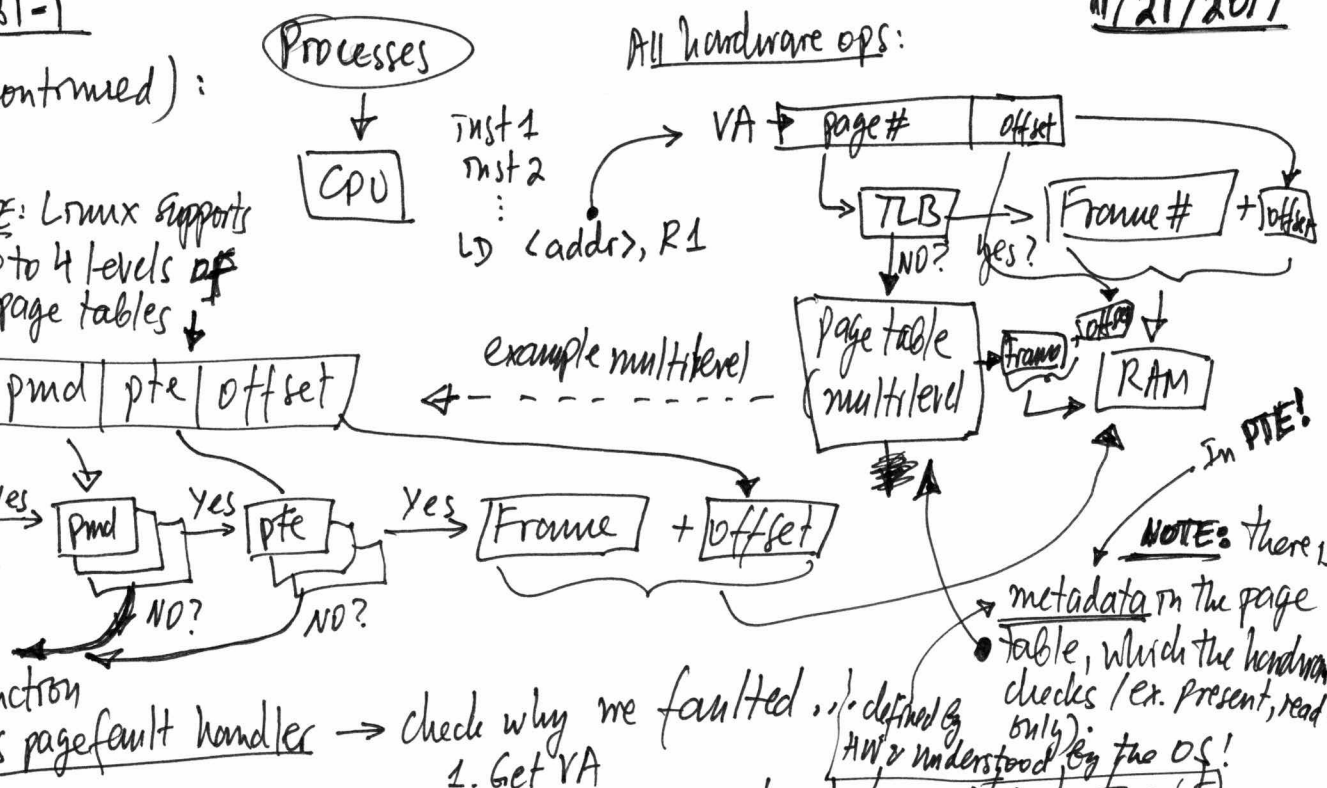
COMSW4118-21-1 11/21/2017

- **Memory** (continued):



(1) Paging — NOTE: Linux supports up to 4 levels of page tables

1. Stalls the instruction
2. Jumps to OS pagefault handler → check why we faulted..
   1. Get VA
   2. Get vm_area_struct / mm_struct for it
   3. If pgd does not exist, create & allocate pmd and point the pgd entry to it
   4. Repeat for pmd & pte to frame #

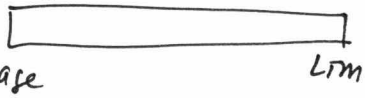
- Frames are obtained from the **Free List** (a list of free pages)

(2) decoupled!

- In parallel, a **swapper process/thread** (dep. on implementation) that runs (ex. kswapd) and checks if the free list is too small and, if so, runs:
- **Page Replacement Algorithm** (ex. second chance) to find and replace frames: for second chance we look at access bits (hw and software bits)
  - if a page is dirty, it will be swapped to disk (if we decide to replace it).
  - then the (now clear) page is added to the free list

(3) In parallel, an **OOM killer** runs and, if the free list is persistently small, chooses and kills processes to reclaim their memory (on some systems may also run when thrashing occurs (page fault rate is high), i.e. each process does not have enough memory, i.e. the amount of memory the OS is less than the working set of the process, the minimum amount of memory the process needs to run "normally" (thus varies by the nature of the process)).

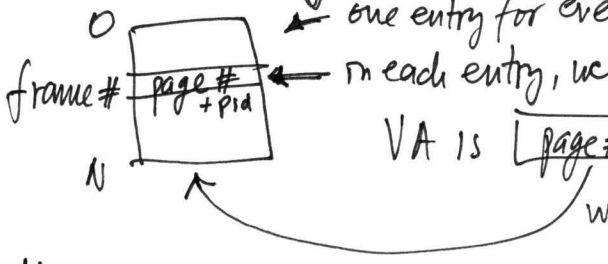
- **Segments** = variable size pages (if supported by hardware only!)
  - Base address + Limit:



Nice thing is the variable size

- We could have a code segment (RO, Exec), data segment, etc.
- **In practice**, segments are not used, even of supported:
  - page calculations are easy because sizes are 2^n and ~~but~~ virtual address can be split
  - for segments, you have a segment # and offset, but calc. is complicated, you need to keep track of which segment is used (since their size is different)
- **x86** was originally a segment based architecture, still supports segment instructions – Linux supports these by assuming each segment starts at 0 and continues till $2^{64}-1$, thus treating each segment as a linear address.
- Segments are looked up in a segment table & all offsets are specific to a given segment.

- **Inverted Page Table** = because most virtual memory in processes is empty anyway.



one entry for every frame of RAM
in each entry, we store the page # (and usually the PID)

VA is [page # | offset]

we find the VA page # in the inverted page table and thus have the frame # to which it corresponds

Benefit: we only need one table for all RAM (not separate page tables for each process)

Hardware supports efficient lookup (context enabled) for the page # in the table.

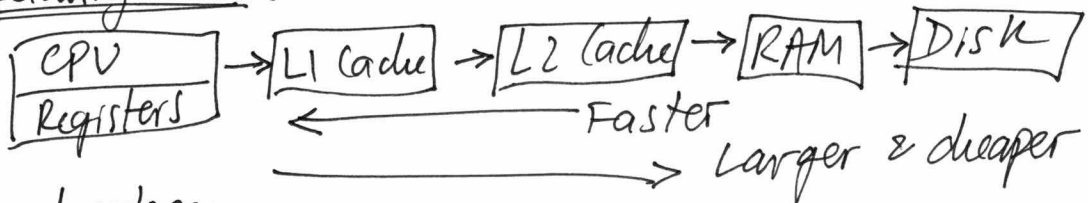
Disadvantage: Can't share memory between processes easily, can't share the kernel portion of each to process to point to the same kernel (as Linux does).

- **Improving Paging Performance**

(1) ▶ **Pre-paging** = since on-demand paging may be too slow w/ many page faults Bring all pages you are sure you will need, at one time in bulk.

(2) ▶ **Use Larger pages** = similar idea, reduces the number of faults

Pre-fetching data is the common to both:



Disadvantage:

Alpha CPU (DEC) → pre-fetched the data into the CPU → BUT, this did not work good in practice since it was consuming the I/O bandwidth of the CPU (meaning the CPU pins themselves!).

- Pre-fetching is a bad idea if your bottleneck is the data pipe itself.

COMSW4118-21-3 11/21/2017

(3) ▶ **Program Structure** = how you access data matters

Say you access an array (2-dim) in doubly-nested loop, first loop i, inner loop j:
X = A[i][j] as opposed to Y = A[j][i]

If we store the array linearly: [0,0 | 0,1 | 0,2 | 0,3 | ....]
Then the X access takes advantage of locality of data! With Y you may fault a lot!
So, depends if arrays are stored Row Major (most often) or Column Major.

● **Disks** are persistent but (traditional ones) have electro-mechanical properties that make them behave differently than memory:

← goal is to avoid Seeks (movement of the arm)

▶ **Locality** is essential to minimize mechanical movement

- File data needs to be grouped together (taken care of by the FileSystem)
- Disk organized in **Blocks** (like pages) ← the "addresses" of the disk and we access by blocks (can't get only 1 byte, you get the whole block, similar to pages)
- File is a bunch of blocks:

[0 | 1 | 2 | 3 | 4 | 5] ← How do we put them on the disk?

→ we can put them **contiguously**, but then how do we know how large the file will end up being (how many blocks)

→ we can have **Linked Blocks:**

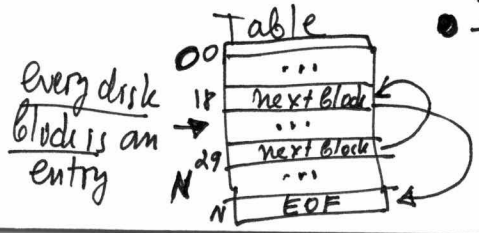
[Block 1] → [Block 2] → [Block 3] → ....

- Link stored in block.
- we can put blocks any place on disk we want.

Turns out the linked scheme is not a good idea:
1) blocks can get dispersed throughout disk
2) very slow for random access! → performance is poor!
3) if you lose a block, you lose everything = reliability is poor!

Reliability is more important topic w/ mechanical devices!

→ **FAT** = File Allocation Table



- It is linked blocks in a table
- If table gets corrupted, you lose everything
- FAT can be backed up, cached in memory, copied and all this can be done easily since this is only the metadata.

- Caching the FAT in memory is a major improvement, since it avoids the seeking when accessing the FAT (a frequent operation).
- Thus, you can get better performance and reliability with FAT.

▶ UNIX (and Linux) does not use any of the above storage approaches.

■