COMSW4118-19-1 11/14/2017

- Page tables are supported by hardware, but the data/mappings are put there by the OS!

For HW5: Linux has huge pages → ignore them for HW5

Only 1 process may track another processe's page tables

1 part: how to expose the page table to the user space

2 part: how to get updates to the page table exposed to the user space RUNTIME!

## Linux Memory Management:

Linux has 4 levels of paging:
(/include/linux/mm_types.h)

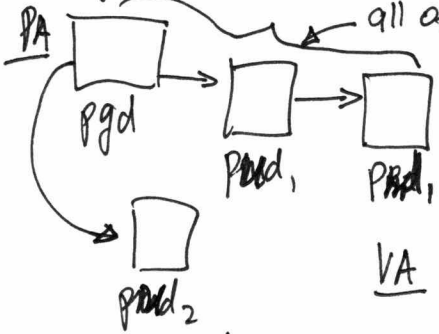
1. mm_struct ⇒ corresponds the the virt. addr. space for a task
2. vm_area_struct
3. page table (pgd, pud, pmd, pte)
4. page

① task_struct → mm_struct
- so, mm_struct is per process & shared b/n threads

Fields: vm_area_struct's

② vm_area_structs describes a VM area,
- those portions of the address space that are in use! Contignous:
  vm_start, vm_end fields
- Linked together in a linked list (sorted!)
- A vm_area_struct may contain multiple pages, but that is not important, it is just a range of addresses.
- Also contained in an RB tree for easier searching!

The PTBR stores physical addresses & all the page redirections are done using physical addresses. For HW5 we will map this to a virtual address (contignous). We will still have frame numbers in that ~~no~~ mapped memory region. With multiple levels in Linux:




NOTE: the above is sparse! You don't need all of these, just the ones that are used.

⇨ EX: when malloc is called it basically requests to use a portion of the VA space, the OS finds that VA space and keeps track of what portions are used in the vm_area_structs

③ pagetable: mm_struct → pgd = the start address of the first-level page table
- when the os changes data in the page table, it goes through the pgd. and, if needed, traverses the page table.

④ page - some what confusing Linux notion → not the same as page in a page table!
- struct that is used to represent/keep track of a frame of physical memory!
- has info like addres, who's using, etc. for each frame in memory! (metadata)

COMSW4118-19-2 11/14/2017

▶ Depending on how you want to access memory, you can use any of the four:
- VA to vm_area_struct mapping
- VA to page table mapping (through pgd → ...)

Example: /mm/memory.c → follow_page_mask()
get a VA, we have to have an mm to use it (since a VA only makes sense in a virtual address space). When you get a frame of memory, you need to update the portions of the page table, so the hardware can map a range of VAs to it. The OS does that update by walking the page table (in software),
**NOTE:** not when you are simply accessing the memory (HW does that), only when allocate/deallocate.

pgd_offset() = finds the entry in the pgd using the mm and VA
pud_offset() = finds the pud entry using the pgd from above and the VA
pmd_offset() = finds the pmd from pud and VA
pte_offset_...() = variety of functions to get the pte from mm, pmd, VA, etc.

VA | pgd | pud | pmd | pte | offset

---

▶ Linux uses the above 4 levels and it is assumed the hardware will support them.
- If hardware supports less, some of the pgd/pud/pmd/pte offset functions will be NOOP, just returning the address passed in (identity ops, rather)

▶ Some Functions:
- *_offset functions = map mm&address to paging view of the VM (the page tables)
- find_vma() = map mm&address to the vm_area_struct
- ~~pfn_to_page()~~ = get to the physical frame number, given a page/pte
  pte_pfn(), virt_to_page() ← from the VA, get to the page metadata of the frame

---

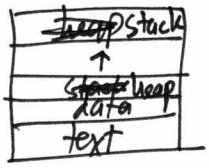
▶ OS gives physical memory to processes, when they need it (malloc, mmap, etc)
- The OS does not actually give the physical space right away. It just marks the request, but it does:
  mmap: map a file to address space, so reading the memory space = reading the file.
- On-demand paging = physical memory is given only when the process that requested it has something to store. This is done at the point when you write the data. Until then, the request operations (malloc, etc.) just setup structures to keep track of the requested memory

➡ When a write op takes place, we translate the VA to a PA (hardware) and, if there is no PA, you get a PAGE FAULT.

▶ **Page Fault** = hardware tries to translate a VA to a PA but fails since there is no entry in the page table.

- An exception that happens within the instruction (like div. by zero)
- Goes to a page fault handler (provided by the OS)
  - ▲ handler inspects the VA
  - ▶ traverses the page table (pgd_offset, pud_offset, pmd_offset, pte_offset)
  - ▲ When a missing entry is encountered, only then is the physical memory allocated (in the form of a page/frame of memory)
  - ▶ then the page table is populated with the new frame mappings
- Instruction that generated the page fault is then restarted.
- Page faults are slow (100s of instructions) → locality helps since previous instructions would have loaded the page. (principle of locality) = overhead gets amortized!
- You can malloc as much as you want, the OS will give it to you in pages. only when it is needed. If you only use 64k out of 10G allocated and the frame size is 8k (typically), the OS will actually only allocate 8 frames.
- Thus, you can run multiple processes w/ huge VA spaces with limited memory.

**NOTE:** The above applies to all process memory:



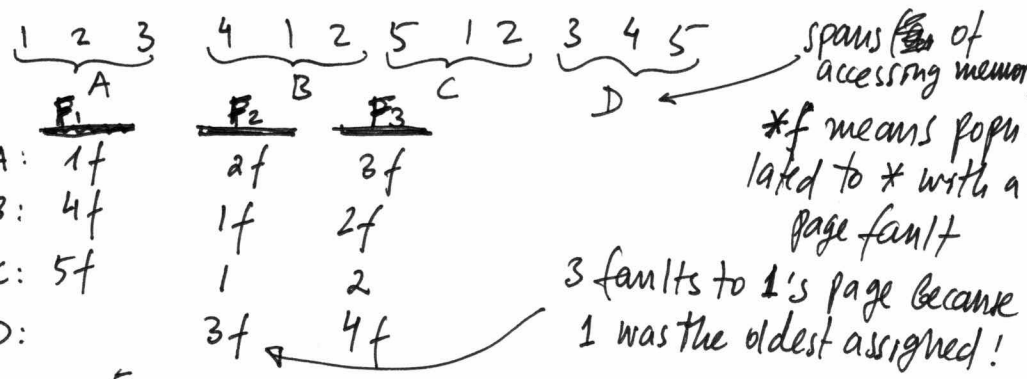

← all areas are allocated w/ the page fault mechanism

EX: When you need stack space w/out phys. memory mapped from it, you will go through a page fault.

**NOTE:** When Linux runs low on memory, it utilizes a low memory killer to kill some processes and free up memory. Generally, this is used to prevent thrashing. With low memory, a lot of page faults are generated from many processes (which is slow) and the system will be hosed up processing page faults (not useful work)
This is **Thrashing** = not enough memory left and system is stuck w/ page faults.

▶ **Page replacement** may be used when memory is low to decide which process can use the memory and which one gives it up (ex. FIFO = first one to obtain is first one to give up).

EX. with FIFO:




NOTE: Many page faults!
**9 total**

Say we add more memory (+1 frame) to above example:

| | $F_1$ | $F_2$ | $F_3$ | $F_4$ |
|---|---|---|---|---|
| A: | 1f | 2f | 3f | 4f |
| B: | 1 | 2 | | |
| C: | 5f | 1f | 2f | 3f |
| D: | 4f | 5f | | |

1 2 3 4 | 1 2 | 5 | 1 2 3 | 4 5
A B C D ← spans of accessing memory

⇦ **NOTE**: We added memory but now we have more faults (10 faults total)

- **Belady's Anomaly** = for a certain types of page replacement algos, adding more memory may increase the page faults. Non-desirable property of a page replacement algo (FIFO has that, as illustrated above).

## ~~Better approach~~ Algo for page replacement

- victimize (replace) the farthest into the future to be used:

| | $F_1$ | $F_2$ | $F_3$ | $F_4$ |
|---|---|---|---|---|
| A: | 1f | 2f | 3f | 4f |
| B: | 1 | 2 | | 5f |
| C: | 1 | 2 | 3 | |
| D: | 4f | | | 5 |

↙ put 5 in 4's frame since 4 is the farthest in future to be used

Only **6 Faults**

But, problem is you don't know the future.
Still, you can use the past to approximate the future:

- **LRU = Least Recently Used** algorithm: ← similarly, you can use LFU (least frequently used).

| | $F_1$ | $F_2$ | $F_3$ | $F_4$ |
|---|---|---|---|---|
| A: | 1f | 2f | 3f | 4f |
| B: | 1 | 2 | 5f | |
| C: | 1 | 2 | | 3f |
| D: | 1 | 2 | 4f | |
| E: | 5f | | | |

since 3 was least recently used.

4 was LRU here

**8 faults total!**

Not as good as knowing the future, but better than FIFO!

(!) In practice, LRU is not used often for page replacement:

WHY?