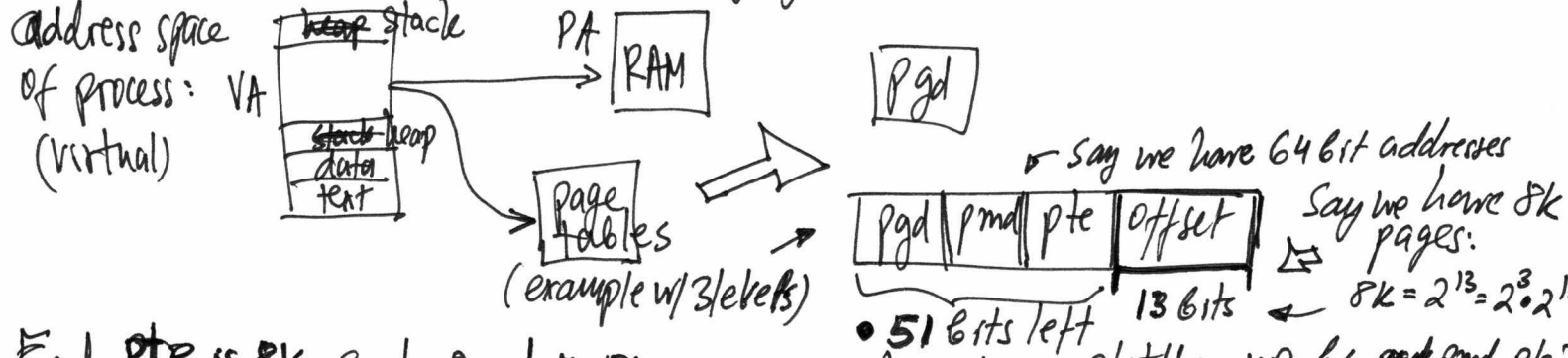
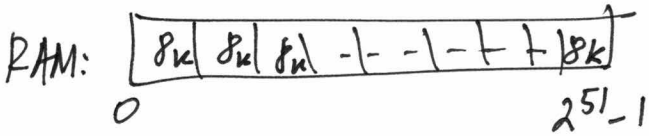
Review of Memory management / Paging:



- Each pte is 8k, each pmd is 8k, each pte is 8k.
- How do we split them up b/n pgd, pmd, pte?
- We dynamically allocate these, so paging!
- Only physical frames are guaranteed to be contiguous chunks of memory.
- With 8k pages we use 13 bits to store the offset and we are left w/ 51 bits for the page tables. NOTE: pgd is NOT paged!
- pmd stores addresses, the pte also stores addresses. With pages of 8K, we have 51 bits that refer to the frame number (the other 13 is offset). This frame number (51 bits) is what we need to store in the pgd, pmd, pte
- But, there is also metadata, so we can approximate that (the frame number size) as 64 bits (51 + metadata) = 8 bytes ← so, we index that
- With an 8k page we can index 1024 8-byte entries = 2^{10} in the VA pgd, pmd, pte parts!
- So, pgd size = 10, pmd size = 10, pte size = 10 in the VA! (in example)
  WHAT IS "LEFT OVER" = 51-10-10
- pgd/pmd/pte store the frame number of each following page table: pgd → pmd → pte. That number is 51 bits for a 64 bit addressing w/ 8k page (64 - 13 for the page offset). THE BASE PHYSICAL ADDRESS
- The RAM is divided (for 64 bit addressing) into 2^{51} pages of size 8k:



- Overall, it works as:

COMSW4118-20-2 11/16/2017

▶ **Metadata** (included in the page tables) ← different than the struct page in Linux these are bits in the pte.

- readonly = to reject writes to the page table (frame, ex: program text.
- present = is the actual address present in memory, i.e. mapped to a frame at the next level (NOTE: Say we use 0 for the address but that is also a valid address)
- dirty = relevant for page replacement (when choosing a frame to victimize), whether something was changed
- access = has the page been accessed recently

NOTE: HW5 page lists some macros that can be used to get to those bits.

(!) **THE PGD IS NOT PAGED!** It is a "starting point" w/ 1 contiguous chunk of memory. Unlike pmd & pte which are in page chunks.

Every process has its own address space & its own page table w/ mappings to RAM. Normally, you would have PTEs for diff. processes pointing to diff frames in RAM.

BUT, you could have them point to the same frame in RAM.

Ex: When you get space in processes via malloc(), the OS will start mapping it to RAM when it is used, different processes will map to different frames.

However: You can use mapping in your address space (VA space) to use a portion of the VA and map anything to it:

▶ **mmap** = take something and put it in the address space

← Ex: map a file to your address space, it can also be shared b/n processes

Similarly, you can take an area of Physical Memory and map it to a process' VA space in multiple processes.

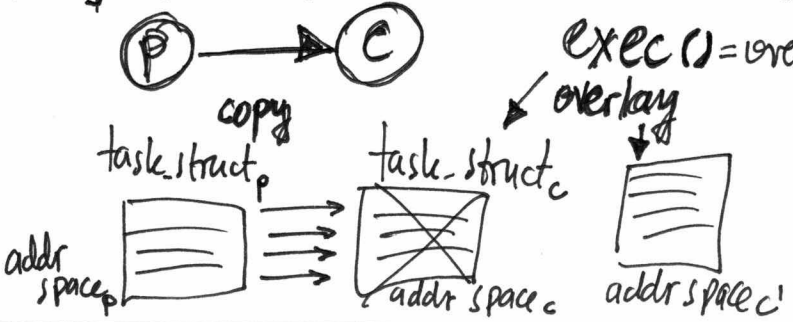
NOTE: Every process has its own page table, thus its own mappings. Threads share both the page table and the mappings.

(!) By 'page table' we mean all of pgd, pmd, pte!

mappings ↑ which may map to the same RAM frame for diff. processes.

▶ **fork()** = copies the addr space? Not really → **IN REALITY**, fork() just sets the child's page table to point to the parent's page table. If the child or parent try to write, only then is the RAM frame copied.

COPY-ON-WRITE (COW)



COMSW4118-20-3 11/16/2017

COW is implemented by setting the readonly page flag (in the pte) and using it with conjunction with the ~~struct page flags and~~ the vm_area_struct flags, which are different than the COW flag. (also has readonly flag)

- If the PTE says readonly but the vm_area_struct says writable then Linux interprets it as COPY ON WRITE. This is triggered by a page fault by the hardware when you try to write to readonly PTE frame - the OS will realize this is not a critical error but COW situation if vm_area_struct says the area is writable.

Linux (3.10 kernel)

- page fault code is architecture specific:

/arch/arm/mm/fault.c → **do_page_fault()**

↳ __do_page_fault()

① find the VMA (using the address and mm) ② check if it was found, ex: if you try to access memory which you did not malloc, i.e. no vm_area_struct for it

③ handle_mm_fault() in /mm/memory.c

① Find the pte (through pgd, pud, pmd) ← any of those can fail

② check if finding the **pud** failed = meaning page fault was because of missing mapping; If so, call pud_alloc(mm, pgd, address) to allocate a pud, if one does not exist (VM_FAULT_OOM if can't be allocated!).

③ repeat for pmd (find or allocate)

④ repeat to pte and call handle_pte_fault() /mm/memory.c

① check if pte entry is present, if not valid this means you are accessing malloc-ed memory page for the first time = get a frame using **do_anonymous_page**(mm, vma, address, pte, pmd, flags)

! "anonymous" memory = generic memory (not mapped to a file), as opposed to memory which is backed by a file

② check if it is a write fault (PTE not writable), if so call do_wp_page() which checks the vm_area flags to see if COW or error.

COMSW4118-20-4 11/16/2017

▶ To deal w/ page faults, go through path:
do_page_fault() → __do_page_fault() → handle_mm_fault() → handle_pte_fault()

Page Replacement (cont. from last time)

▶ FIFO is not very useful
▶ Future Looking algo is optimal!
▶ LRU is an approximation but not used a lot → you need to find the least recently used page → do you update timestamp or resort page each time you access data in it → NOT FEASIBLE EVEN IF HARDWARE SUPPORTS IT! (hardware updates the dirty bit, etc. but can't support a full blown timestamp usually)
▶ SECOND CHANCE ALGO (AKA clock algo)
- there is an access bit (1 bit) which the hardware updates from 0 to 1 when the page is accessed. (in the page table)
- Treat pages with the flag = 1 as recently used and reset the bit to 0.
- If a 0 flag page is found, replace that one
- Otherwise, repeat search and you will find/replace the first one for which the flag was cleared and not reset back to 0 (hence the "clock" name as in two hands chasing each other)

▶ TWO-BIT SECOND CHANCE ALGO
- One access bit set by hardware from 0 to 1 when page accessed
- One software bit to store the access bit before clearing it to 0.

access bit	software bit	
0	0	→ definitely not recently used
0	1	→ kind of "in the middle"
1	0	
1	1	→ definitely recently used

Linux does the 2-bit second chance (at least in the near past)

IN PRACTICE

- you don't really just replace pages when page fault occurs (in page fault, the process is stalled, so replacing the page may be too long)

▶ We use a **Free List** to decouple the processes:
(all free memory (list of frames that are available))
- when there is a page fault because you need more memory, you get it from the free list.
- Separately, there is a daemon (swapd, swapper) that every so often monitors the free list and, if too small, runs page replacement.