COMSW4118-18-1 11/9/2017

● Memory Management (cont.)

Problem: program uses virtual/logical addresses that are mapped to physical addresses by the OS.

Ex: On ARM64, the OS will map 0 to 2^{64} logical addresses for each process (in practice, that is $0-2^{48}$).
Obviously, there are not 2^{64} bytes of physical memory, so mapping is not set for all virtual addresses.

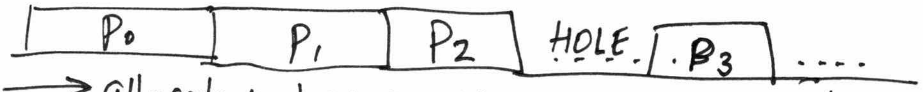
▶ OS decides when & how much of the virtual address space gets mapped to physical.

EX: Dynamically allocate memory, but never use it → OS has no reason to map.
OS gives you memory when you actually have stuff to store!

▶ We will discuss how the OS allocates memory to applications → the OS managing its own memory is a completely separate topic

there are a number of ways to allocate memory:

① Contiguous allocation:



Problem is how to "fill" the holes when new processes show up:
→ allocate contiguous regions as processes show up
→ as you deallocate, you get holes

None optimal, but first fit is usually best
- Ⓐ First fit = first hole big enough for new process to go in
- Ⓑ Best fit = least wasted space after new process goes in
- Ⓒ Worst fit = use the largest hole

Better complexity + no useless holes

may make large holes useless

Both have to visit all holes

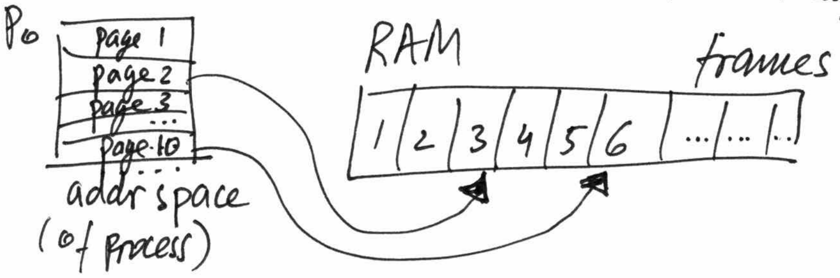
leaves a lot of little spaces that are useless

PROBLEM W/ CONTIGUOUS ALLOCATION: You don't know how much mem. you need / code is static, but stack/heap diff)

② Paged allocation:

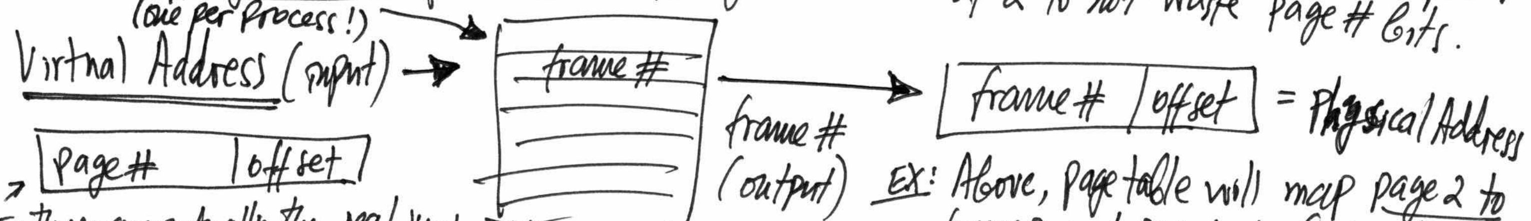
Divide virtual address space ~~and~~ into pages and RAM into frames where their sizes are equal:

sizeof(page) = sizeof(frame)



Map them, as needed. Note: no longer contiguous!

Use a page table for the mapping: ← NOTE: need to select page ~~number~~ ~~size~~ as power of 2 to not waste page # bits.



NOTE: those are actually the real virt. addresses used!

EX: Above, page table will map page 2 to frame 2 and page 10 to frame 6.

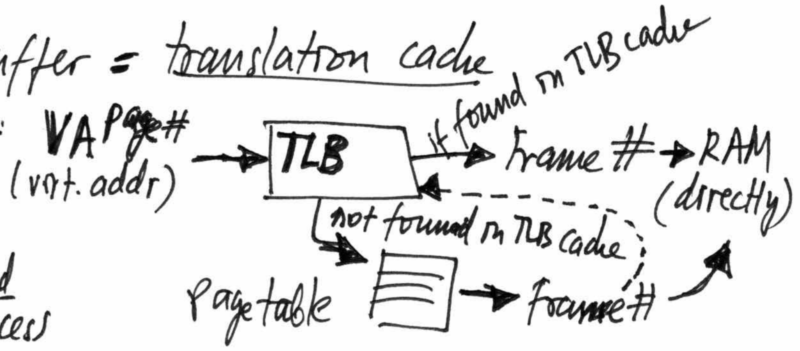
COMSW4118-18-2 11/9/2017

- We store the page table ~~in~~ memory. There is a page table base register (PTBR) that points to the page table. Page table size is known in advance, since RAM size & page size are known in advance.
- Each process has its own page table!
- Each memory access is mapped through the page table in hardware. (Software would be too slow)
- The software (OS) just populates the page table w/ the frame numbers, as ~~they~~ it maps virtual to real memory.
- Issues: (A) you hit memory (RAM) twice for each access: pagetable + actual access

To fix this, we use: ← in hardware!

▶ **TLB** = Translation Look-aside Buffer = translation cache

Caches translations in fast memory:



(!) TLB is per CPU, so there is only one process in its context; TLB is flushed when the CPU is switched to another process

Ex: Say we hit TLB 90% and its access time is 10ns and normal RAM is 100ns: our average access time = 19ns = 90% · 10ns + 10% · 100ns.
W/o paging we access the RAM in **100ns**, w/ paging & TLB it is 119ns approximately!
More accurately: 90% · (10+100) + 10% (10+100+100) = 120ns

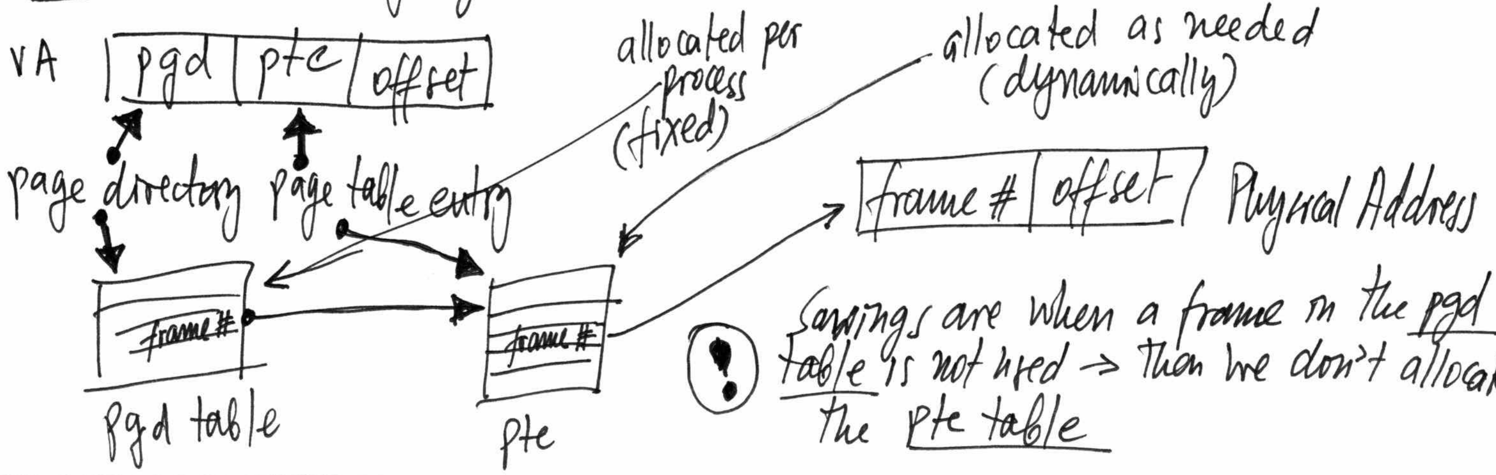
(B) Page table size · # processes = **320 MB** for page tables for 32-bit addresses!
(ex. 16 MB for 32 bit) (say 200)

for 64-bit addresses w/ 1k pages (2^{10} = 10 bits), the page table will be of size 2^{54} and each entry will have 4 bytes, so page table will have 2^{56} bytes, which is huge and per process.

~~ex.~~ Space Problem!

▶ Don't store the whole page table in memory, only store parts we care about: "Page the page table":

- Multi-Level Paging - is one way to do this:



(!) Savings are when a frame in the pgd table is not used → then we don't allocate the pte table

COMSW4118-18-3 11/9/2017

- Memory access is slowed a bit. Consider 3-level paging w/o TLB:

pgd	pmd	pte	offset

→ 400ns = 4 · 100ns

↑ page middle directory

W/ TLB the cost is 119 ns if you hit the TLB and mapping is there. (same TLB example as before) w/ 90% hit rate

On average (assuming 99% TLB hit ~~success~~ rate):

99% (10+100) + 1% (10+300+100) ≈ 114 ns

▶ Linux:

/mm folder , /include/linux/sched.h → task_struct has:

mm_struct *mm

everything associated w/ memory mgmt. (defined in mm_types.h)

```
struct mm_struct {
        struct vm_area_struct  *mmap  ← allocated memory areas of your virtual address space
         pgd_t  *pgd ← base address of first-level page table
         ...
}
```

■