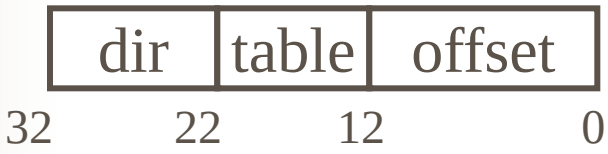
Project 5: Virtual Memory

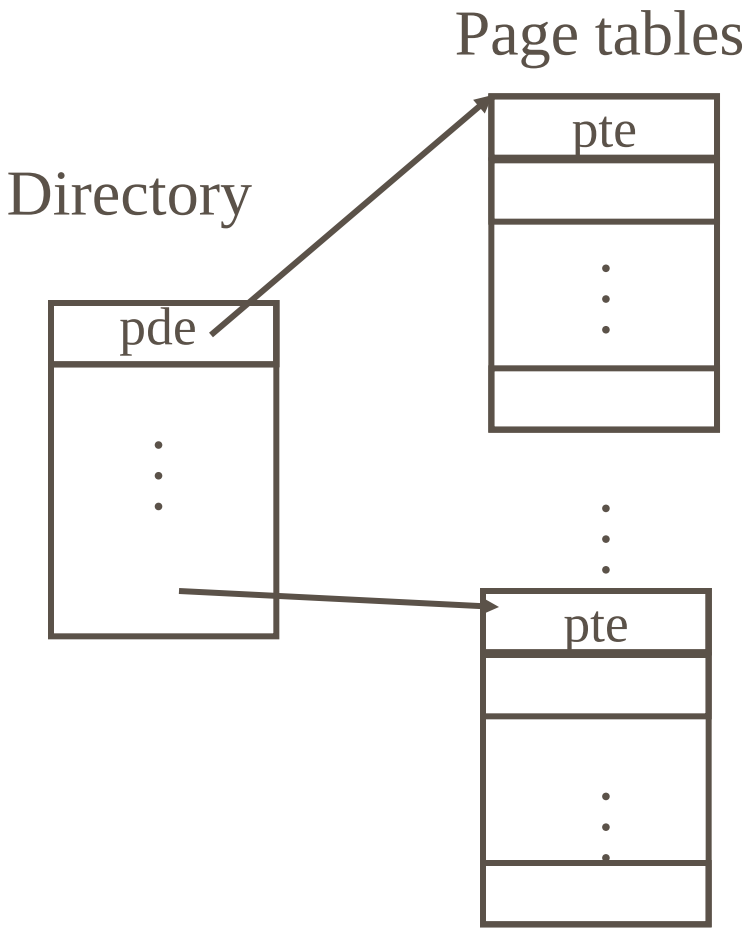
- Two-level page tables
- Page fault handler
- Physical page frame management—page allocation, page replacement, swap in and swap out

Two-Level Page Tables

Virtual address

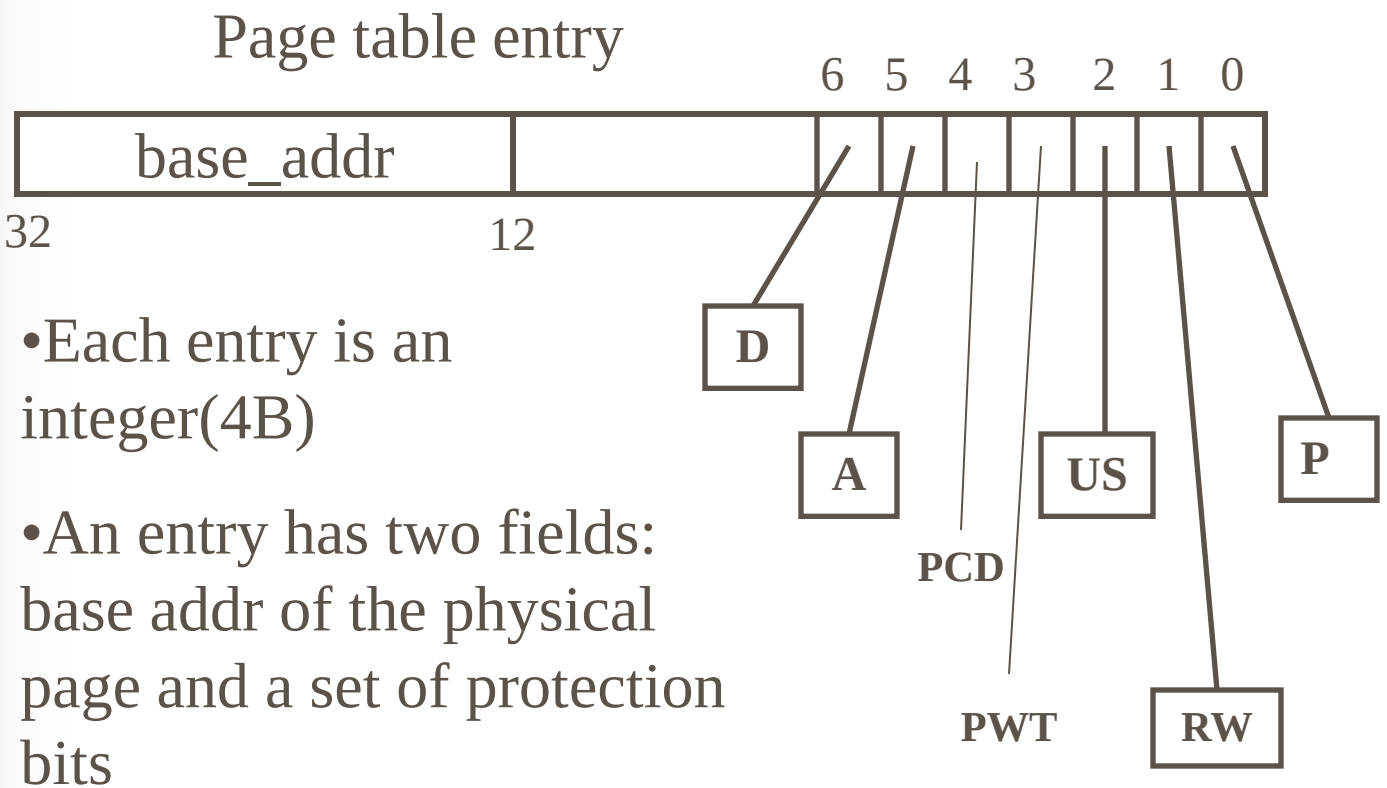


Physical address



- A page is 4KB
- The size of one directory or one page table is 4KB, i.e. one page.
- A directory is like “the page table for the page tables”.

Two-Level Page Tables(cont'd)



- Each entry is an integer(4B)
- An entry has two fields: base addr of the physical page and a set of protection bits
- A directory entry has similar structure

Two-Level Page Tables (cont'd)

- First level: page directory, contains pointers to actual page tables. The size is 4KB, each entry takes 4 bytes, so a page directory points to 1024 page tables.
- Second level: page tables. Each of the 1024 entries points to a page.
- A directory or page table is just a physical page.(So their addresses must be page-aligned)

Protection bits

- Present bit(P): Set if the physical page is in memory
- Read/Write bit(RW): Set if this page can be read/written
- User/Supervisor bit(US): Set if this page can be accessed in user mode. Otherwise the page can only be accessed in supervisor mode

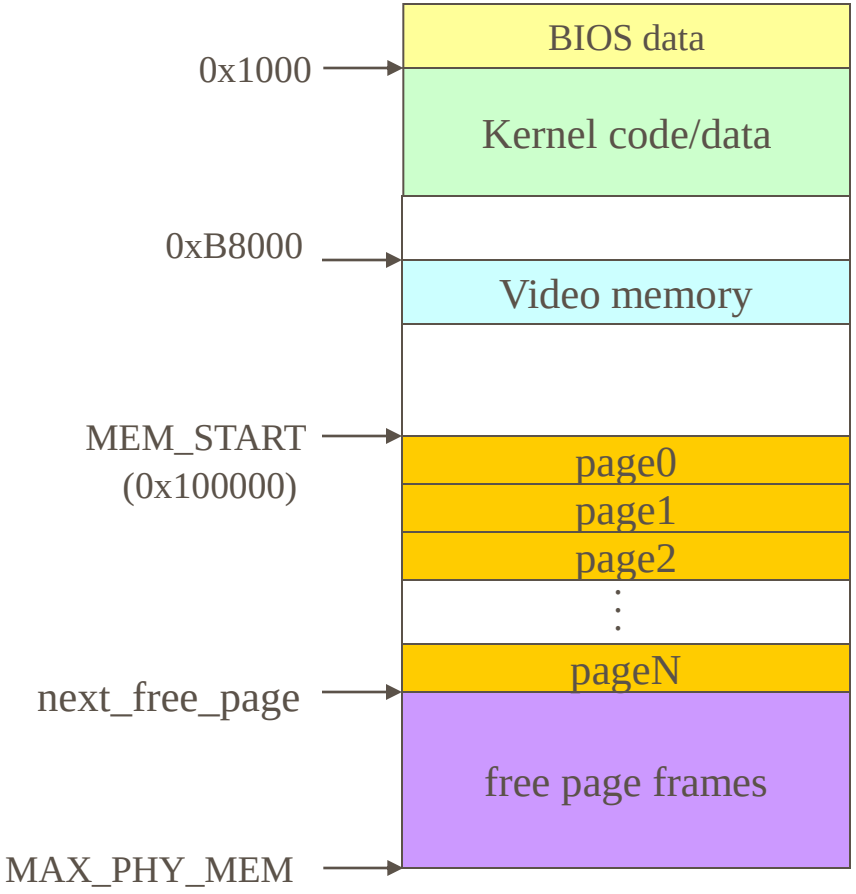
How are page tables used?

- Each process has its own page directory and a set of page tables.
- The address of page directory is in CR3(page directory register) when the process is running.
- CR3 is loaded with pcb->root_page_table at context switch
  - done in given code

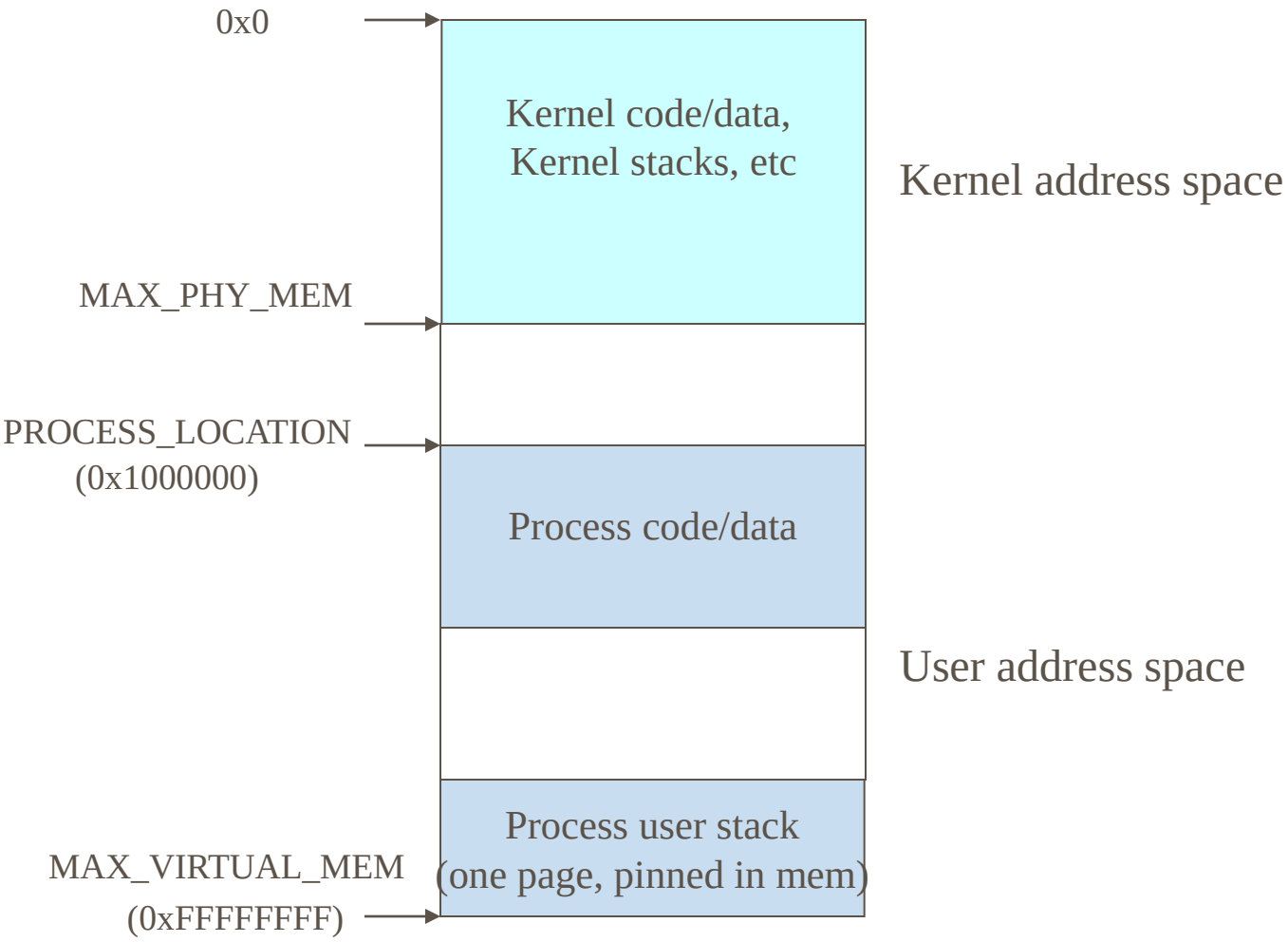
How are page tables used?(cont'd)

- MMU uses CR3 and the first 10 bits of the virtual addr to index into the page directory and find the physical address of the page table we need. Then it uses the next 10 bits of the virtual addr to index into the page table, find the physical address of the actual page. The lowest 12 bits are used as the offset in the page.

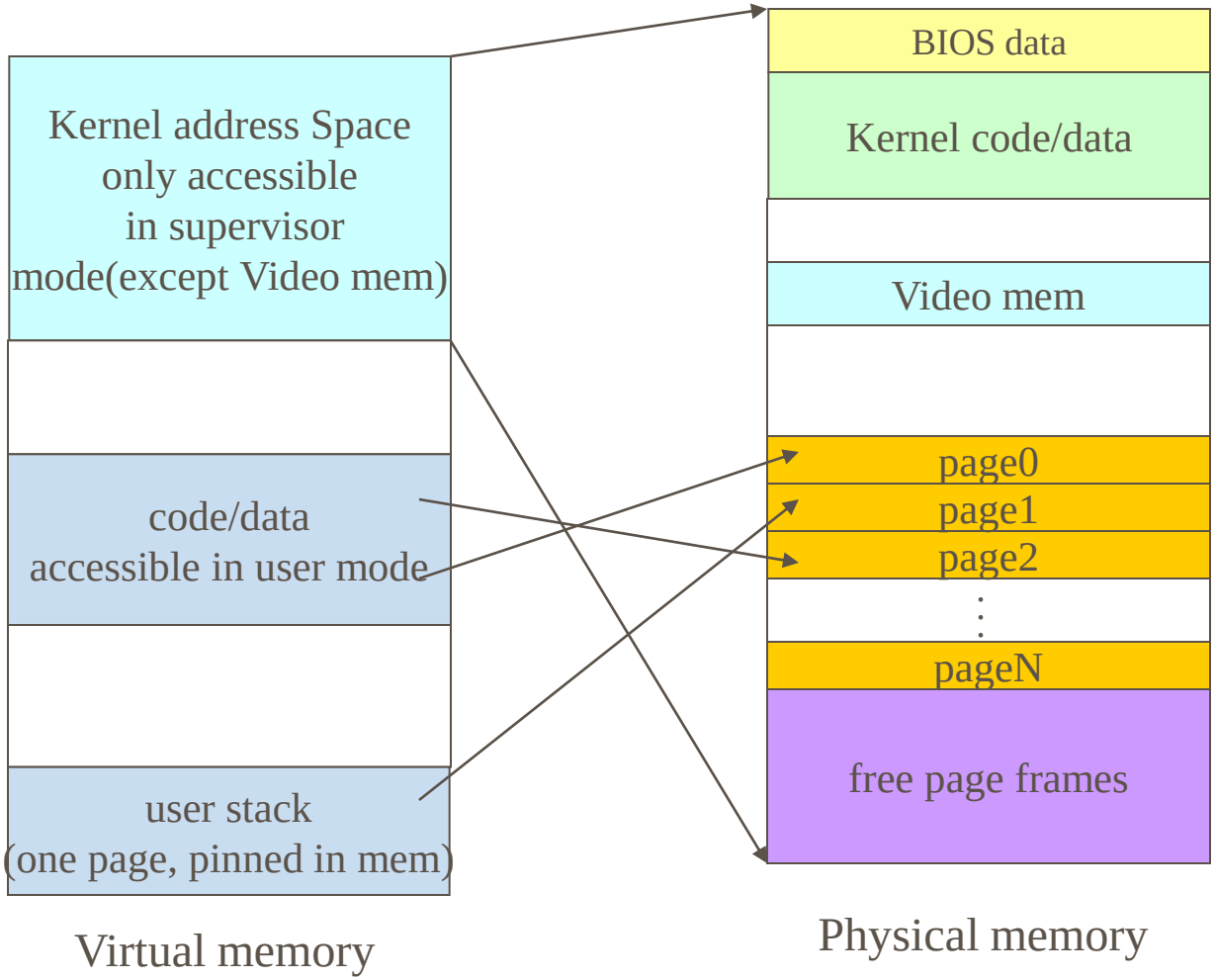
Physical Memory Layout



Virtual Memory (Process) Layout



Virtual-Physical Mapping



Virtual address Mapping

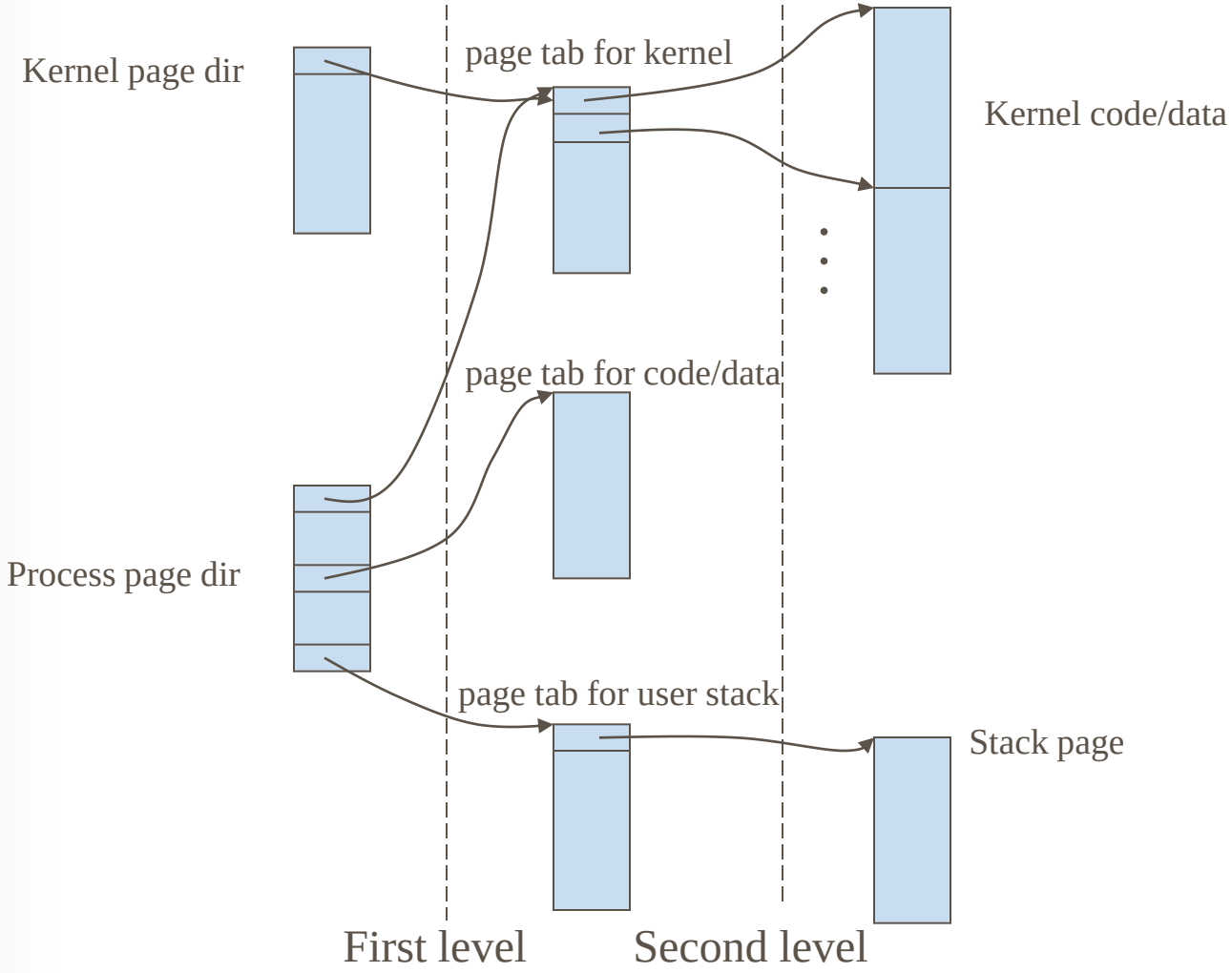
- Kernel addresses are mapped to exactly the same physical addresses
- All threads share the same kernel address space
- Each process has its own address space. It must also map the kernel address space to the same physical address space
  - Why?

Virtual address Mapping(cont'd)

So what do we need to do?

- Setup kernel page tables that are shared by all the threads. (In init_memory())
- Setup process page tables when creating the process (In setup_page_table())

Kernel page tables and Process page tables



Some clarifications:

- It is OK to setup only one page table for each of the following:

  kernel, process' data/code and process' user-stack.

- The page directories and page tables are themselves pages and must be allocated using page_alloc()
- We need only one page for user-stack

Setup Kernel Page Table

- Allocate and pin two physical pages: one for kernel page directory and the other for kernel page table
  - Do we need to allocate pages for kernel code/data?
- Fill in the kernel page table.
  - What value should be filled in the base_addr field and the protection bits?

Setup Kernel Page Table(cont'd)

- Set US bit for video memory area (SCREEN_ADDR in common.h)
  - Why?
  - one page is enough
- Don't forget to map kernel page table into kernel page directory

Set up a Process's Page Tables

- If it's a thread, just store the address of the kernel page directory into the pcb

For processes:

- Allocate and pin four physical pages for each of the following:

  Page directory, page table for code/data, page table for stack, and stack page

Set up a Process's Page Tables(cont'd)

- Map the page tables into the page directory
- Fill in the page table for code/data
  - Which bits should be set?
- Fill in the page table for user stack
  - What values should be filled in here?
- At last, don't forget to store the physical address of the page directory into pcb->root_page_table.

Paging Mechanism

- After init_memory(), the kernel enables paging mode by setting CR0[PG] to one.
  - Done in kernel.c
- In dispatch(), the kernel load CR3 register with current_running->root_page_table
  - Done in scheduler.c

Paging Mechanism(Cont'd)

- When the physical page of a virtual address is not present in memory(the P bit is not set), the MMU hardware will trigger a page fault interrupt(int 14).
- The exception handler saves the faulting virtual address in

  current_running-> fault_addr

  and then calls page_fault_handler()
  - done in interrupt.c

Page Fault Handler

- That's what you are to implement
- Only code/data pages will incur page fault

  all other pages(page directory, page tables, stack page) are pinned in memory
- So assume the page table is always there and go directly to find the corresponding entry for the faulting virtual address.

Page Fault Handler(Cont'd)

- Allocate a physical page

  (Possibly swapping out another page if no free page available)
- Fill in the page_map structure

  (coming soon in the following slides)
- Swap in the page from disk and map the virtual page to the physical page

Physical Page Management—The page_map structure

- Defined in memory.c
- An array that maintains the management information of each physical page. All physical pages are indexed by a page no.
- Fields in each page_map structure
  - The pcb that owns the page
  - Page_aligned virtual address of the page
  - The page table entry that points to this page
  - Pinned or not

Page Allocation

- Implement page_alloc() in memory.c
- A simple page allocation algorithm

  If (there is a free page)

  allocate it

  Else

  swap out a page and allocate it

Page Allocation(Cont'd)

- How do we know whether there is a free page and where it is?

  A pointer is necessary (next_free_page)

- If no free pages, which page to swap out?

  Completely at your discretion, but be careful not to swap out a pinned page

Swap in and Swap out

- From where and to where?

  The process's image on the floppy disk.

  Location and size are stored in pcb->swap_loc and pcb->swap_size
- The floppy_* utilities will be useful
- If the dirty bit (D bit) of the page table entry is clear, do we need to write the page back?

Swap in and Swap out(Cont'd)

- Don't read or write too much

  The images on disk are sector-aligned , but not page-aligned. You should only swap in the data belonging to this process. And be careful not to override other process's image when swapping out.

- Don't forget to modify the protection bits of the corresponding page table entry after swapping in or swapping out

Swap in and Swap out (Cont'd)

- Invalidate TLB entry when swapping out a page.
  - done in memory.c
- So there is no assembly for you to do

Synchronization Issue

- The page map array is accessed and modified by multiple processes during setup_page_table() and page_fault_handler().
- floppy_* operations may call yield()
- So what should we do?

Some clarifications:

- Only the process's code/data pages could be swapped in or out. The following pages are allocated for once and pinned in memory for ever:

  Page directories, page tables, user stack pages

- It is OK not to reclaim the pages when a process exits

Summary

- You need to implement the following three functions in memory.c:

  Init_memory(), setup_page_table(struct pcb_t *), page_fault_handler()

- You need also implement the following auxiliary functions and use them in the above three functions:

  page_alloc(), page_replacement_policy(), page_swap_out(), page_swap_in()

Summary(Cont'd)

- Add whatever other auxiliary functions you need to make your code more readable

Extra Credit

- FIFO replacement policy
- FIFO with second chance
- You may need to modify the page_map structure we provided.
- However, since dead process's pages are not reclaimed. The current page_map structure may suffices

Extra Credit (Cont'd)

- Asynchronous I/O
- Find out where the kernel does busy waiting
- You may need to modify some other files, such as floppy.c
- Please indicate how you achieve it in your README