# *Pintos project 3*

# Project 2-2 is done and before you start project 3 ..

- Make a new branch named “project3” and do your work at that branch.
    - git branch project3
    - git checkout project3

- Without finishing 2-2, you can’t do 3. Please finish 2-2 first.

- **Make commits frequently!**

- **Due**
    - **6/11(Wednesday)**

# Pre-work

- Read pintos manual.
  - Project 3: Virtual Memory
  - https://web.stanford.edu/class/cs140/projects/pintos/pintos_4.html#SEC53

- There is no files in the vm directory.
  - You should write your own files in the vm directory from scratch.

    ( *frame.c, frame.h, page.c, page.h, swap.c, swap.h* )
  - Also, add your own files (*src/Makefile.build*)
  - `$ vi Makefile.build`

```
# No virtual memory code yet.
#vm_SRC = vm/file.c      # Some file.
vm_SRC += vm/page.c
vm_SRC += vm/frame.c
vm_SRC += vm/swap.c
```

# Goal

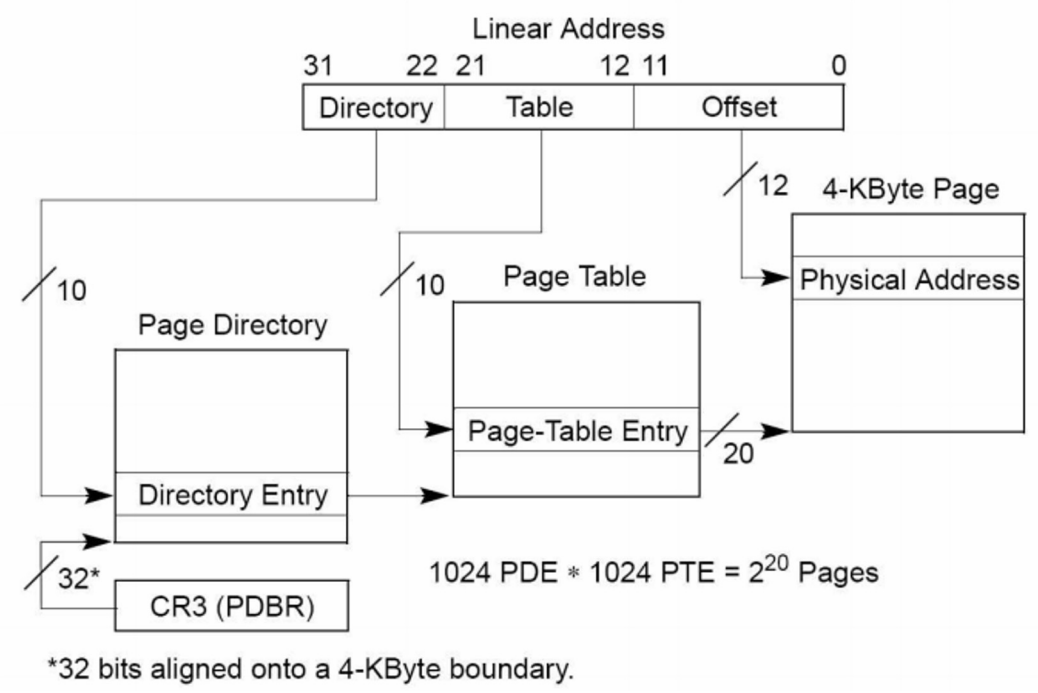
- **Implement Virtual Memory**
  - Page Table Management
    - Page fault handler
    - Eviction policy
  - Swap Disk Management
    - Evicted pages
  - Stack Growth
  - Memory Mapped Files

# Suggested Order of Implementation

- Follow “4.2 Suggested Order of Implementation” in pintos manual
  - Implement Frame Table.
    - See section “4.1.5 Managing the Frame Table”
    - Modify *userprog/process.c* to use your frame table allocator.
    - After this step, your kernel still pass all the project 2 test cases.
    - Do not implement swapping yet
  - Supplemental page table and page fault handler
    - See section “4.1.4 Managing Supplemental Page Table”
    - Modify userprog/process.c to record the necessary information in supplemental page table when loading an executable and setting up its stack
    - Implement loading of code and data segments in the page fault handler
  - Implement Swapping, Stack Growth, Mapped Files

# Page Table Management

- Paging in the x86 architecture



# Page Table Management

- Current Pintos Implementation
  - Use paging
  - Page size: 4KB
  - Each process has its own page tables

  - The page directory is allocated when the process is created
    ( `pagedir_create( )` - *userprog/pagedir.c* )

  - Thread→pagedir points to the page directory
    ( `load()` - *userprog/process.c* )

  - The (secondary) page tables are dynamically created if necessary
    ( `lookup_page()` - *userprog/pagedir.c* )

  - For kernel region, processes have the same mapping
    (`PHYS_BASE ~ 0xffffffff` )

# Page Table Management

- Current Pintos Implementation
  - No demand paging
    - When a process is created, all the contents of code and data segments are read into the physical memory
      ( `load_segment()` – *userprog/process.c* )
  - Fixed stack size
    - Only one stack page is allocated to each process
      ( `setup_stack()` - *userprog/process.c* )
  - No page faults in the user mode
    - Everything needed by each process is in the physical memory

# Lazy Loading (Demand Paging)

- Loading code/data from the executable file
  - ( `load_segment()` – *userprog/process.c* )
- A process do not need all the pages immediately

- Use the executable file as the backing store
  - Only when a page is needed at run time, load the corresponding code/data page into the physical memory
  - Loaded pages will have valid PTEs

- Supplemental page table
  - The page table with additional data about each page
  - On a page fault, find out what data should be there for the faulted virtual page
  - On a process termination, decide what resources to free
  - Any data structure for the supplemental page table
    ( refer to *lib/kernel/hash.c* )

# Lazy Loading (Demand Paging)

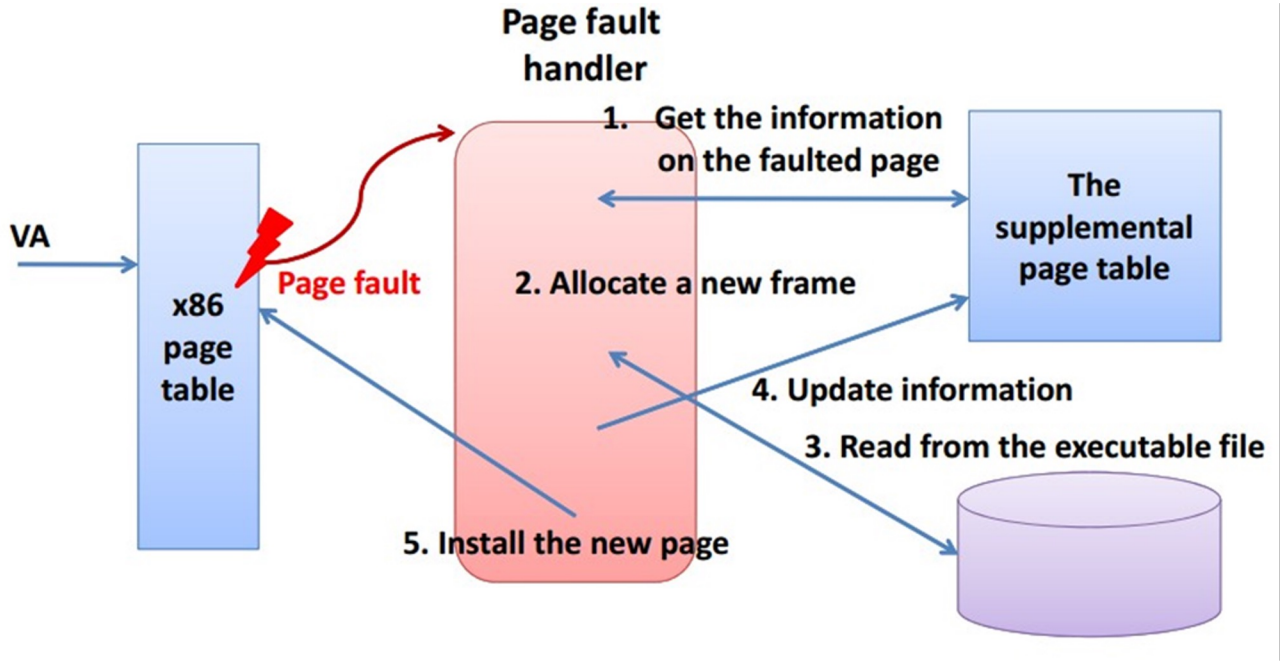
- Page Fault Handler (section 4.1.4)
  - Implement by modifying `page_fault()` in *userprog/exception.c*

  - Locate the page that faulted in the supplemental page table.
  - If the memory reference is valid
    - Obtain a frame to store the page
    - Fetch the data into the frame
    - Point the page table entry for the faulting virtual address to the physical page
      ( use the functions in *userprog/pagedir.c* )

  - Any invalid access terminates the process and thereby frees all of its resources

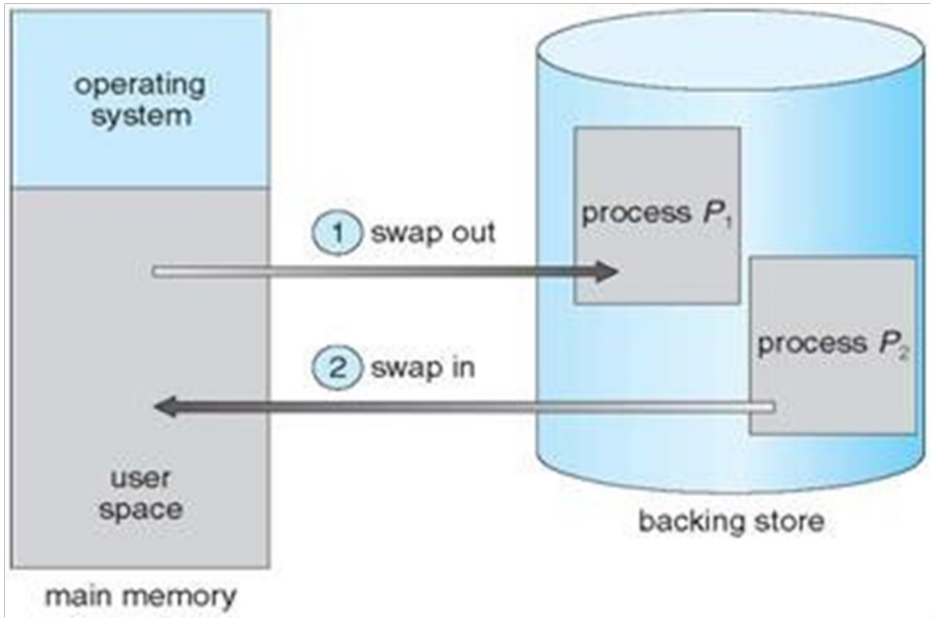
# Lazy Loading (Demand Paging)

- Page Fault Procedure



# Swapping

- Swap disk
  - Used when the amount of physical memory is full
  - If the system needs more memory resources and the RAM is full,
  - Inactive pages in memory are moved to the swap space



# Swapping

- Create Swap disk
  - Use the following command to create an 4 MB swap disk in the *vm/build* directory
  - **`$ pintos -mkdisk swap.dsk 4`**

- Accessing swap disk
  - The swap disk is automatically attached as hd1:1 when you run Pintos
  - Use the disk interface in *devices/disk.h*
    - `struct disk *disk_get (int chan_no, int dev_no);`
    - `disk_sector_t disk_size (struct disk *);`
    - `void disk_read (struct disk *, disk_sector_t, void *);`
    - `void disk_write (struct disk *, disk_sector_t, const void *);`

# Swapping

- A swap disk consists of swap slots
  - A swap slot is a continuous, page-size region of disk space on the swap disk


- Managing swap slots
  - Pick an unused swap slot for evicting a page
  - Free a swap slot when its page is read back or the process is terminated

- Swap table
  - The swap table tracks in-use and free swap slots
  - *lib/kernel/bitmap.c* will be useful

# Swapping

- Page replacement policy
  - Implement global page replacement algorithm that approximates LRU
  - The “second chance” or “clock” algorithm

- Get/Clear Accessed and Dirty bits in the PTE
  - In *userprog/pagedir.c*,
  - `pagedir_is_dirty(), pagedir_set_dirty()`
  - `pagedir_is_accessed(), pagedir_set_accessed()`

# Swapping

- Frame Table
  - Allows efficient implementation of eviction policy
    - Use the frame table while you choose a victim page to evict when no frames are free
  - One entry for each frame that contains a user page

- Frame Allocation
  - `palloc_get_page()` and `palloc_free_page()` in *threads/palloc.c*
  - If there are free frames in the user pool, allocate one by calling `palloc_get_page()`
  - If none is free
    - Choose a victim page using your page replacement policy
    - Remove references to the frame from any page table that refers to it
    - If the frame is modified, write the page to the file system or to the swap disk
    - Return the frame

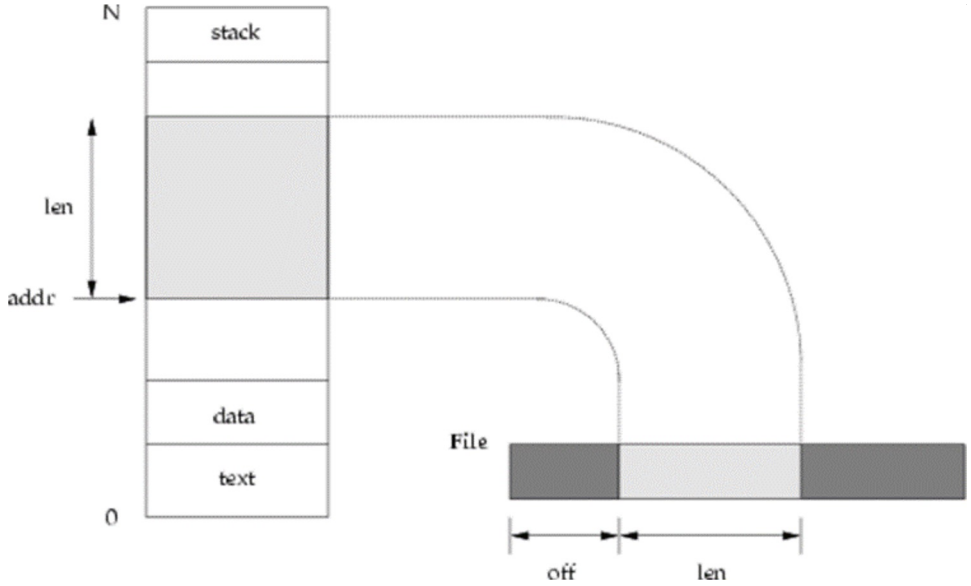
# Stack Growth

- Growing the stack segment
  - Grow the stack dynamically, as page faults occur
  - Identify stack accesses by looking at the `esp` of the `struct intr_frame`
    - When the page fault occurred in the user mode
      - Use `(struct intr_frame * ) f->esp`
    - When the page fault occurred in the kernel mode
      - Save `esp` into struct thread on the initial transition from user to kernel mode

- Set an absolute limit on stack size, 8 MB

# Memory Mapped Files

- Memory mapped file
  - Map files into process address space
  - On page fault, load file data into page
  - When swapping, write file pages back to disk



# Memory Mapped Files

- Implement `mmap()` System calls ( In *userprog/syscall.c* )
  - `mapid_t mmap (int fd, void *addr)`
    - Need to connect mapids to actual files
    - `mmap()` fails if
      - fd is 0 or 1
      - The file has a length of zero bytes
      - addr is 0
      - addr is not page-aligned
      - The range of pages mapped overlaps any exisitng set of mapped pages
  - `void munmap (mapid_t mapping)`
    - All mapped files should be unmapped when a process exits

# Grading

- Code (pintos grade) : 90%
  - **Pass all test cases**

- Design document : 10%
  - Answer questions:
  - Please answer as many questions as you can even if you can’t implement

# About questions

- We will not answer the question about detailed implementation and code, etc.
- Please ask questions only related to environment and high-level design of project.
- Please read pintos manual carefully.

# *Thank you*


22