# Operating Systems Lab

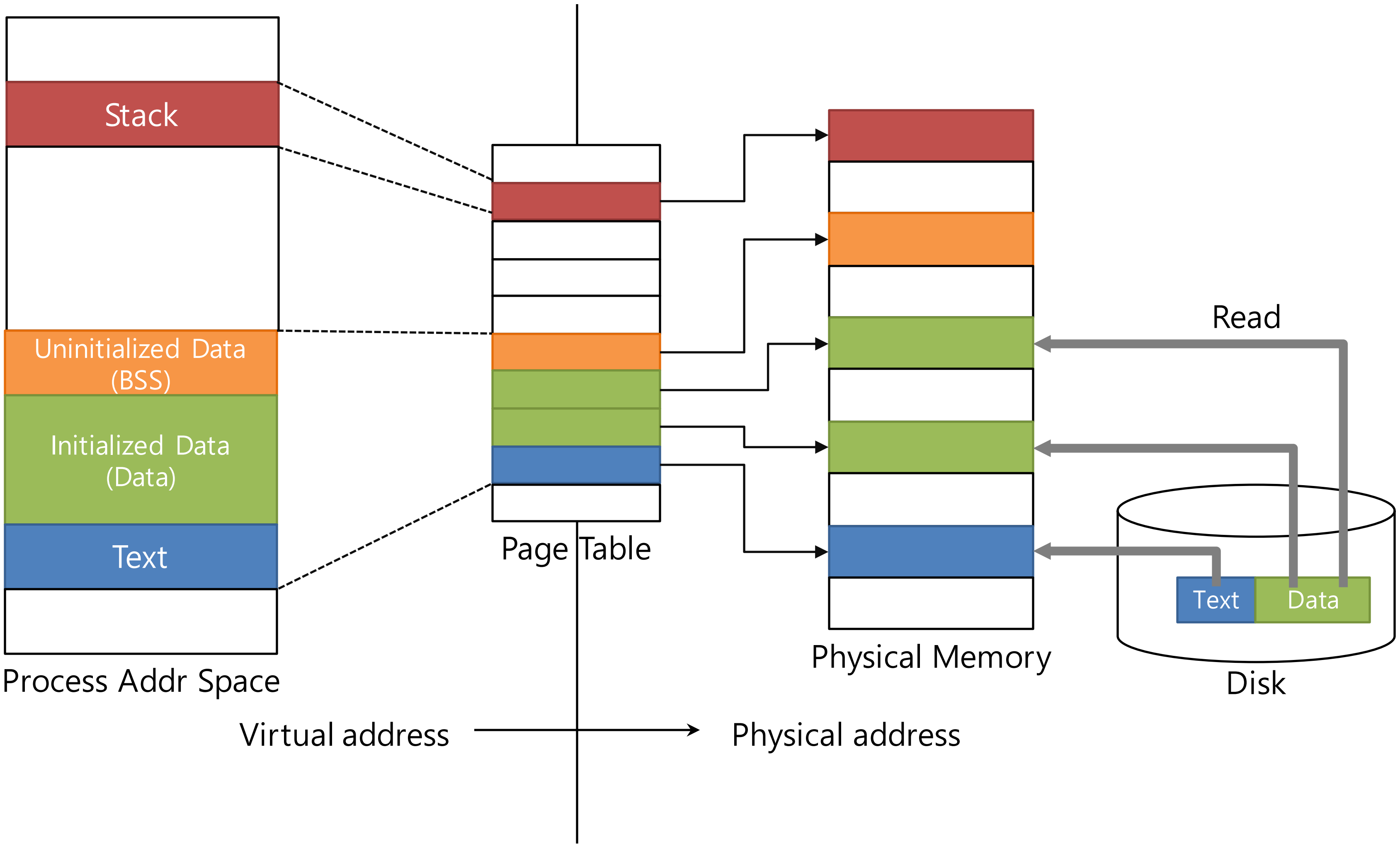
# Part 3: Virtual Memory

KAIST EE

**Youjip Won**

# Address space of process in Pintos

- Pintos memory layout before project



# Virtual Memory

- Entire executable file is loaded at once at the beginning.
- Physical addresses of each page in address space are fixed at the beginning of ‘`fork/exec`’.
- Result

**Implement “Virtual Address”.**

# To Do’ s

**Implement “Virtual Address”.**

- Enable Demand paging/Swapping.
- Enable Stack Growth.
  - Dynamic page allocation for page fault on stack
- Implement Memory mapped file.
  - Implement `mmap()` and `munmap()`.
  - For a physical page, differentiate file_backed page and anonymous page.
- Enable Accessing User Memory.

# Demand Paging

# Basics

- Virtual page: Virtual Page number (20 bit) + page offset (12bit)
- Page frame: physical frame number (20 bit) + page offset (12 bit)
- Page table:
  - VPN → PFN
  - It is hardware.
- Swap space: array of page sized blocks

# A page in virtual address space

- Load the page from the disk as requested.
- A page in VM can be either in-memory only or part of a file.
  - text: part of file
  - Data: part of file
  - BSS: in memory
  - Stack: in memory
  - Heap: in memory
  - `mmap()`ed region: part of file

# Page fault in current Pintos

```
Userprog/exception.c

static void
page_fault (struct intr_frame *f)
{
…

  /* To implement virtual memory, delete the rest of the function
     body, and replace it with code that brings in the page to
     which fault_addr refers. */
  printf ("Page fault at %p: %s error %s page in %s context.\n",
          fault_addr,
          not_present ? "not present" : "rights violation",
          write ? "writing" : "reading",
          user ? "user" : "kernel");
  kill (f);
}
```

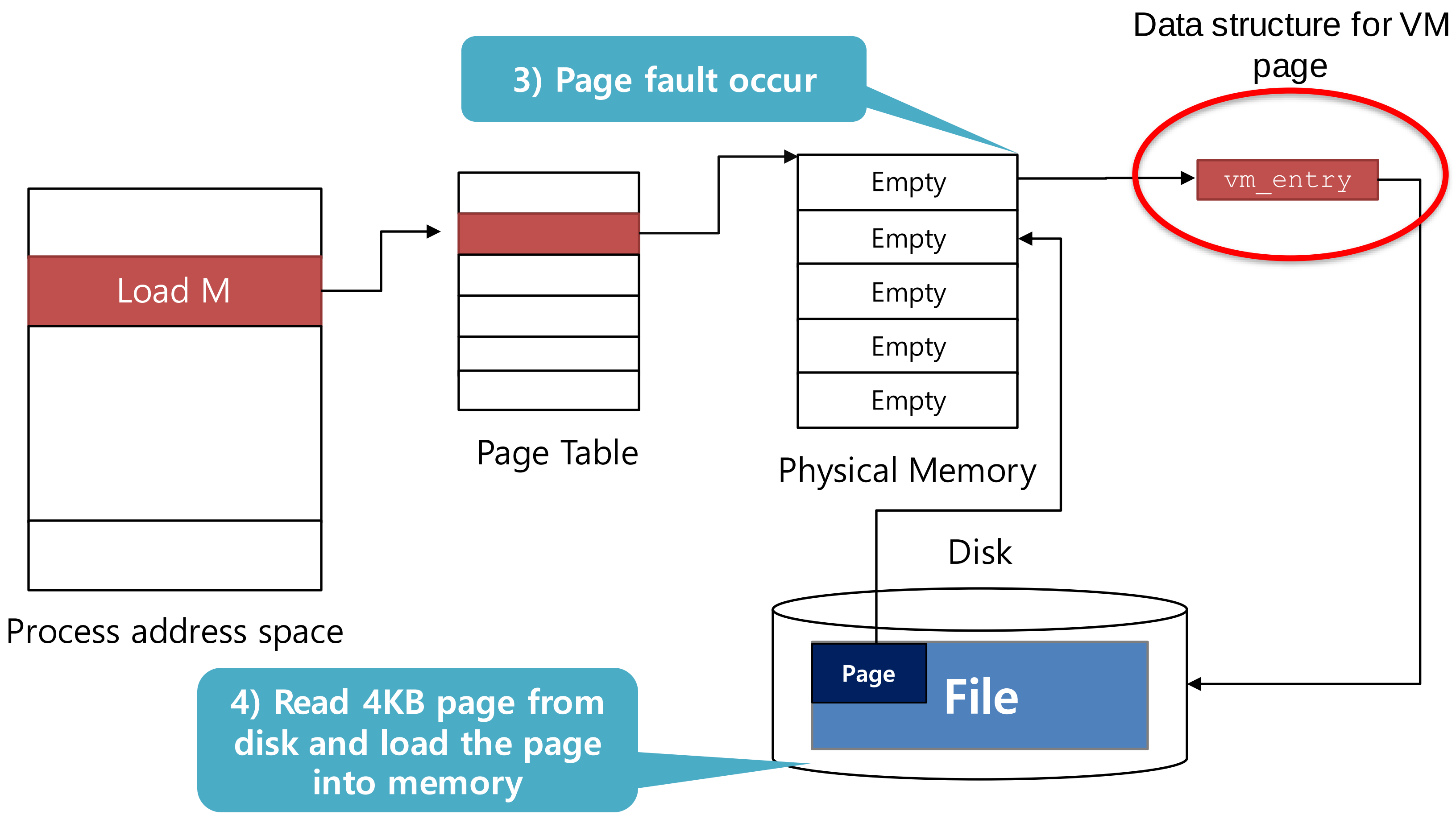
# Page fault in  Pintos with VM

- When page fault occurs? Modify the `page_fault` function
  - Check if the memory reference is valid.
    - → locate the content that needs to go into the virtual memory page
    - → from the file, from the swap or can simply be all-zero page.
  - For shared page, the page can be already in the page frame, but not in the page table
  - Invalid access → kill the process
    - Not valid user address
    - Kernel address
    - Permission error (attempt to write to the read-only page)
  - Allocate page frame.
  - Fetch the data from the disk to the page frame.
  - Update page table.

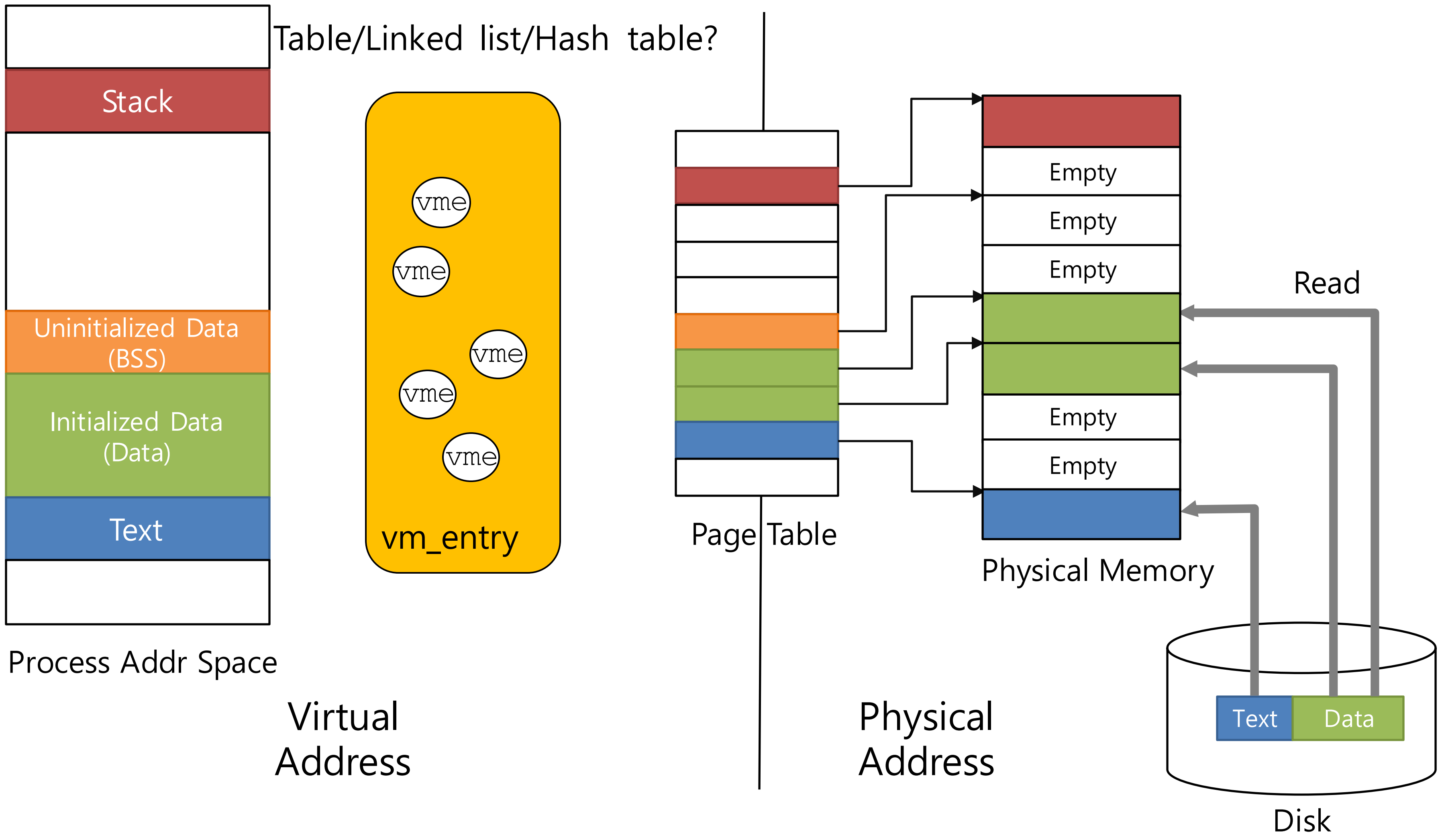
# We need additional information for a virtual page

- Virtual page number
- Read/write permission
- Type of virtual page
  - a page of ELF executable file
  - a page of general file
  - a page of swap area
- Reference to the file object and offset(memory mapped file)
- Amount of data in the page
- Location in the swap area
- In-memory flag: is it in memory?

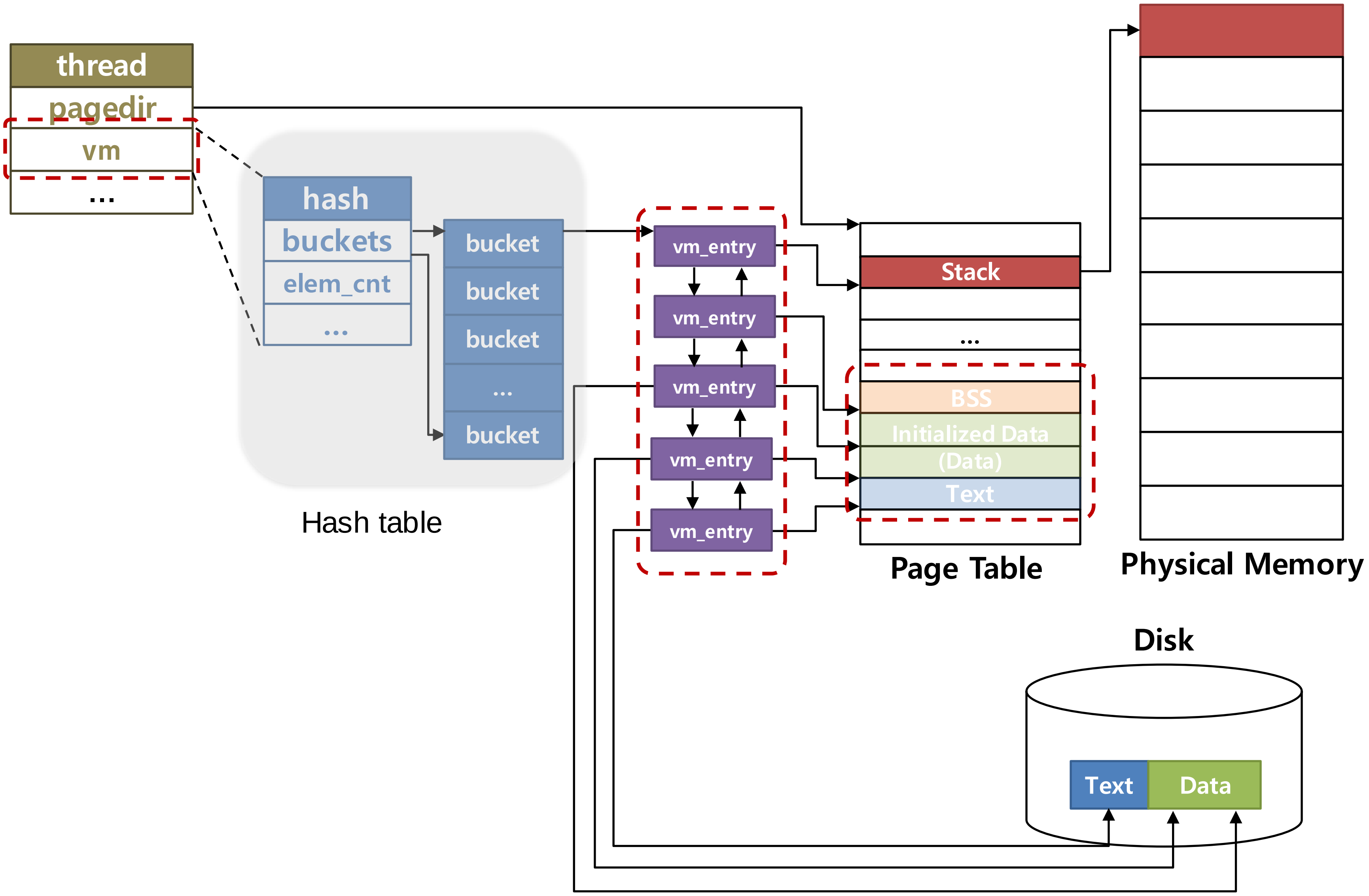
# vm_entry


Data structure for VM page
vm_entry
3) Page fault occur
Load M
Process address space
Page Table
Empty
Empty
Empty
Empty
Empty
Physical Memory
Disk
Page
File
4) Read 4KB page from disk and load the page into memory

# A set of virtual pages for a process: a set of vm_entry



# Address Space in Pintos with VM



# vm_entry

pintos/src/vm/page.h

```
struct vm_entry{


// fill this out.


}
```

- Organize the vm_entry: Hash table(src/lib/kernel/hash.*), linked list, or etc.

# Add vm_entry set to thread structure

```
struct thread
```

Since virtual address space is allocated for each process, define the hash table to manage virtual pages.

pintos/src/threads/thread.h

```
struct thread{
    /* Owned by thread.c. */
    tid_t tid;                        /* Thread identifier. */
    enum thread_status status;        /* Thread state. */
    ...
    /* Owned by thread.c. */
    unsigned magic;                   /* Detects stack overflow. */

    struct hash vm;  /*Hash table to manage virtual address space of thread*/
}
```

# Modify `start_process()`

pintos/src/userprog/process.c

```
static void start_process (void *file_name_)
{
        ...

/* Initializing the set of vm_entries, e.g. hash table */

/* Initialize interrupt frame and load executable */
        memset (&if_, 0, sizeof if_);
        ...
}
```

# Modify `exit()`

- remove vm_entries when the process exits.

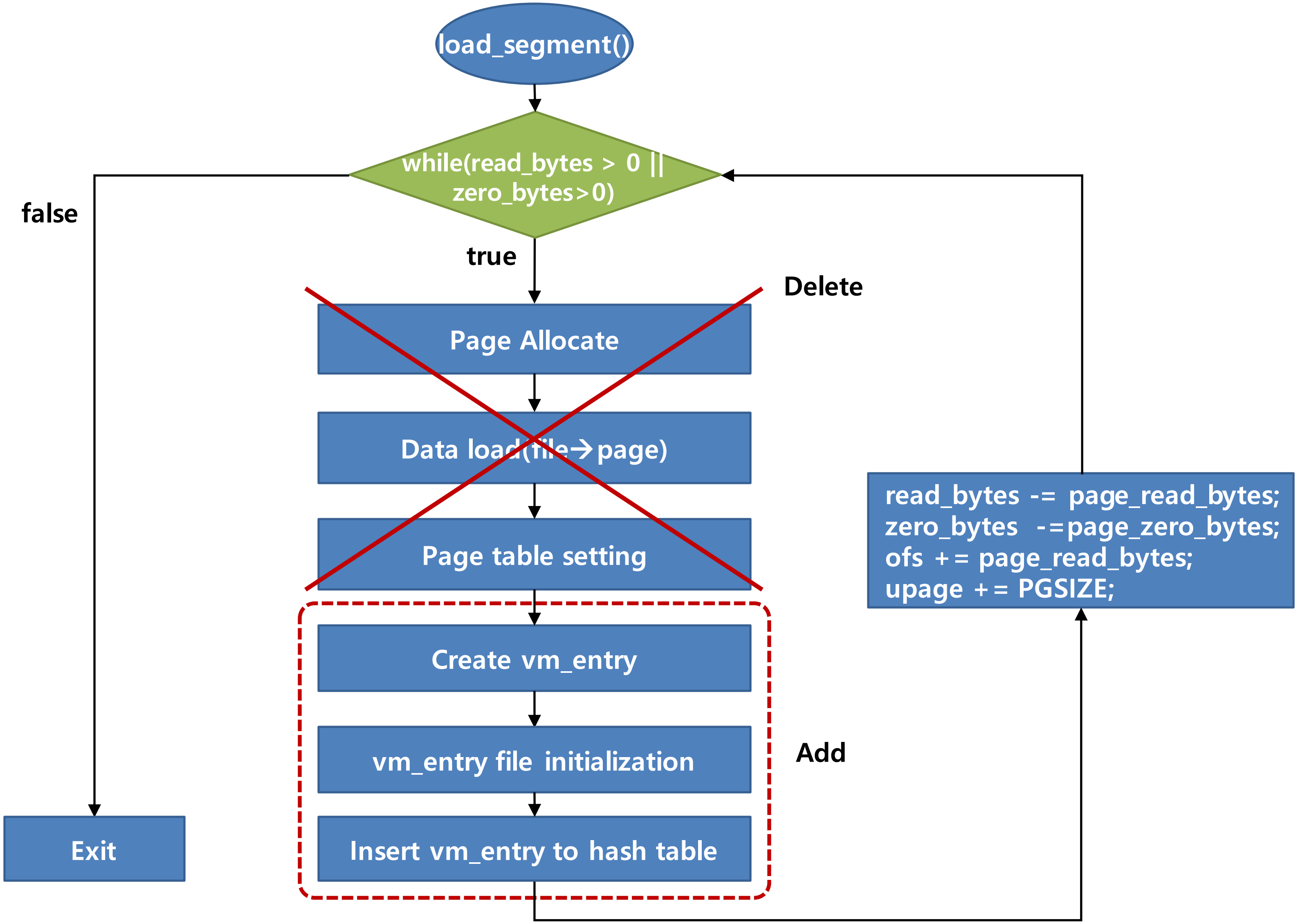
pintos/src/userporg/process.c

```
void process_exit (void){
    struct thread *cur = thread_current();
    uint32_t *pd;
    ...
    palloc_free_page(cur -> fd);
    /* Add vm_entry delete function */
    pd = cur->pagedir;
    ...
}
```

# Address Space Initialization

- Original Pintos: Allocate physical memory by reading all ELF image.
  - Read Data and code segment by load_segment().
  - Allocate physical page of stack by setup_stack().
- Pintos with VM
  - Allocate page table: all entries are invalid.(not mapped).
  - Allocate vm_entry for each page instead of allocating of physical memory.
  - Modify `load_segment()`.
    - Add a function that initializes structures related to virtual address space.
      - Remove the following: loading the binary file to virtual address space.
      - Add the followings.
        - allocate vm_entry structure.
        - Initialize the field values.
        - insert it to the hash table.

# Modify `load_segment()`



```
read_bytes -= page_read_bytes;
zero_bytes -=page_zero_bytes;
ofs += page_read_bytes;
upage += PGSIZE;
```

# Modify `load_segment()`

pintos/src/userprog/process.c

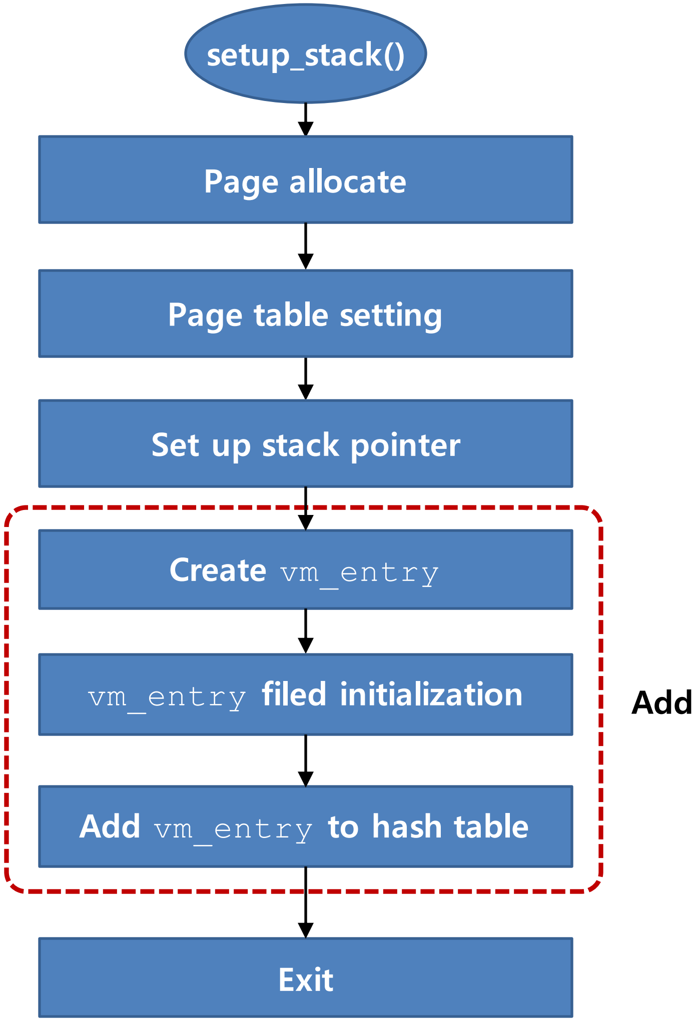
```
static bool load_segment (struct file *file, off_t ofs, uint8_t *upage,
                          uint32_t read_bytes, uint32_t zero_bytes, bool writable)
{
        ...
        while (read_bytes > 0 || zero_bytes > 0)
        {
                size_t page_read_byters = read_bytes < PGSIZE
                                                          ? read_bytes : PGSIZE;
                size_t page_zero_bytes = PGSIZE - page_read_bytes;

                .....

                /* Create vm_entry(Use malloc) */
                /* Setting vm_entry members, offset and size of file to read
when virtual page is required, zero byte to pad at the end, … */
                /* Add vm_entry to hash table by insert_vme() */

                read_bytes -= page_read_bytes;
                zero_bytes -= page_zero_bytes;
                ofs += page_read_bytes;
                upage += PGSIZE;
        }
```

Delete allocating and mapping physical page part

# Modify stack initialization function

- Original
  - Allocate a single page
  - Page table setting
  - Stack pointer(esp) setting

- Add
  - Create `vm_entry` of 4KB stack
  - Initialize created `vm_entry` field value
  - Insert vm hash table

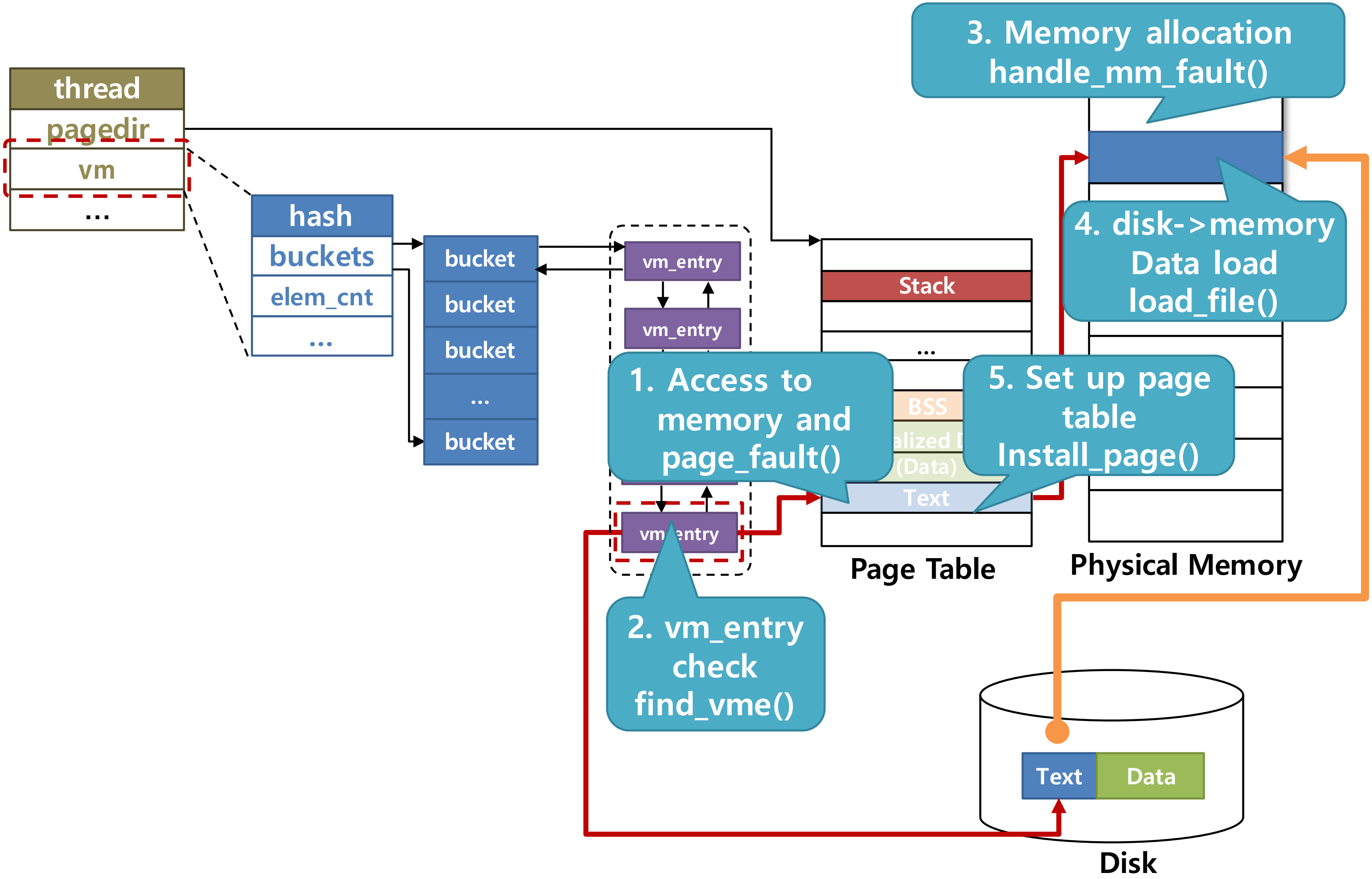


# Modify setup_stack()

pintos/src/vm/page.c

```
static bool setup_stack (void **esp)
{
        ...
        if (kpage != NULL)
        {
                ...
        }
        /* Create vm_entry */
        /* Set up vm_entry members */
        /* Using insert_vme(), add vm_enty to hash table */

        ...
}
```

# Design: Demand Paging


thread
pagedir
vm
...
hash
buckets
elem_cnt
...
bucket
bucket
bucket
...
bucket
vm_entry
vm_entry
vm_entry
Stack
...
BSS
alized D
(Data)
Text
Page Table
Physical Memory
Text
Data
Disk
1. Access to memory and page_fault()
2. vm_entry check find_vme()
3. Memory allocation handle_mm_fault()
4. disk->memory Data load load_file()
5. Set up page table Install_page()

# To do 1: page fault handling

- `page_fault()` exists in Pintos to manage the page fault.
  - `pintos/src/userprog/exception.c`
    - `static void page_fault (struct intr_frame *f)`
    - When existing Pintos manage page fault, after checking permission and validation of address, if error occurs, generate "segmentation fault" and kill(-1) to terminate.
    - Delete code related to `kill(-1)`.
    - Check Validation of fault_addr.
    - Define the new page fault handler and call it.
      - `handle_mm_fault(struct vm_entry *vme)`

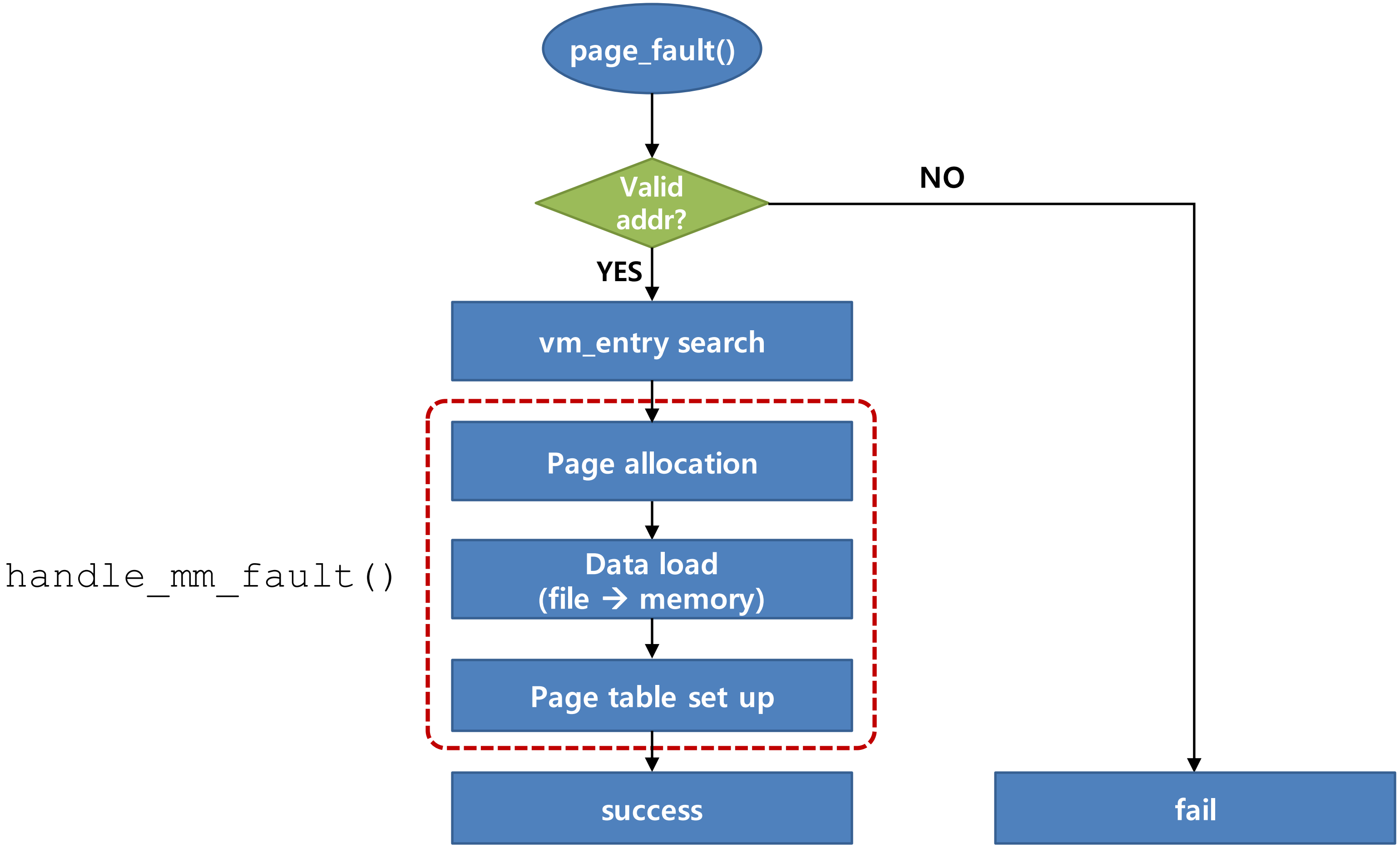
# page fault management

pintos/src/userporg/exception.c

```
  static void page_fault (struct intr_frame *f){
      ...
      /* Determine cause. */
      not_present = (f->error_code & PF_P) == 0;
      write = (f->error_code & PF_W) != 0;
      user = (f->error_code & PF_U) != 0;

      exit(-1);
      /* To implement virtual memory, delete the rest of the function
       body, and replace it with code that brings in the page to
       which fault_addr refers. */
      printf ("Page fault at %p: %s error %s page in %s context.\n",
            fault_addr,
            not_present ? "not present" : "rights violation",
            write ? "writing" : "reading",
            user ? "user" : "kernel");
      kill (f);
  }
```

↳ Delete & implement code

# page fault management



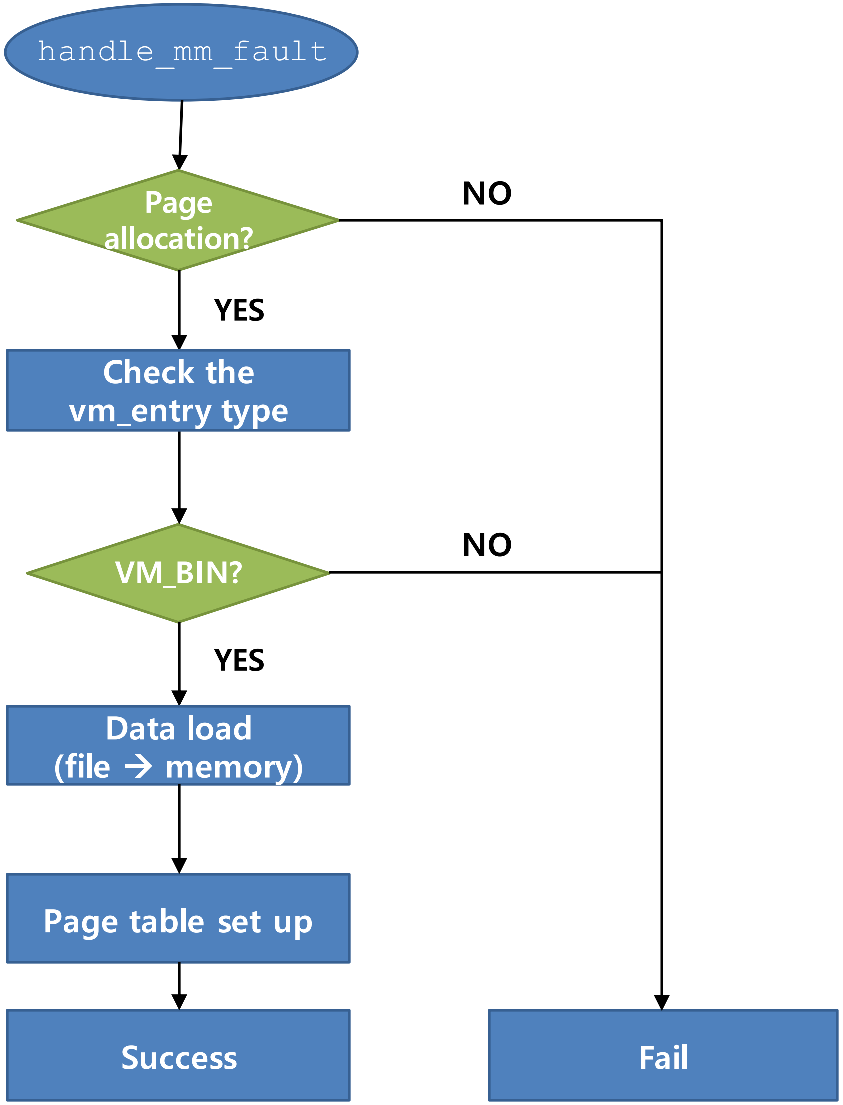
# To do 2: implement page fault handler

- Page fault handler function(pintos/src/userprog/process.c)
  - bool handle_mm_fault(struct vm_entry *vme)
    - handle_mm_fault is called to handle page fault.
    - When page fault occurs, allocate physical memory.
    - Load file in the disk to physical memory.
      - Use load_file (void* kaddr, struct vm_entry *vme).
    - Update the associated page table entry after loading into physical memory.
      - Use static bool install_page(void *upage, void *kpage,bool writable).

```
bool handle_mm_fault (struct vm_entry *vme)
{


}
```

# page fault handler for loading the ELF file

Later, we will cover anonymous page and the other file backed page.
Here, we only consider the ELF file.

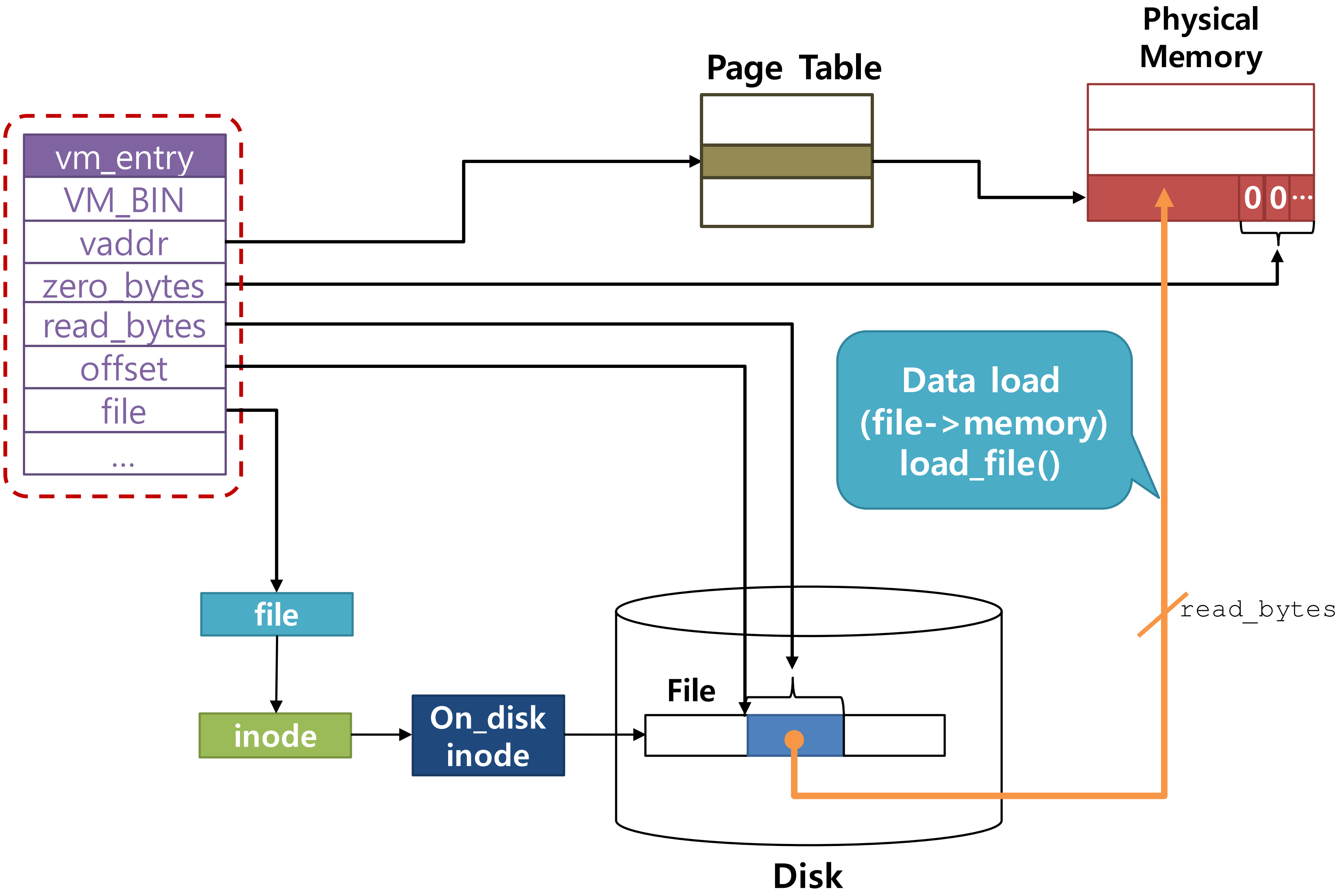


# To do 3: load the file to physical memory

- After physical memory allocation, load the file page from the disk to physical memory(Pintos/src/vm/page.c)
  - bool load_file (void* kaddr, struct vm_entry *vme)
    - Function to load a page from the disk to physical memory
    - Implement a function to load a page to kaddr by <file, offset> of vme.
    - Use file_read_at() or file_read() + file_seek().
    - If fail to write all 4KB, fill the rest with zeros.

```
bool load_file (void *kaddr, struct vm_entry *vme)
{
/* Using file_read_at()*/
/* Write physical memory as much as read_bytes by file_read_at*/
/* Return file_read_at status*/
/* Pad 0 as much as zero_bytes*/
/* if file is loaded to memory, return true */

}
```

# To do 3: load a file page to physical memory



# Functions for demand paging

- pintos/src/userprog/exception.c

```
static void page_fault (struct intr_frame *f)
        /* When page fault occurs, existing code kill(-1) to terminate*/
        /* Delete code related to kill(-1) */
        /*  Modify code to search for vm_entry and allocate page using handle_mm_fault() */
```

- pintos/src/vm/page.c

```
bool load_file (void* kaddr, struct vm_entry *vme)
        /* Load page in disk to physical memory */
        /* Implement function to load a page to kaddr by <file, offset> of vme */
        /* Use file_read_at() or file_read() + file_seek() */
```

- pintos/src/userprog/process.c

```
bool handle_mm_fault(struct vm_entry *vme)
       /* handle_mm_fault is function to handle page fault */
       /* If page fault occurs, allocate physical page */
```

# Files to modify

- Modify Makefile.build
  - Add code to use added page file

pintos/Makefile.build

```
...
userprog_SRC += userprog/tss.c       # TSS management.

# No virtual memory code yet.
#vm_SRC = vm/file.c                  # Some file.
vm_SRC = vm/page.c

# Filesystem code.
filesys_SRC  = filesys/filesys.c     # Filesystem core.
filesys_SRC += filesys/free-map.c    # Free sector
bitmap.
filesys_SRC += filesys/file.c        # Files.
filesys_SRC += filesys/directory.c   # Directories.
filesys_SRC += filesys/inode.c       # File headers.
filesys_SRC += filesys/fsutil.c      # Utilities.
...
```

# Files to modify (Cont.)

- Modify Makefile.tests
- If not, occurs fail when `make check`
  - Test run times may be exceeded depending on the environment.

pintos/tests/Make.tests

```
...
ifdef PROGS
include ../../Makefile.userprog
endif

TIMEOUT = 60  /* Change the test run time for Pintos from 60
seconds to 120 seconds */

clean::
        rm -f $(OUTPUTS) $(ERRORS) $(RESULTS)

grade:: results
       $(SRCDIR)/tests/make-grade $(SRCDIR) $< $(GRADING_FILE)
| tee $@
...
```

# Additional Functions you may want to implement

```
void vm_init(struct hash* vm)
    /* hash table initialization */
void vm_destroy(struct hash *vm)
    /* hash table delete */
struct vm_entry* find_vme(void *vaddr)
        /* Search vm_entry corresponding to vaddr in the address space of the
   current process */
bool insert_vme(struct hash *vm, struct vm_entry *vme)
        /* Insert vm_entry to hash table*/
bool delete_vme(struct hash *vm, struct vm_entry *vme)
        /* Delete vm_entry from hash table */
```

# Functions to add/modify

```
  static unsigned vm_hash_func(const struct hash_elem *e, void *aux
UNUSED)
          /* Calculate where to put the vm_entry into the hash table */


  static bool vm_less_func(const struct hash_elem *a, const struct
hash_elem *b, void *aux UNUSED)
          /* Compare address values of two entered hash_elem */


  static void vm_destroy_func(struct hash_elem *e, void *aux UNUSED)
          /* Remove memory of vm_entry */
```

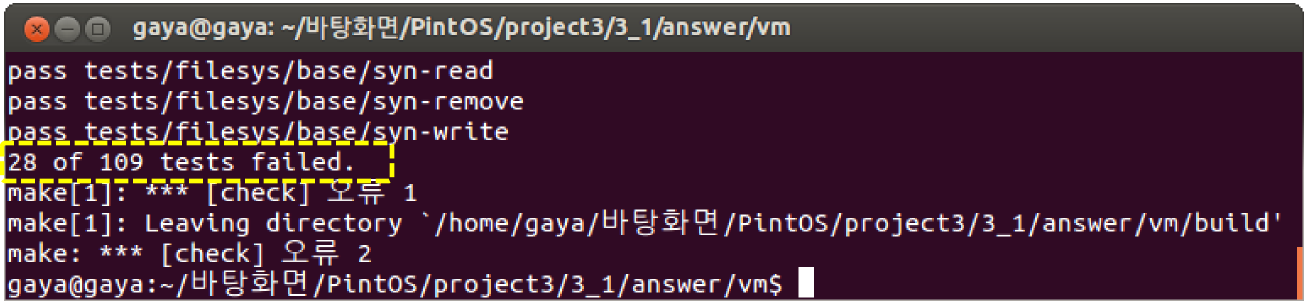
# Verify virtual memory project

- Confirm code behavior after completing virtual memory task
  - path : pintos/src/vm

```
$ make check
```

- 28 of 109 tests found to fail as a result of execution

  - pt-grow-stack
  - page-linear
  - page-merge-stk
  - mmap-unmap
  - mmap-exit
  - mmap-inherit
  - mmap-over-data
  - pt-grow-pusha
  - page-parallel
  - page-merge-mm
  - mmap-overlap
  - mmap-shuffle
  - mmap-misalign
  - mmap-over-stk
  - pt-big-stk-obj
  - page-merge-seq
  - mmap-read
  - mmap-twice
  - mmap-bad-fd
  - mmap-null
  - mmap-remove
  - pt-grow-stk-sc
  - page-merge-par
  - mmap-close
  - mmap-write
  - mmap-clean
  - mmap-over-code
  - mmap-zero



gaya@gaya: ~/바탕화면/PintOS/project3/3_1/answer/vm

```
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
28 of 109 tests failed.
make[1]: *** [check] 오류 1
make[1]: Leaving directory `/home/gaya/바탕화면/PintOS/project3/3_1/answer/vm/build'
make: *** [check] 오류 2
gaya@gaya:~/바탕화면/PintOS/project3/3_1/answer/vm$
```

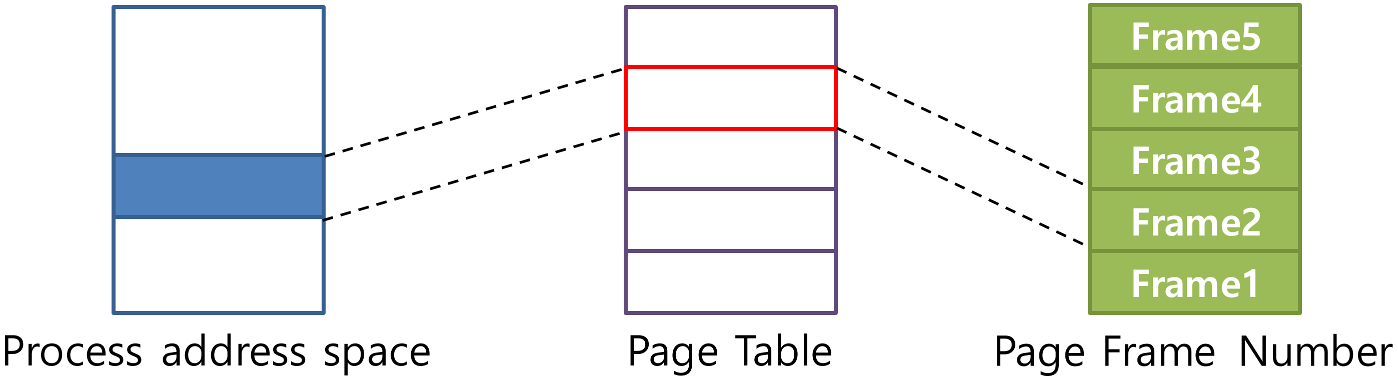
# Appendix

# Page address mapping function

```
#include "usrprog/process.c"

  static bool install_page(void *upage, void *kpage,
                           bool writable)
```

- Map physical page kpage and virtual page upage
- writable: writable(1), read-only(0)

Set up page table



# Physical page allocation and releasing interface

```
#include <threads/palloc.h>
```

```
void *palloc_get_page(enum palloc_flags flags)
```

- Allocate a 4KB page.
- Return physical address of page.
- flags
  - PAL_USER: allocate pages from user memory pool.
  - PAL_KERNEL: allocate pages in kernel memory pool.
  - PAL_ZERO: initialize pages to ‘0’.

```
void palloc_free_page(void *page)
```

- Use physical address of page as argument.
- Put page back in free memory pool.

# Pintos dynamic memory allocation and releasing interface

```
#include <threads/malloc.h>
```

```
void *malloc(size_t size)
```

- Allocate the memory chunk of ‘size’ and return start address.
- Use to allocate memory for dynamic objects such as `vm_entry`.

```
void free(void* p)
```

- Release the memory space allocated by malloc().
- Use address allocated memory through malloc() as argument.