Operating Systems Lab

Part 3: Virtual Memory

KAIST EE

Youjip Won

Overview of Virtual Memory

- Background of Virtual Memory in Pintos
- Requirements
  - Paging(swapping)
  - Growing stack
  - Memory mapped file
  - Accessing user memory

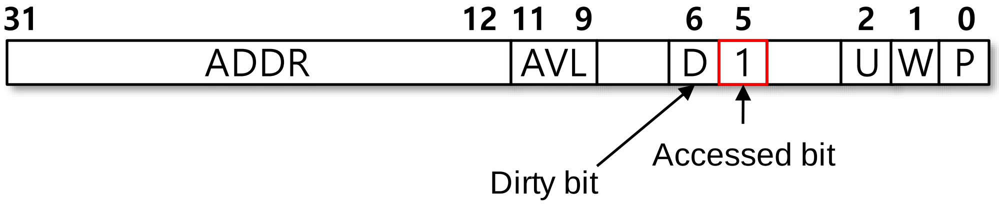
Swapping

To Do’ s

- Implement data structure to represent physical page frame.
- Implement page replacement policy such as LRU, clock, second-chance
- swapping
  - Store victim pages in swap space when they belong to data segment or stack segment.
  - swap-out pages are reloaded into memory by demand paging.

Hardware Support

- The dirty bit of page table is set to "1" by hardware when writing to the memory space
- The accessed bit in page table is set to '1' by hardware each time the page is referenced



- When page with dirty bit "1" is selected as victim, the changes must always be stored on disk
- Hardware does not re-zero the accessed bit.

Page Table Manipulation in Pintos (`userprog/pagedir.c`)

- `bool pagedir_is_dirty (uint32_t *pd, const void *vpage)`
  - Return dirty bit of pte for `vpage` in `pd`
- `void pagedir_set_dirty (uint32_t *pd, const void *vpage, bool dirty)`
  - Set the dirty bit to `dirty` in the pte for `vpage` in `pd`
- `bool pagedir_is_accessed (uint32_t *pd, const void *vpage)`
  - Return access bit of pte for `vpage` in `pd`
- `void pagedir_set_accessed (uint32_t *pd, const void *vpage, bool accessed)`
  - Set the access bit to `accessed` in the pte for `vpage` in `pd`

`struct page`: New data structure required

Select the physical page frame for replacement.

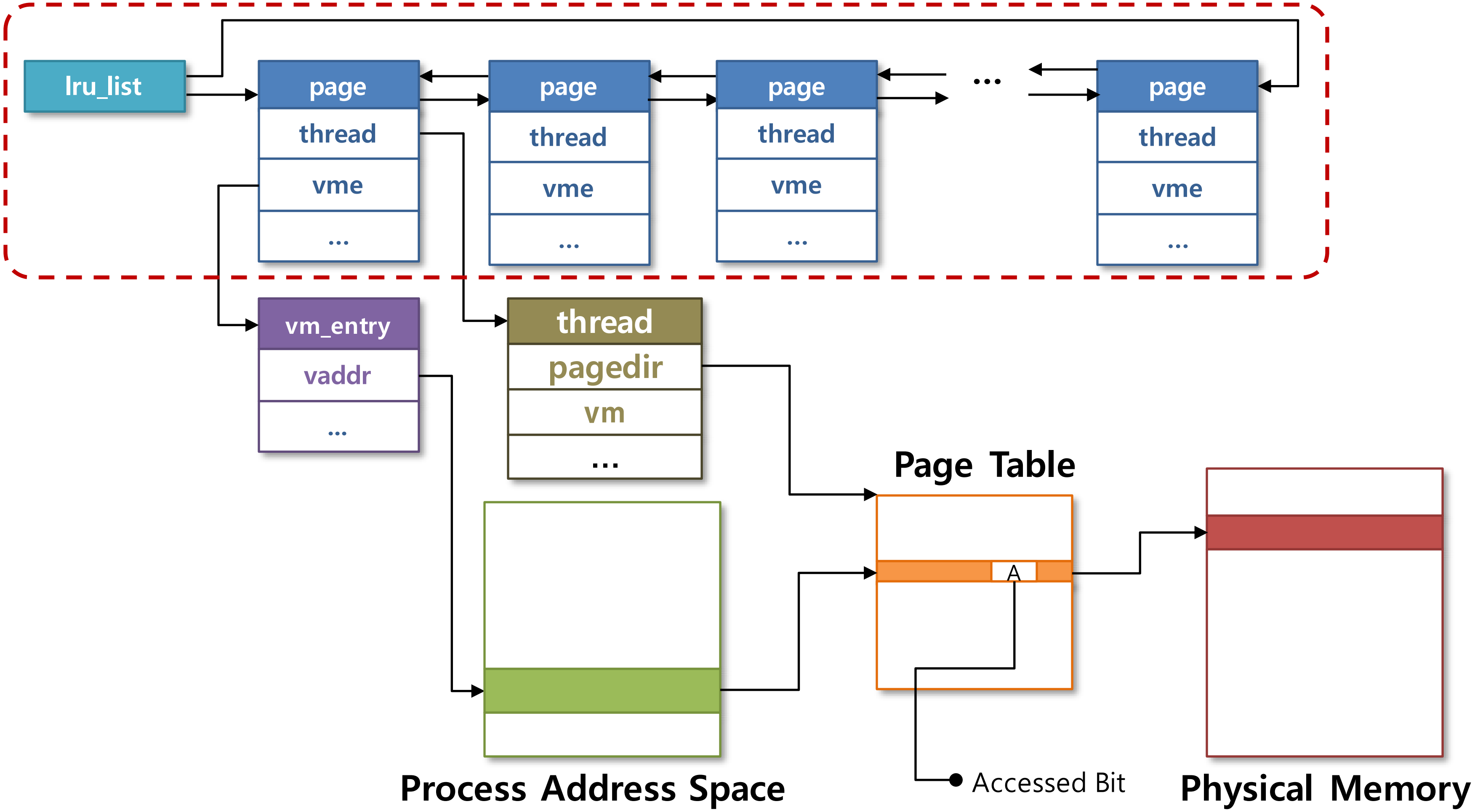
- Data structure representing each physical page that contains a user page
  - physical address of page
  - reference to the virtual page object to which physical page is mapped
  - Reference to the thread structure to which it belongs
  - `lru`: field for list

pintos/src/vm/page.h

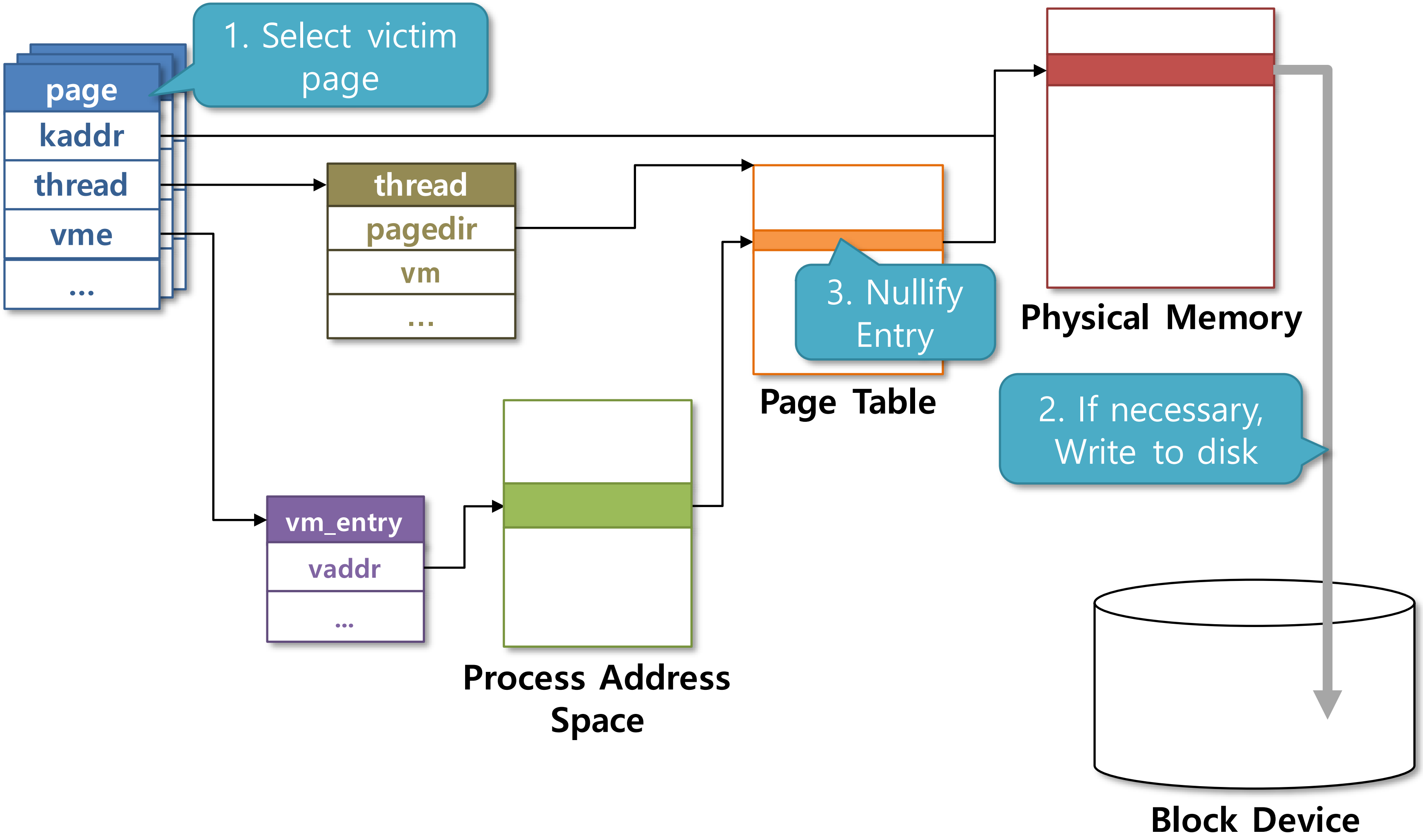
```
struct page {

    // fill this out

};
```

A page pool for swapping

- Manage physical pages in use as a list of pages.
- `lru_list` : global variable

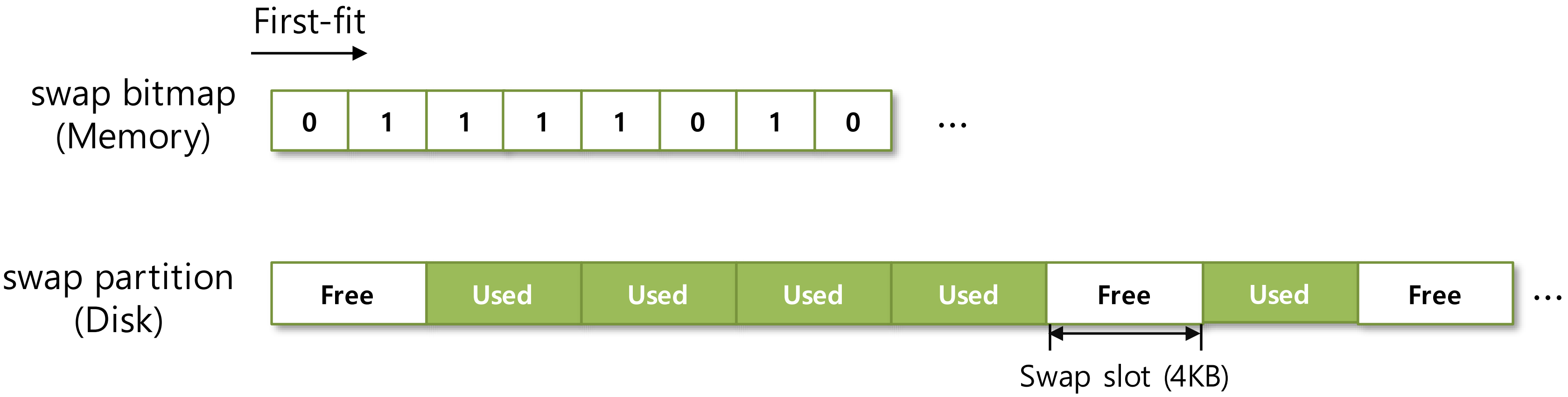


Swap-out


page
kaddr
thread
vme
...
1. Select victim page
thread
pagedir
vm
...
vm_entry
vaddr
...
Process Address Space
3. Nullify Entry
Page Table
Physical Memory
2. If necessary, Write to disk
Block Device

Managing swap partition

- Swap partition is managed per swap slot unit(4 Kbyte).
- Maintaining a swap partition: swap bitmap (global variable in memory)
- Search bitmap for free slot.
- What happens to swap bitmap if the system crashes?



Functions offered by pintos for swap space manipulation

- Swap partition is provided as block device in pintos.
- Functions for block device (`src/block/block.c`)
  - `struct block *block_get_role (enum block_type role)`
    - Return the block device (`struct block *`) fulfilling the given ROLE.
    - ROLEs defined in pintos now (`devices/block.h`)
      - `BLOCK_KERNEL`: OS Partition
      - `BLOCK_FILESYS`: File system
      - `BLOCK_SCRATCH`: Scratch partition
      - `BLOCK_SWAP`: Swap partition
  - `void block_read (struct block *block, block_sector_t sector, void *buffer)`
    - Read contents at `sector` on `block` and save them at `buffer`
  - `void block_write (struct block *block, block_sector_t sector, const void *buffer)`
    - Write contents at `buffer` at `sector` on `block`

Implementation

- LRU list for physical page frame
  - List of `struct page`
  - List of physical pages allocated to user process
- functions for allocate/release physical page frame from the list
  - When there runs out of physical page frame, select a victim and swap it out.
- Modify page fault handler for swapping.
  - Before: Physical page is allocated directly when page fault occurs.
    When there is no page to allocate, pintos is finished.
  - After: Physical page is allocated from LRU list when page fault occurs.
    When there is no page to allocate, pintos swap in the page.

Functions to write

- Function about LRU list (initializing, insert, remove).
- Function to allocate a page from LRU list.
- Function to free page from LRU list.
- Function to select victim page and swap-out the page.
  - e.g.: Clock algorithm, Second chance algorithm
- Function about swapping (initializing, swap in, swap out).

Functions to modify

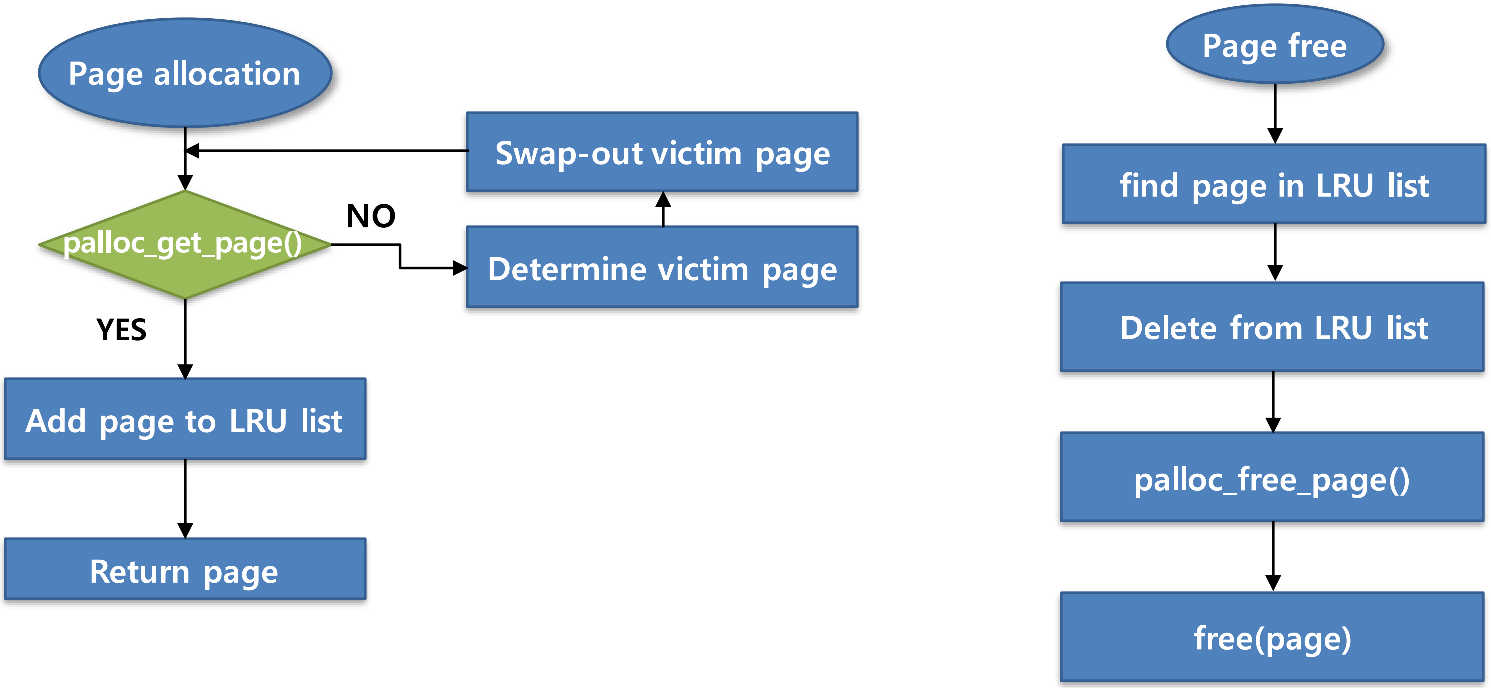
- `bool handle_mm_fault(struct vm_entry *vme)`
  - Modify to allocate physical pages from LRU list when page fault occurs
  - Modify to swap-in if `vm_entry` type is `VM_ANON`

- `static bool setup_stack(void **esp)`
  - Modify to allocate pages from LRU list when page fault occurs

- `int main(void)`
  - Initialize LRU list.

Functions for allocation/free page

- Try to obtain free space when memory cannot be allocated through `palloc_get_page()` within the page allocation function.

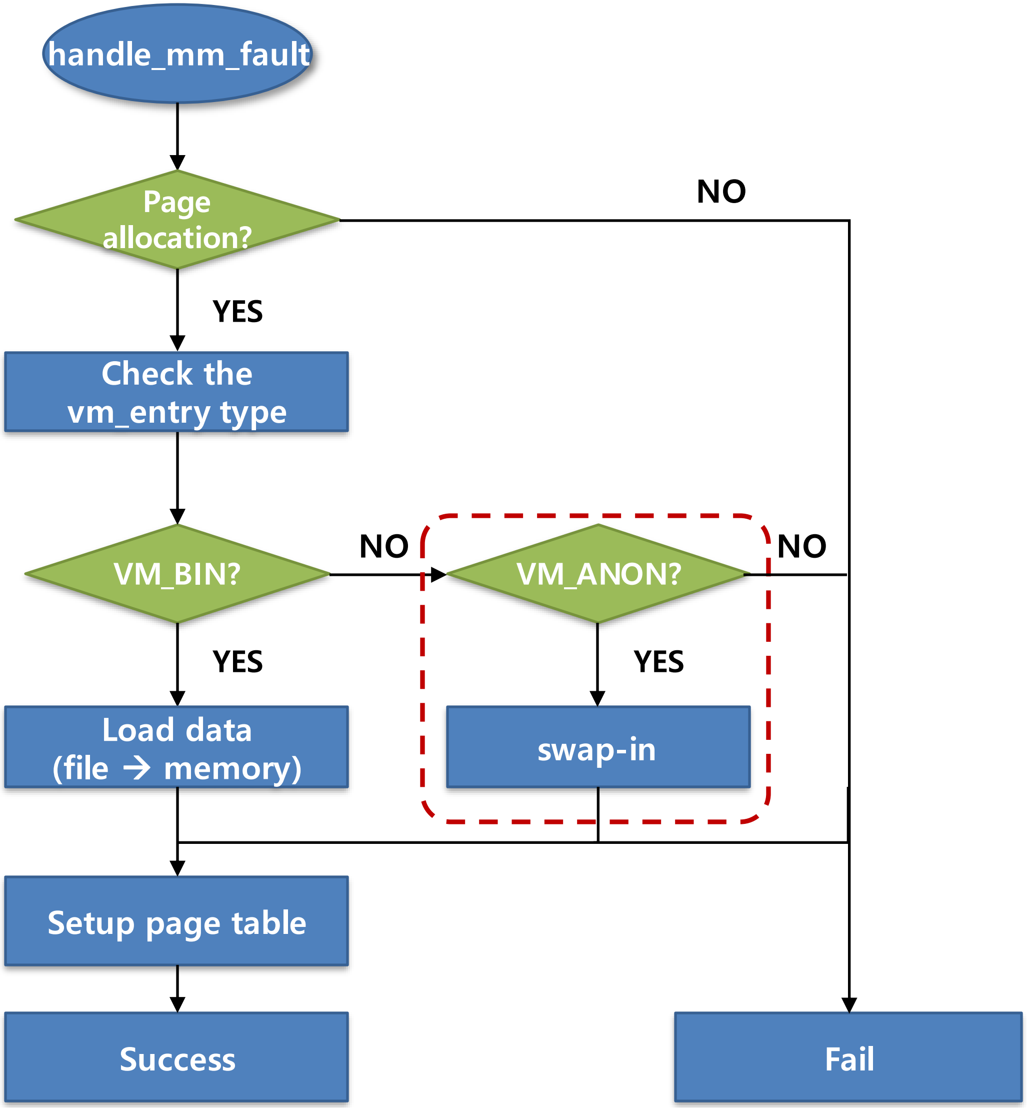


Swap-out

- Type of a page in the physical page frame
  - VM_BIN
    - If dirty bit is “1”, write to the swap partition and free the page frame.
    - Change type to VM_ANON for demand paging
  - VM_FILE
    - If dirty bit is “1”, write the page to the file and free the page frame.
    - If dirty bit is “0”, free the page frame.
  - VM_ANON
    - Write to the swap partition.
- Mark the page “not present” in pd (page directory).

```
void pagedir_clear_page (uint32_t *pd, void *upage)
```

Demand paging for anonymous page (stack or heap)



Modify `handle_mm_fault()`

- If `vm_entry` type is `VM_ANON`, modify code to swap in

pintos/src/userporg/process.c

```
bool handle_mm_fault(struct vm_entry *vme){
    bool success = false;
    viod *kaddr;
    ...
    switch(vme->type){

        case VM_BIN:
        success = load_file(kaddr, vme);
        break;

        case VM_ANON:
        /* insert swap in code */
        break;
    }
    ...
```

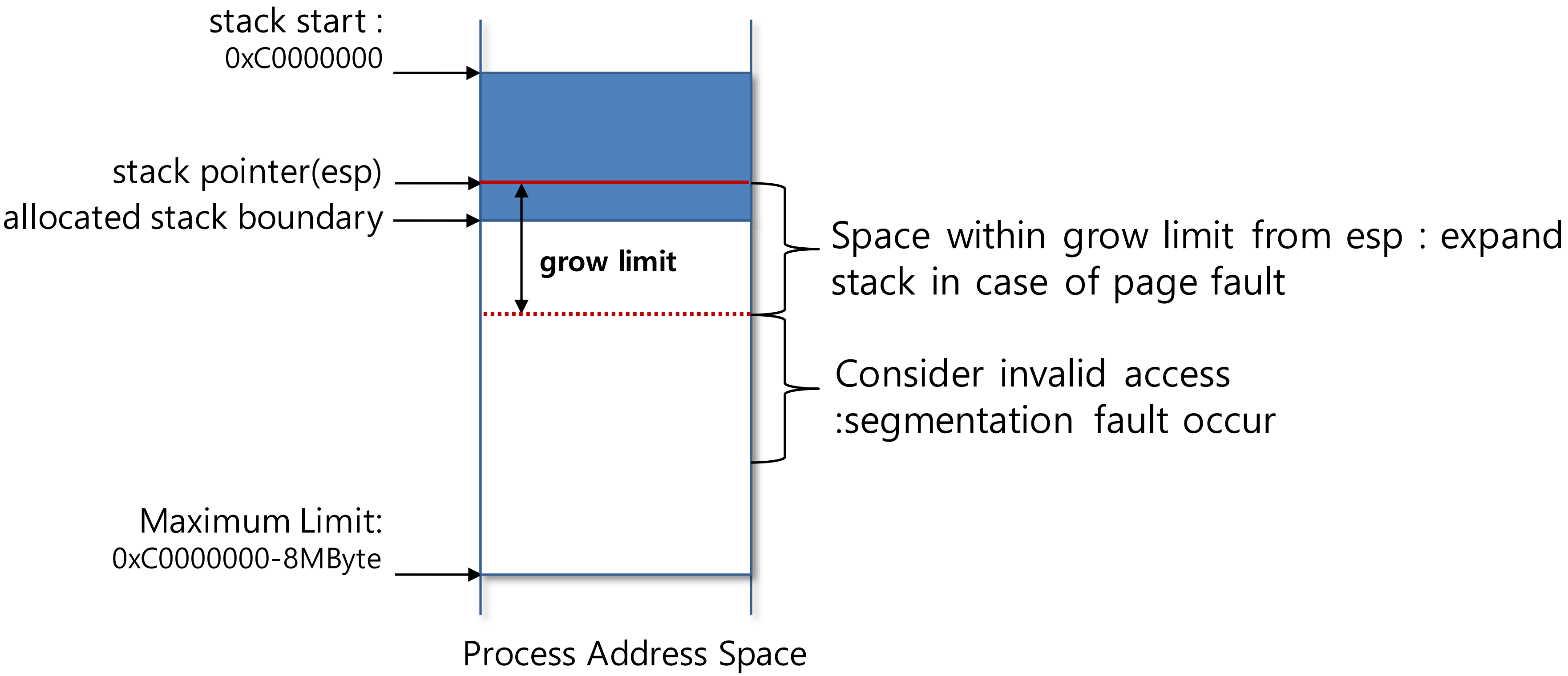
Growing Stack

Expandable Stack

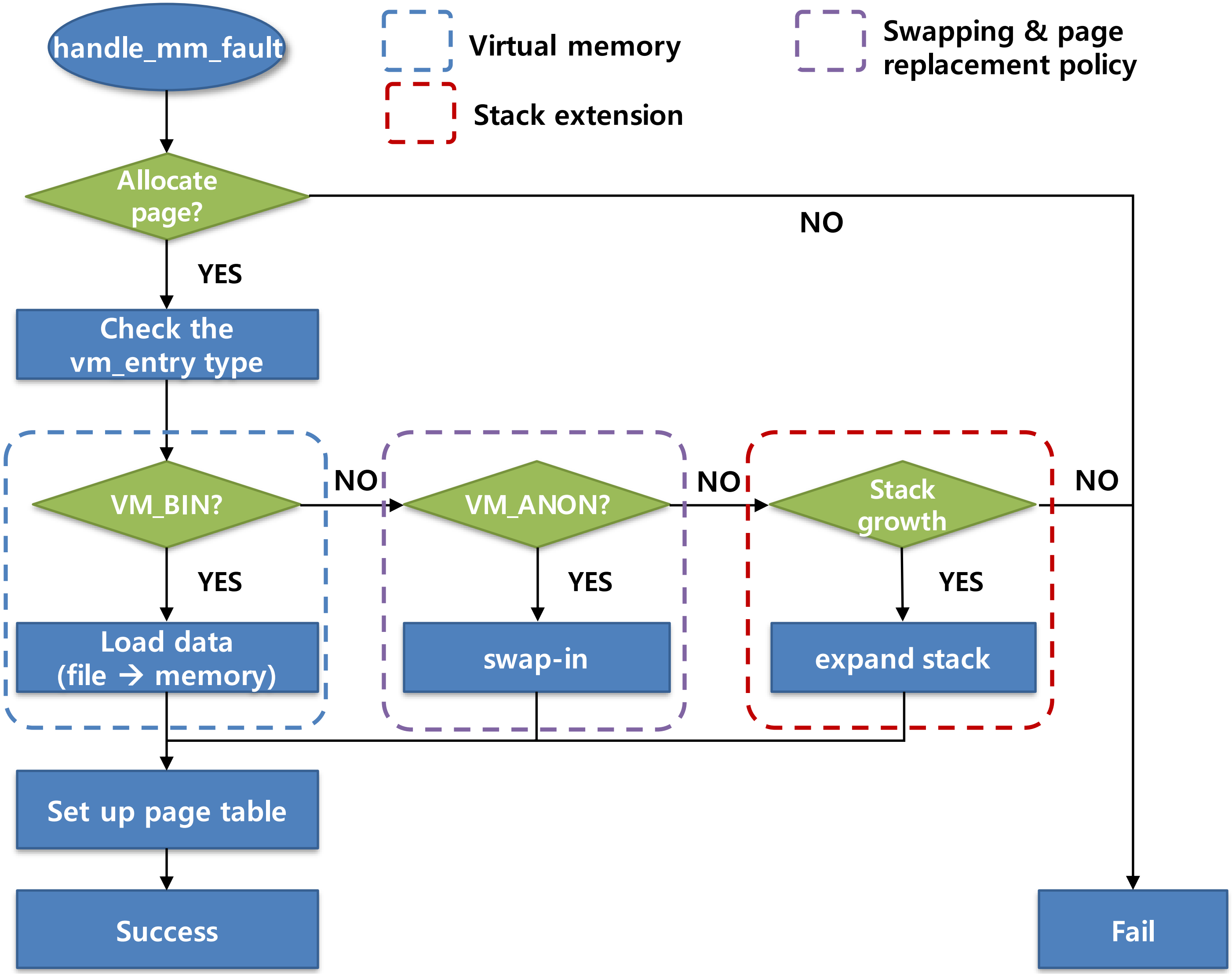
- Implement expandable stack
  - In current pintos, stack size is fixed to 4KB.
  - Make the stack expandable.
    - If a process accesses the address that lies outside the stack and that can be handled by expanding the stack, expand the stack.
      - e.g. (access address < stack pointer – 32) Expand stack
  - maximum size of stack is 8MB.

When to expand stack

- Expand the stack when the memory access is within 32 Byte of stack top.
  - “PUSHA” instruction in 80x86 pushes 32 bytes at once.



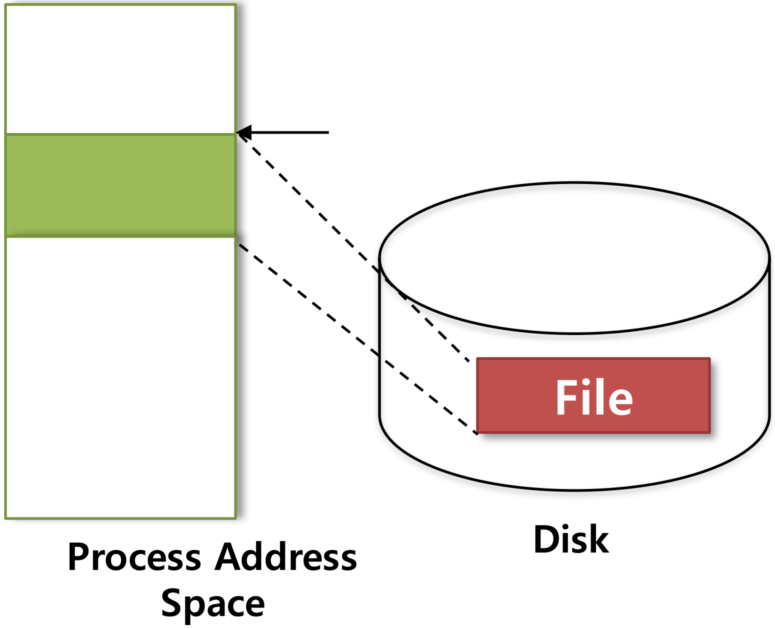
Stack extension mechanism



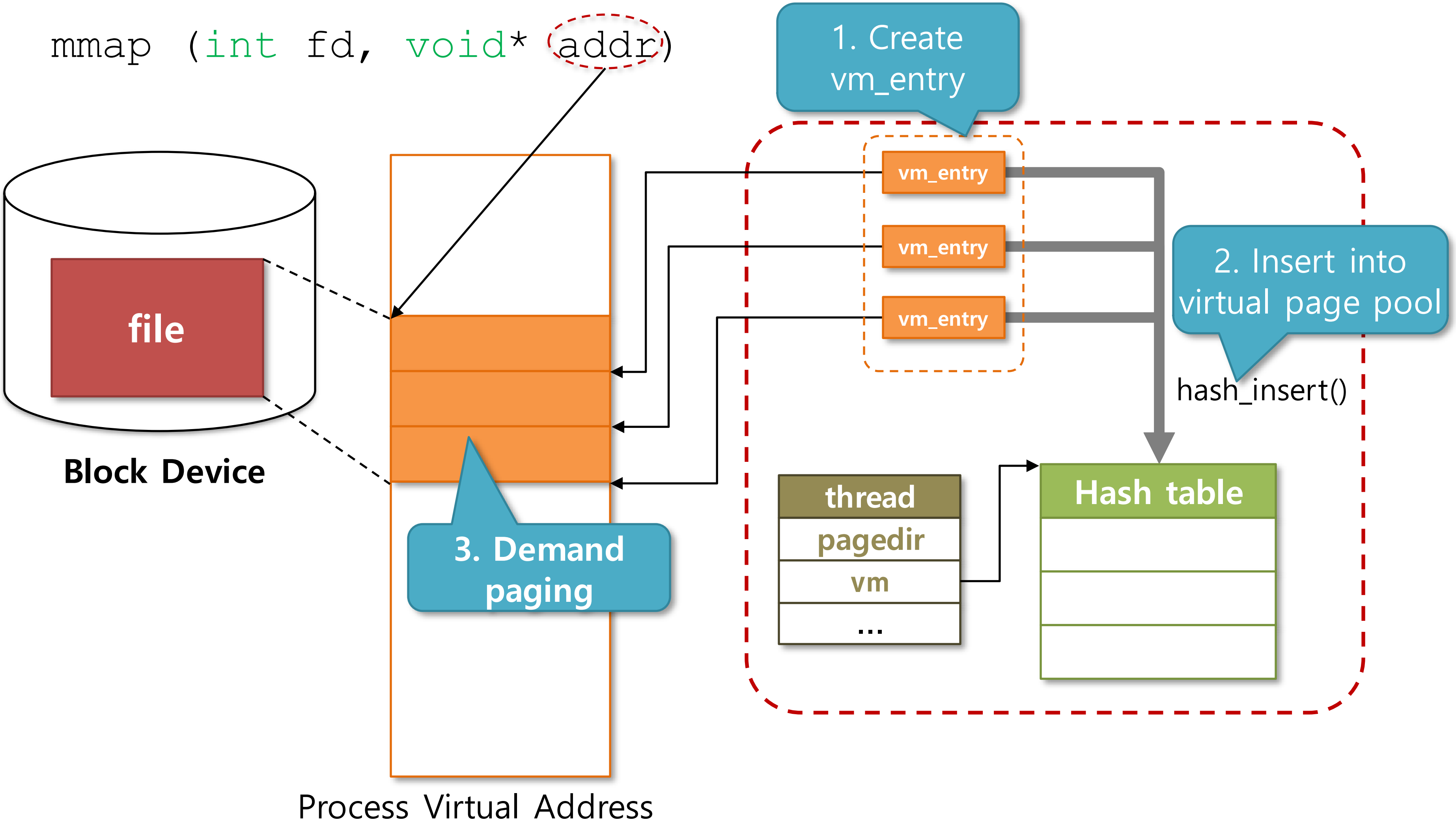
Memory Mapped File

mmap vs. munmap

```
main (int argc, char *argv[]) {
  int i;

  for (i = 1; i < argc; i++)     {
      int fd;
      mapid_t map;
      void *data = (void *) 0x10000000;
      int size;
      fd = open (argv[i]);
      size = filesize (fd);
      map = mmap (fd, data);
      write (STDOUT_FILENO, data, size);
      munmap (map);
    }
  return EXIT_SUCCESS;
}
```



mmap and munmap



mmap() and munmap()

- `int mmap(int fd, void *addr)`
  - Load file data into memory by demand paging.
  - `mmap()`'ed page is swapped out to its original location in the file.
  - For a fragmented page, fill the unused fraction of page with zero.
  - Return mapping_id: unique id within a process to identify the mapped file.
  - Fails if
    - File size is 0.
    - Addr is not page aligned.
    - Address is already in use.
    - Addr is 0.
    - STDIN and STDOUT are not mappable..
- `void munmap(mapid_t mapid)`
  - Unmap the mappings in the `mmap_list` which has not been previously unmapped.

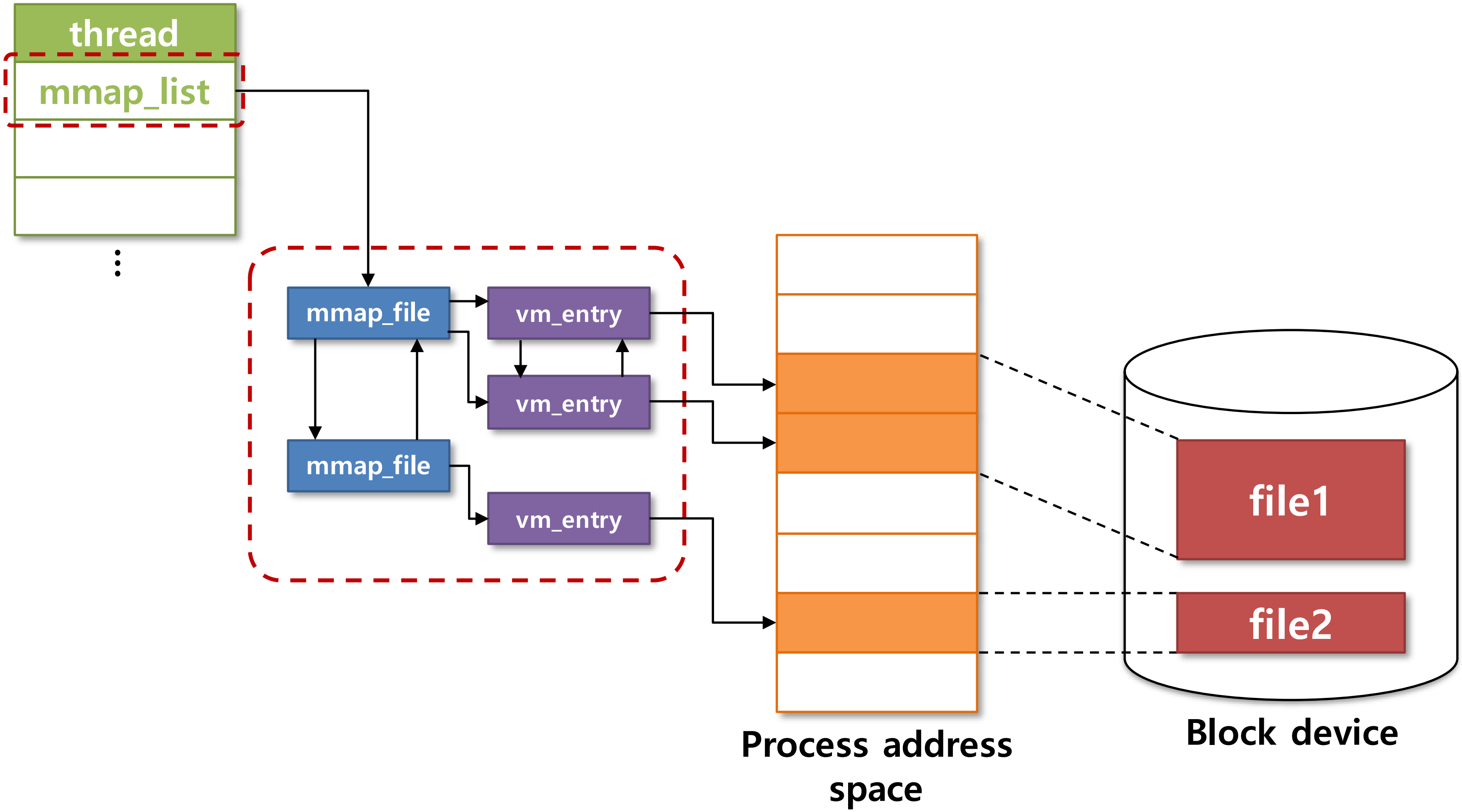
Requirements

- All mappings of a process are implicitly unmapped when the process exits.
- When a mapping is unmapped, the pages are written back to the file.
- Upon munmap, the pages are removed from the process' virtual page list.
- Once created, mapping is valid until it is unmapped regardless of the file is closed or deleted.
- If the two or more processes map the same file, they do not have to see the consistent view.

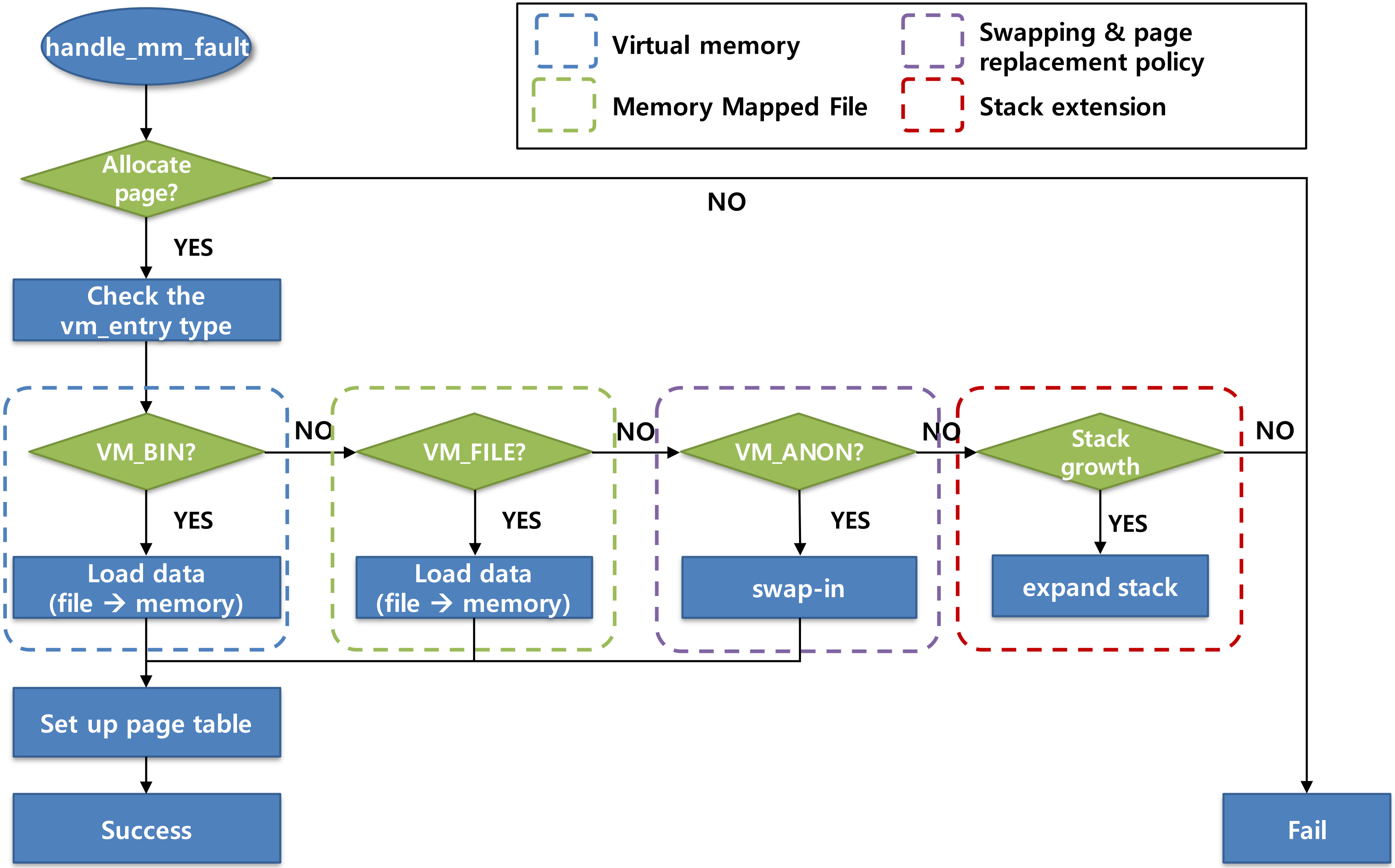
Additional data structure and Functions to modify

- `struct mmap_file`
  - Data structure containing information from mapped files
  - mapping id
  - mapping file object
  - `mmap_file` list element
  - `vm_entry` list.
- `bool handle_mm_fault(struct vm_entry *vme)`
  - Load data if `vm_entry` type is `VM_FILE`
- `void process_exit (void)`
  - Release all `vm_entry` corresponding to `mapping_list` at the end of process.

Managing mapped files



Modify page fault handler for mmap()



Accessing User Address Space

Why should page fault not occur in kernel code?

- A deadlock on kernel resource can occur.

User Context	Kernel Context	Interrupt Handler Context
Call system call: read()		
Context switch (system call)	Execute read() driver locks "struct channel" Read data from disk Write data to user buffer	Context switch (Page fault)
		Handle page fault: • Select a victim page. • Write a victim page to swap space. • **Acquire driver lock.** Deadlock because of double locking on driver lock

Pinning Page

- Prevent evicting the pages accessed during system call
- Define pinning flag about each physical page.
- On every system call,
  - Find the virtual page and pin the associated physical page.
  - After the system call returns and before the system call handler returns, unpin the pages
- On Swapping handler,
  - Do not select a pinned page as a victim.