

SWE3004 Operating Systems, fall 2024

Project 3. Virtual memory

TA)

Osama Khan

Shinhyun Park

Gwanjong Park

Chanu Yu

Project plan

- Total 6 projects

- ~~0) Booting xv6 operating system~~

- ~~1) System call~~

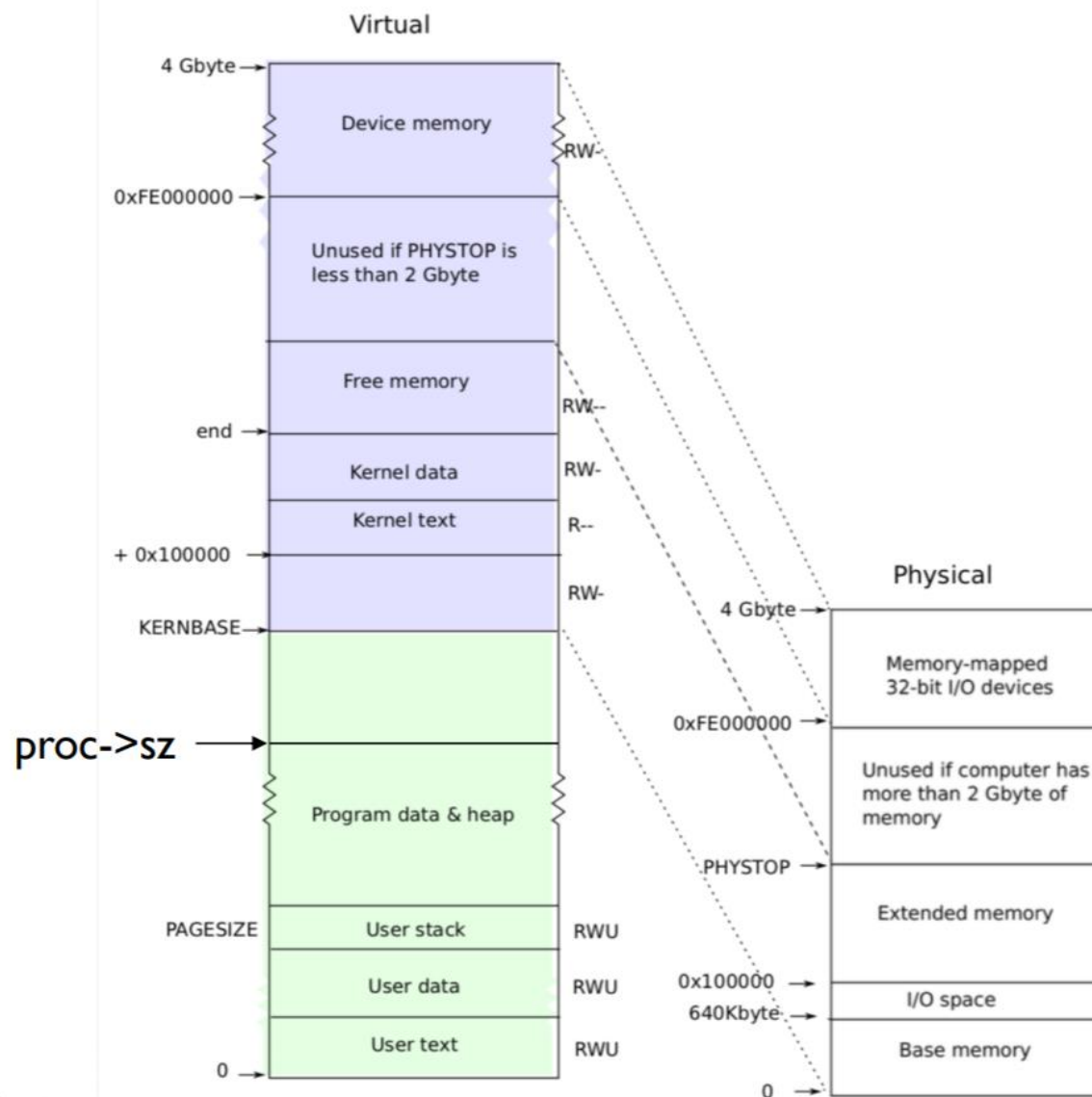
- ~~2) CPU scheduling~~

- 3) Virtual memory**

- 4) Page replacement

- 5) File systems

xv6 Memory Layout



How Physical Memories Initialized in xv6

- main() of main.c

```
int
main(void)
{
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // detect other processors
    lapicinit(); // interrupt controller
    seginit(); // segment descriptors
    picinit(); // disable pic
    ioapicinit(); // another interrupt controller
    consoleinit(); // console hardware
    uartinit(); // serial port
    pinit(); // process table
    tvinit(); // trap vectors
    binit(); // buffer cache
    fileinit(); // file table
    ideinit(); // disk
    startothers(); // start other processors
    kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
    userinit(); // first user process
    mpmain(); // finish this processor's setup
}
```

These two functions divide & manage physical memories with pages

How Physical Memories Initialized in xv6

```
void
kinit1(void *vstart, void *vend)
{
    initlock(&kmem.lock, "kmem");
    kmem.use_lock = 0;
    freerange(vstart, vend);
}

void
kinit2(void *vstart, void *vend)
{
    freerange(vstart, vend);
    kmem.use_lock = 1;
}

void
freerange(void *vstart, void *vend)
{
    char *p;
    p = (char*)PGROUNDUP((uint)vstart);
    for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
        kfree(p);
}
```

```
void
kfree(char *v)
{
    struct run *r;

    if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
        panic("kfree");

    // Fill with junk to catch dangling refs.
    memset(v, 1, PGSIZE);

    if(kmem.use_lock)
        acquire(&kmem.lock);
    r = (struct run*)v;
    r->next = kmem.freelist;
    kmem.freelist = r;
    if(kmem.use_lock)
        release(&kmem.lock);
}
```

- kinit1() sets up for lock-less allocation in the first 4MB
- kinit2() arranges for more memory (until PHYSTOP) to be allocatable (224MB)
- freerange() kfree() with page size unit
- kfree() fills page with 1s, and put it into freelist (page pool)

How Physical Memories Initialized in xv6

- `fork()` creates a child with exactly the same memory contents as the parent

```
int
fork(void)
{
    int i, pid;
    struct proc *np;
    struct proc *curproc = myproc();

    // Allocate process.
    if((np = allocproc()) == 0){
        return -1;
    }

    // Copy process state from proc.
    if((np->pgdir = copyvm(curproc->pgdir, curproc->sz)) == 0){
        kfree(np->kstack);
        np->kstack = 0;
        np->state = UNUSED;
        return -1;
    }
    np->sz = curproc->sz;
    np->parent = curproc;
    *np->tf = *curproc->tf;

    // Clear %eax so that fork returns 0 in the child.
    np->tf->eax = 0;

    for(i = 0; i < NOFILE; i++)
        if(curproc->ofile[i])
            np->ofile[i] = filedup(curproc->ofile[i]);
    np->cwd = idup(curproc->cwd);

    safestrcpy(np->name, curproc->name, sizeof(curproc->name));

    pid = np->pid;

    acquire(&ptable.lock);

    np->state = RUNNABLE;

    release(&ptable.lock);

    return pid;
}
```

- `allocproc()` allocates kernel stack
- `copyvm()` copies parent's page table

```
pde_t*
copyvm(pde_t *pgdir, uint sz)
{
    pde_t *d;
    pte_t *pte;
    uint pa, i, flags;
    char *mem;

    if((d = setupkvm()) == 0)
        return 0;
    for(i = 0; i < sz; i += PGSIZE){
        if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
            panic("copyvm: pte should exist");
        if(!(*pte & PTE_P))
            panic("copyvm: page not present");
        pa = PTE_ADDR(*pte);
        flags = PTE_FLAGS(*pte);
        if((mem = kalloc()) == 0)
            goto bad;
        memmove(mem, (char*)P2V(pa), PGSIZE);
        if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
            kfree(mem);
            goto bad;
        }
    }
    return d;
}

bad:
    freevm(d);
    return 0;
}
```

Project 3: Implement Simple mmap()

- **Implement three system calls and page fault handler on xv6**
- **What your code should handle**
 1. mmap() syscall
 2. Page fault handler
 3. munmap() syscall
 4. freemem() syscall

1. mmap() system call on xv6

- Simple mmap() synopsis

```
uint mmap(uint addr, int len, int prot, int flags, int fd, int
```

1. ***addr*** is always page-aligned
 - MMAPBASE + *addr* is the start address of mapping
 - MMAPBASE of each process's virtual address is 0x40000000
2. ***length*** is also a multiple of page size
 - MMAPBASE + *addr* + *length* is the end address of mapping
3. ***prot*** can be **PROT_READ** or **PROT_READ|PROT_WRITE**
 - *prot* should be match with file's open flag

1. mmap() system call on xv6

- Simple mmap() synopsis

```
uint mmap(uint addr, int len, int prot, int flags, int fd, int offset)
```

4. *flags* can be given with the combinations

- 1) If **MAP_ANONYMOUS** is given, it is anonymous mapping
- 2) If **MAP_ANONYMOUS** is not given, it is file mapping
- 3) If **MAP_POPULATE** is given, allocate physical page & make page table for whole mapping area.
- 4) If **MAP_POPULATE** is not given, just record its mapping area.

If page fault occurs to according area (access to mapping area's virtual address), allocate physical page & make page table to according page

- 5) Other flags will not be used

1. mmap() system call on xv6

- Simple mmap() synopsis

```
uint mmap(uint addr, int len, int prot, int flags, int fd, int offset)
```

5. *fd* is given for file mappings, if not, it should be -1
6. *offset* is given for file mappings, if not, it should be 0

Return

Succeed: return the start address of mapping area

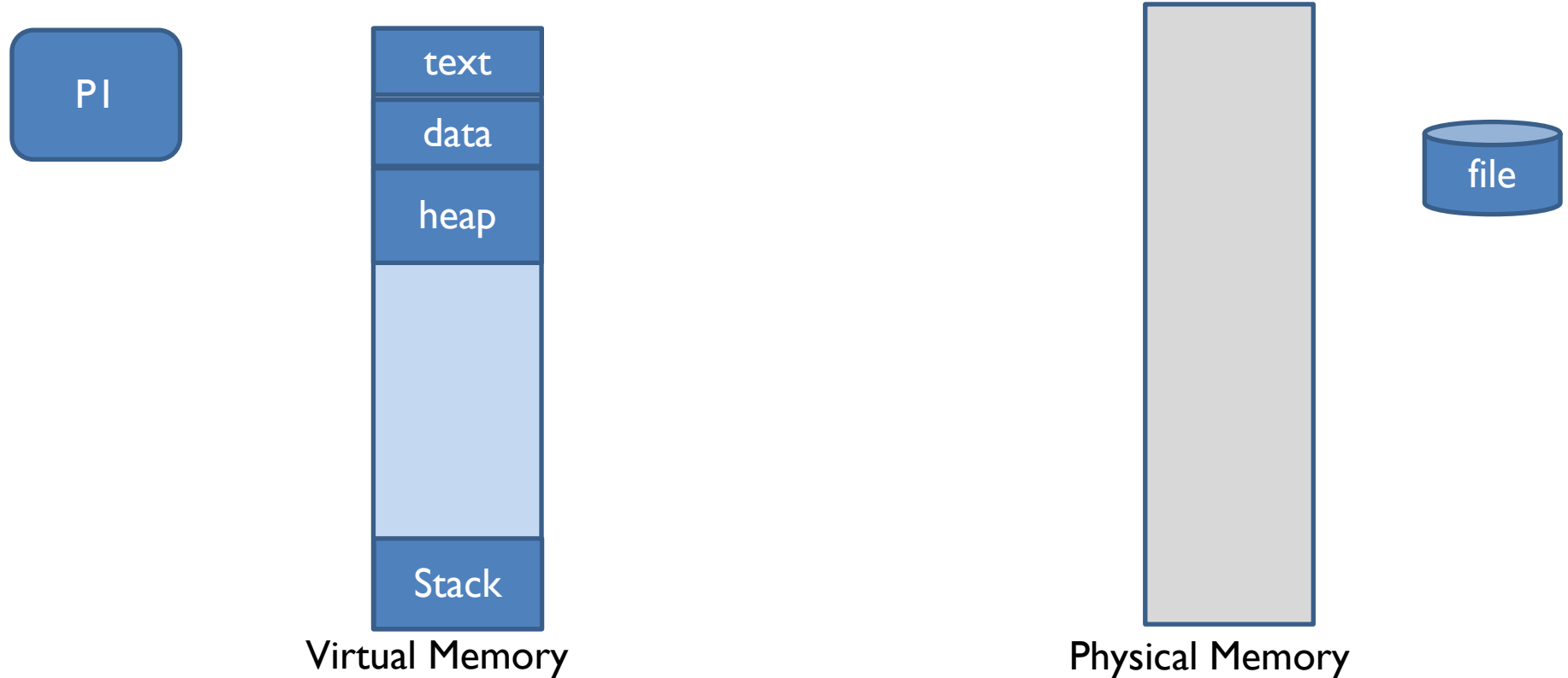
Failed: return 0

- It's not anonymous, but when the *fd* is -1
- The protection of the file and the *prot* of the parameter are different
- The situation in which the mapping area is overlapped is not considered
- If additional errors occur, we will let you know by writing notification

How file mmap() Works

I) Private file mapping with **MAP_POPULATE**

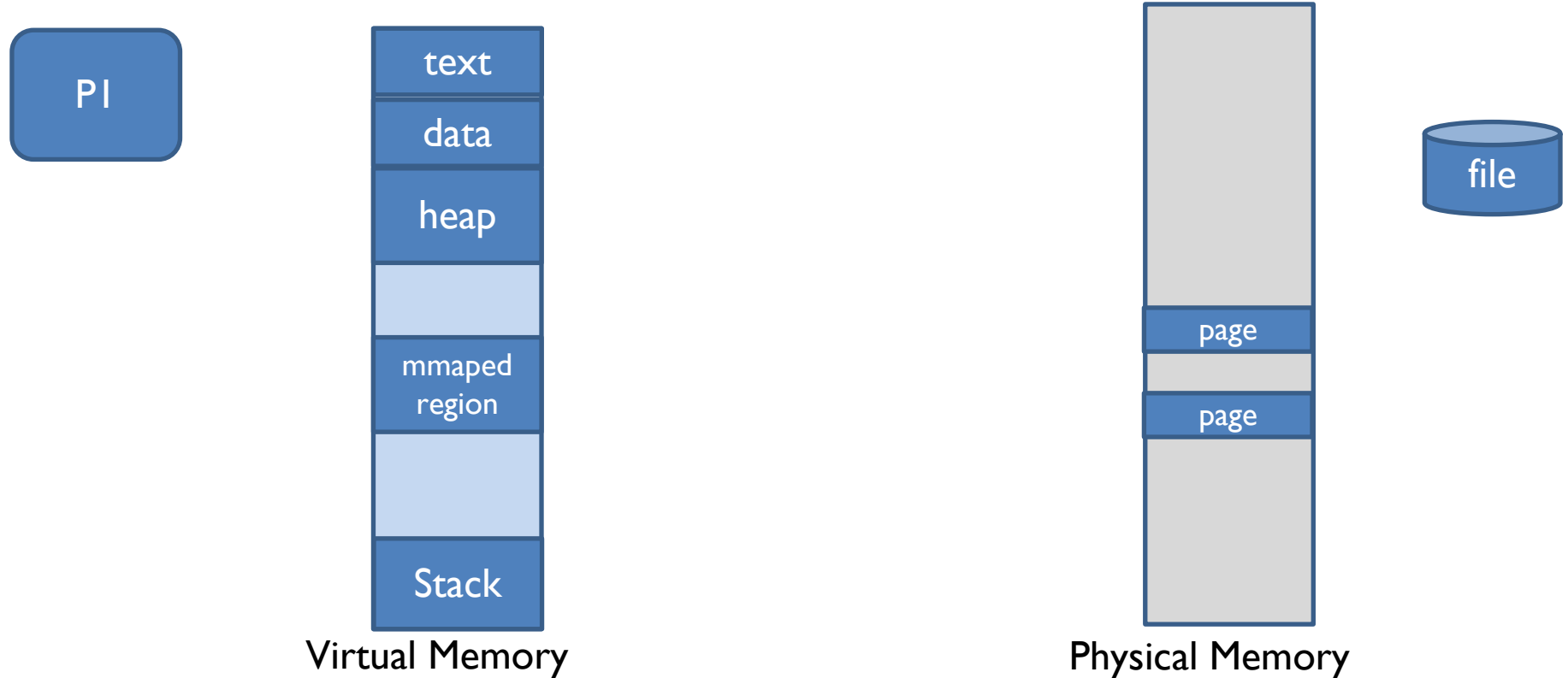
- `mmap(0, 8192, PROT_READ, MAP_POPULATE, fd, 4096)`
 - mmap 2pages



How file mmap() Works

I) Private file mapping with **MAP_POPULATE**

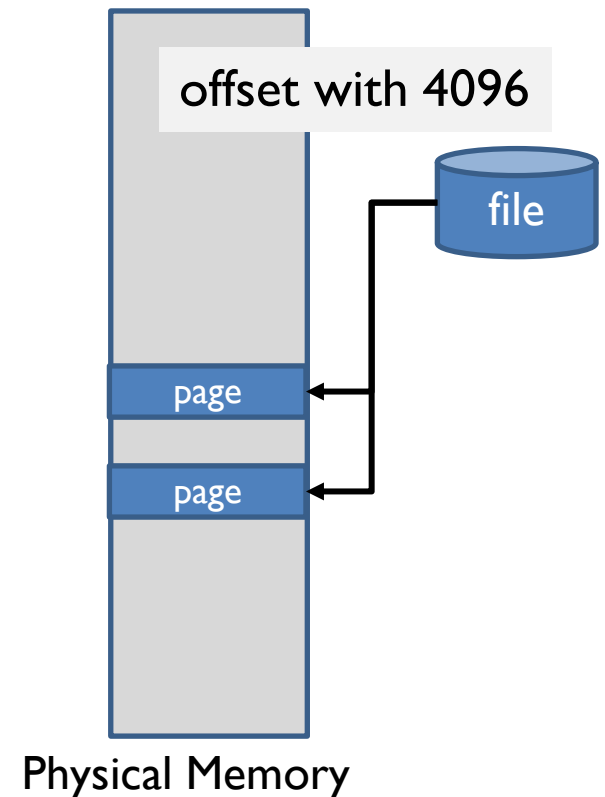
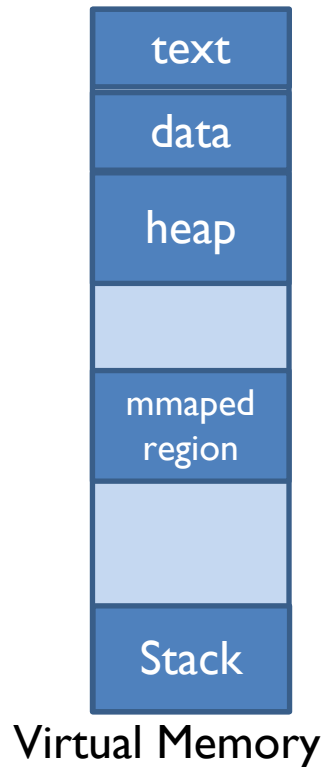
- `mmap(0, 8192, PROT_READ, MAP_POPULATE, fd, 4096)`
 - `mmap 2pages`



How file mmap() Works

I) Private file mapping with **MAP_POPULATE**

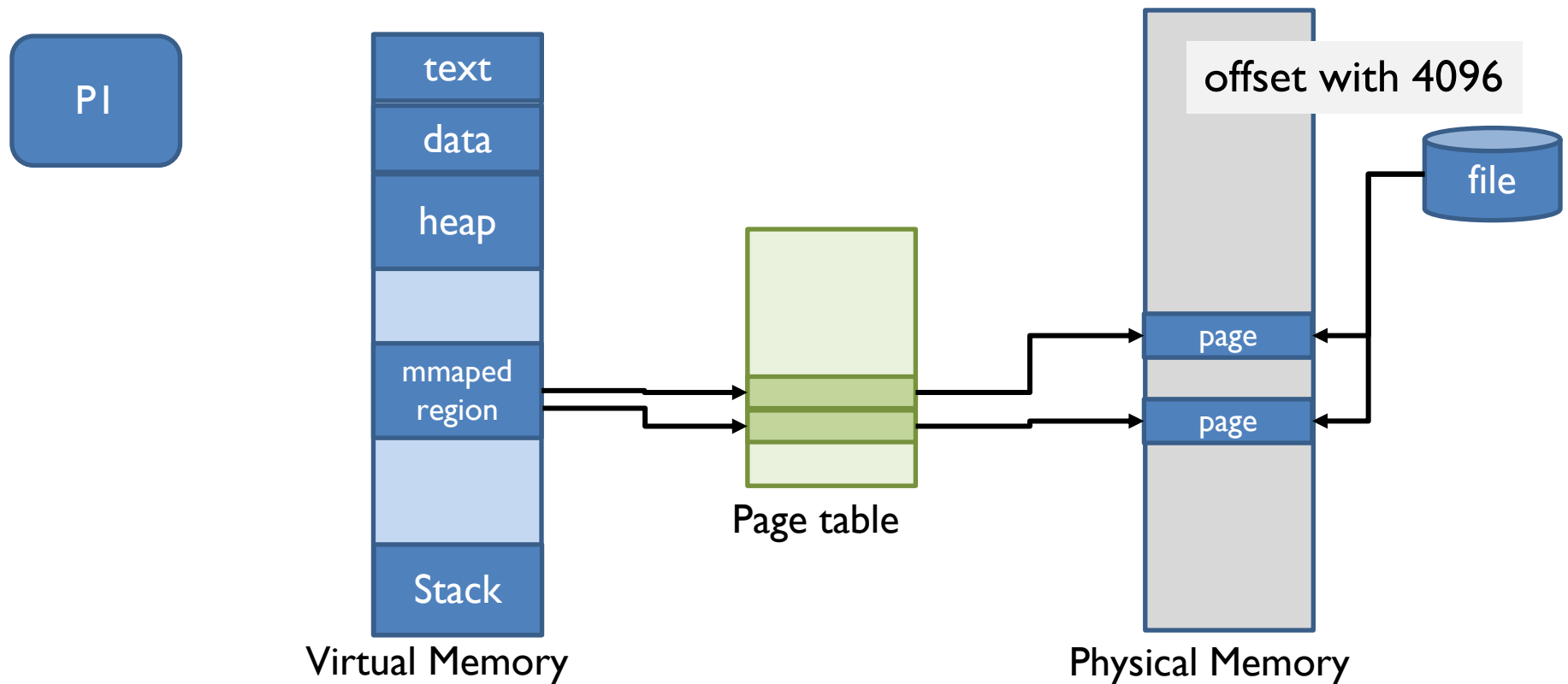
- `mmap(0, 8192, PROT_READ, MAP_POPULATE, fd, 4096)`
 - `mmap 2pages`



How file mmap() Works

I) Private file mapping with **MAP_POPULATE**

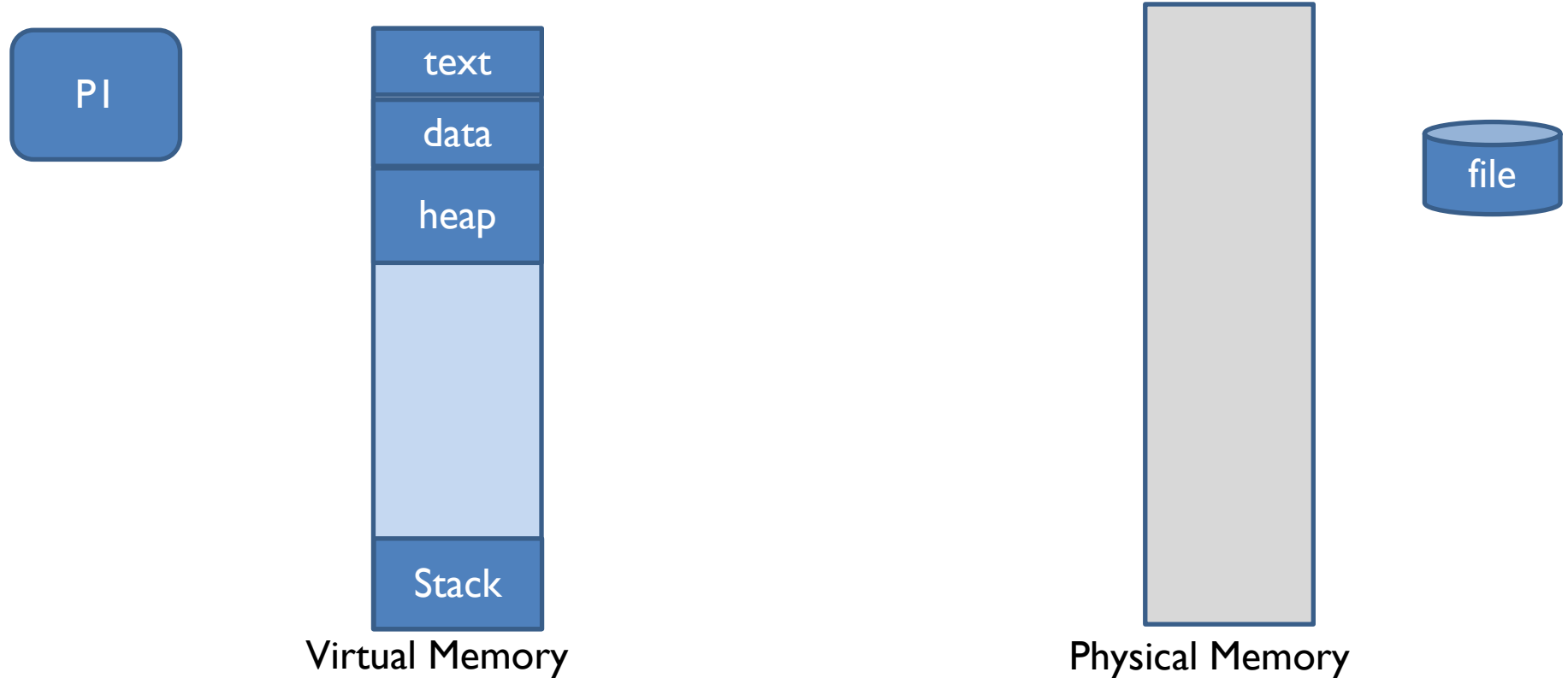
- `mmap(0, 8192, PROT_READ, MAP_POPULATE, fd, 4096)`
 - `mmap 2pages`



How file mmap() Works

2) Private file mapping **without** MAP_POPULATE

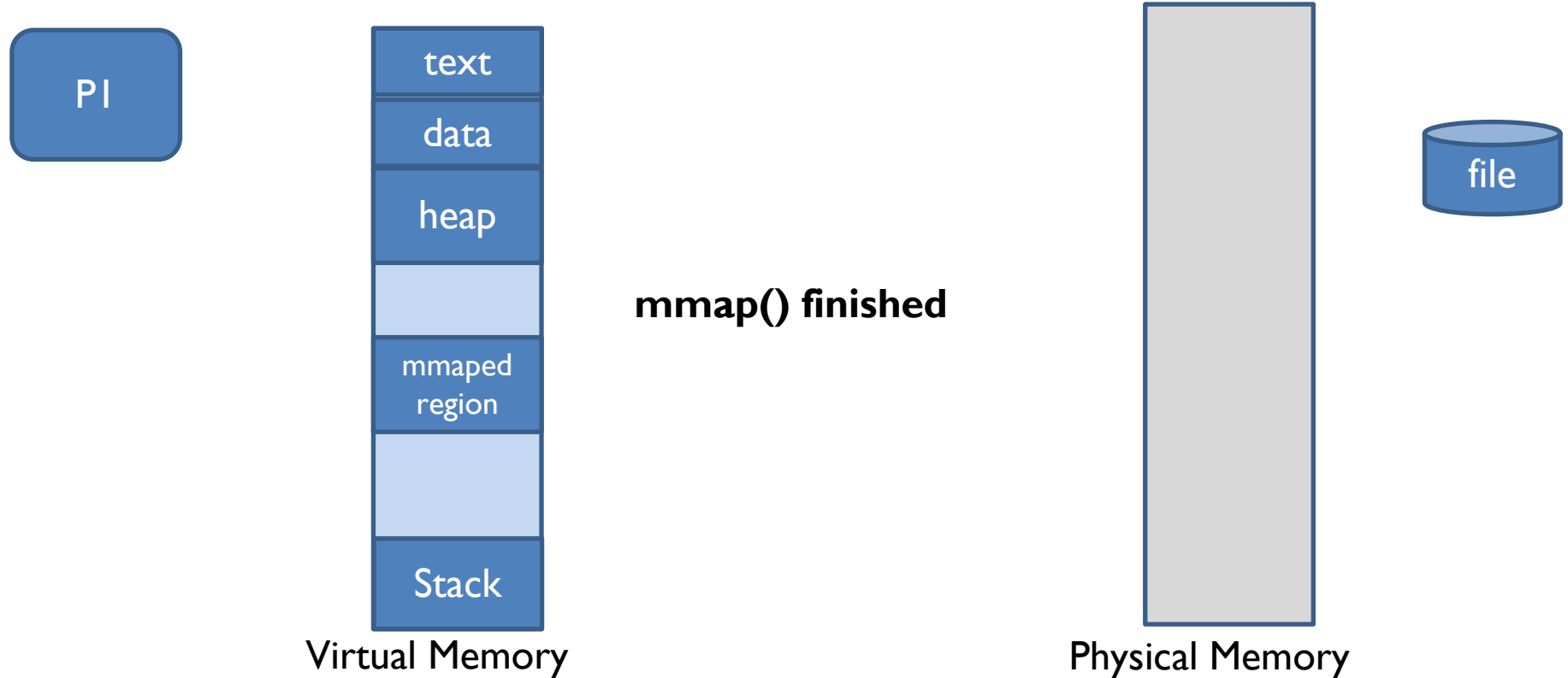
- `mmap(0, 8192, PROT_READ, 0, fd, 4096)`
 - mmap 2pages



How file mmap() Works

2) Private file mapping **without** MAP_POPULATE

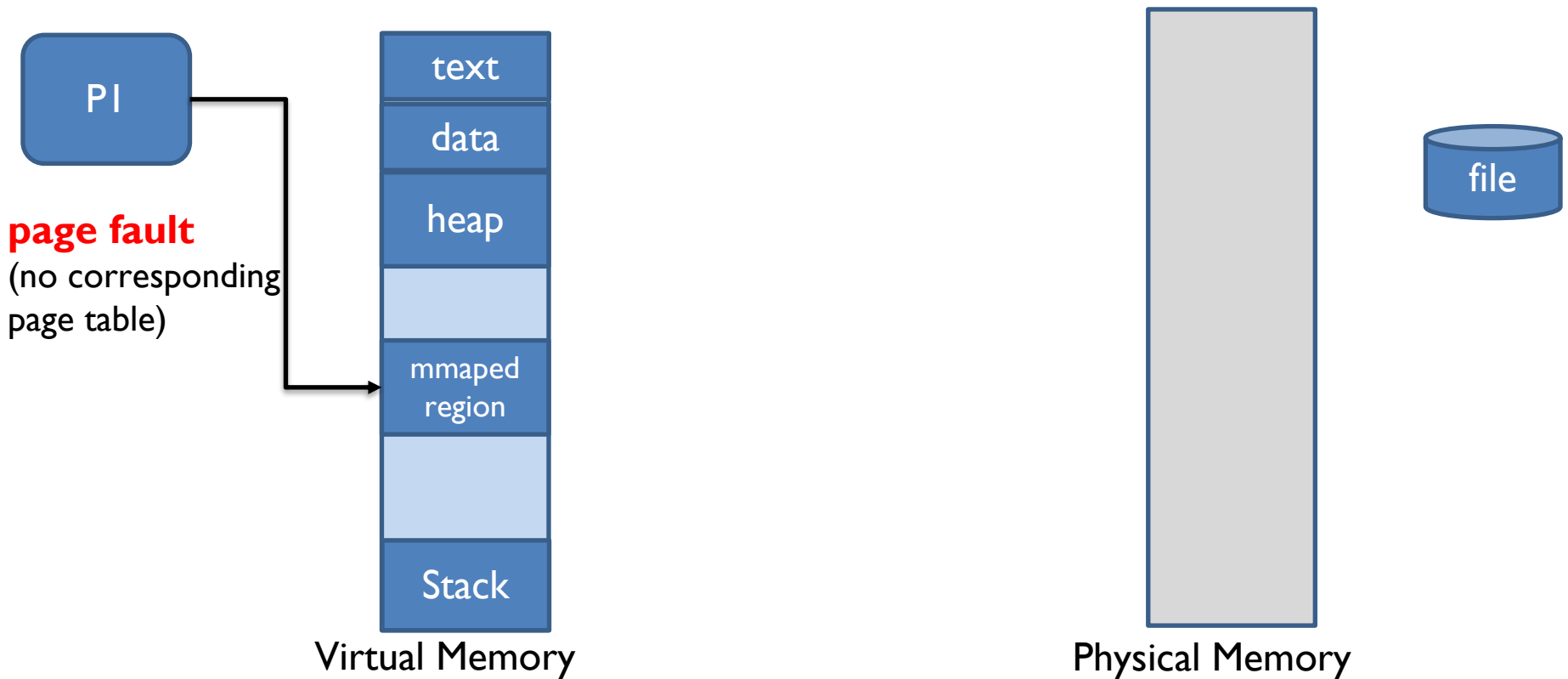
- `mmap(0, 8192, PROT_READ, 0, fd, 4096)`
 - `mmap 2pages`



How file mmap() Works

2) Private file mapping **without** MAP_POPULATE

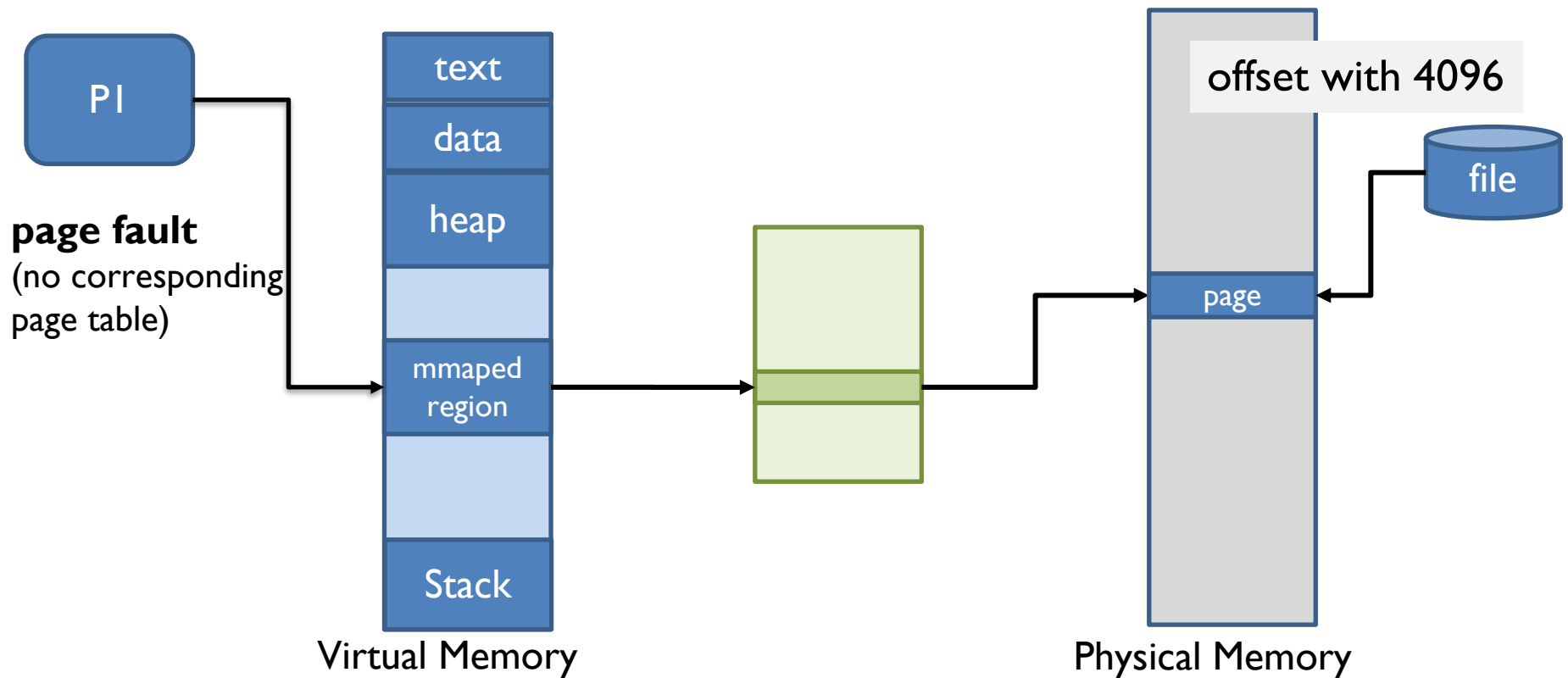
- `mmap(0, 8192, PROT_READ, 0, fd, 4096)`
 - `mmap 2pages`



How file mmap() Works

2) Private file mapping **without** MAP_POPULATE

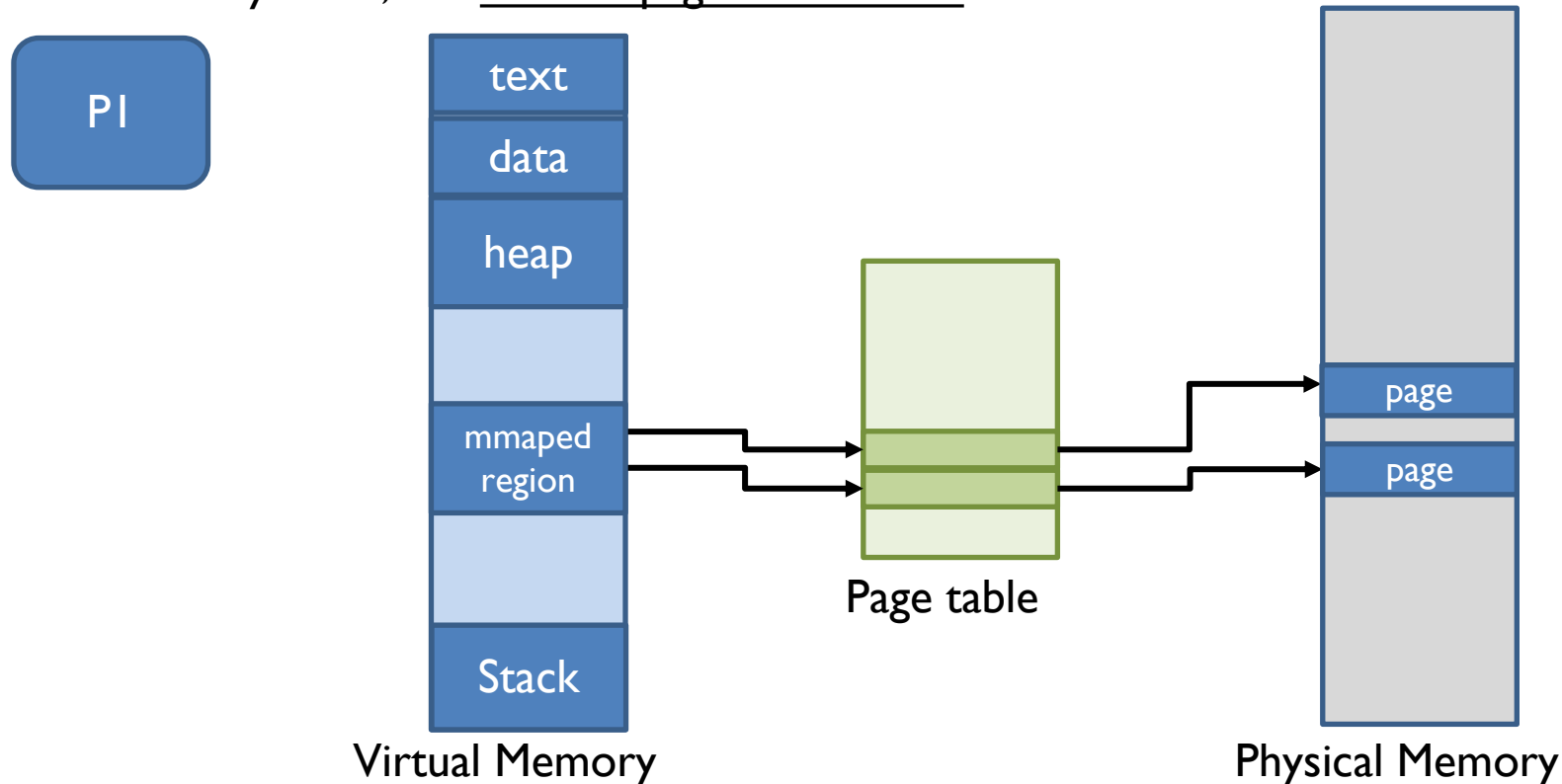
- `mmap(0, 8192, PROT_READ, 0, fd, 4096)`
 - `mmap 2pages`



How anonymous mmap() Works

3) Private anonymous mapping with **MAP_POPULATE**

- `mmap(0, 8192, PROT_READ, MAP_POPULATE|MAP_ANONYMOUS, -1, 0)`
 - `mmap 2pages`
- Mostly same, but allocate page filled with 0



Implementation detail of mmap()

- Parameters will be defined at param.h
 - PROT_READ 0x1
 - PROT_WRITE 0x2
 - MAP_ANONYMOUS 0x1
 - MAP_POPULATE 0x2

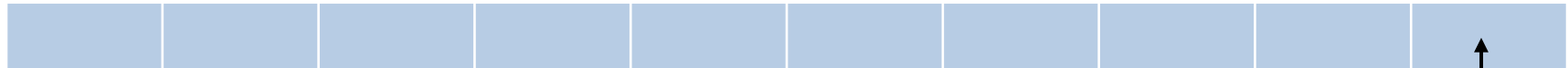
Implementation detail of mmap()

- ```
struct mmap_area {
 strict file *f;
 uint addr;
 int length;
 int offset;
 int prot;
 int flags;
 struct proc *p // the process with this mmap_area
}
```
- Manage all mmap areas created by each mmap() call in one mmap\_area array.
- Maximum number of mmap\_area array is 64.

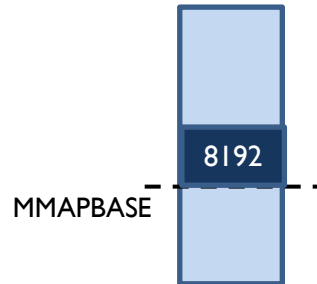
# mmap\_area Array Example

In the case of populate...

struct mmap\_area array



PI



Virtual Memory

`mmap(0, 8192, PROT_READ, MAP_POPULATE, fd, 4096)`

P2



Virtual Memory



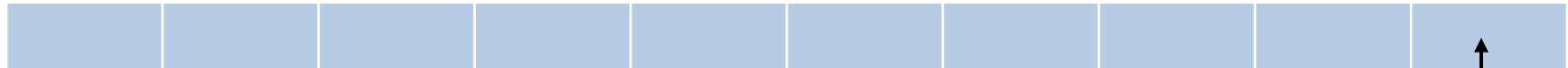
Physical Memory

file \*f;  
addr;  
length;  
offset;  
prot;  
flags;  
proc \*p;

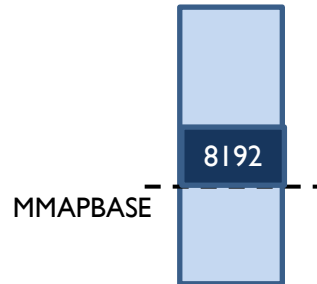
# mmap\_area Array Example

In the case of populate...

struct mmap\_area array



PI



Virtual Memory

`mmap(0, 8192, PROT_READ, MAP_POPULATE, fd, 4096)`

P2



Virtual Memory



Physical Memory

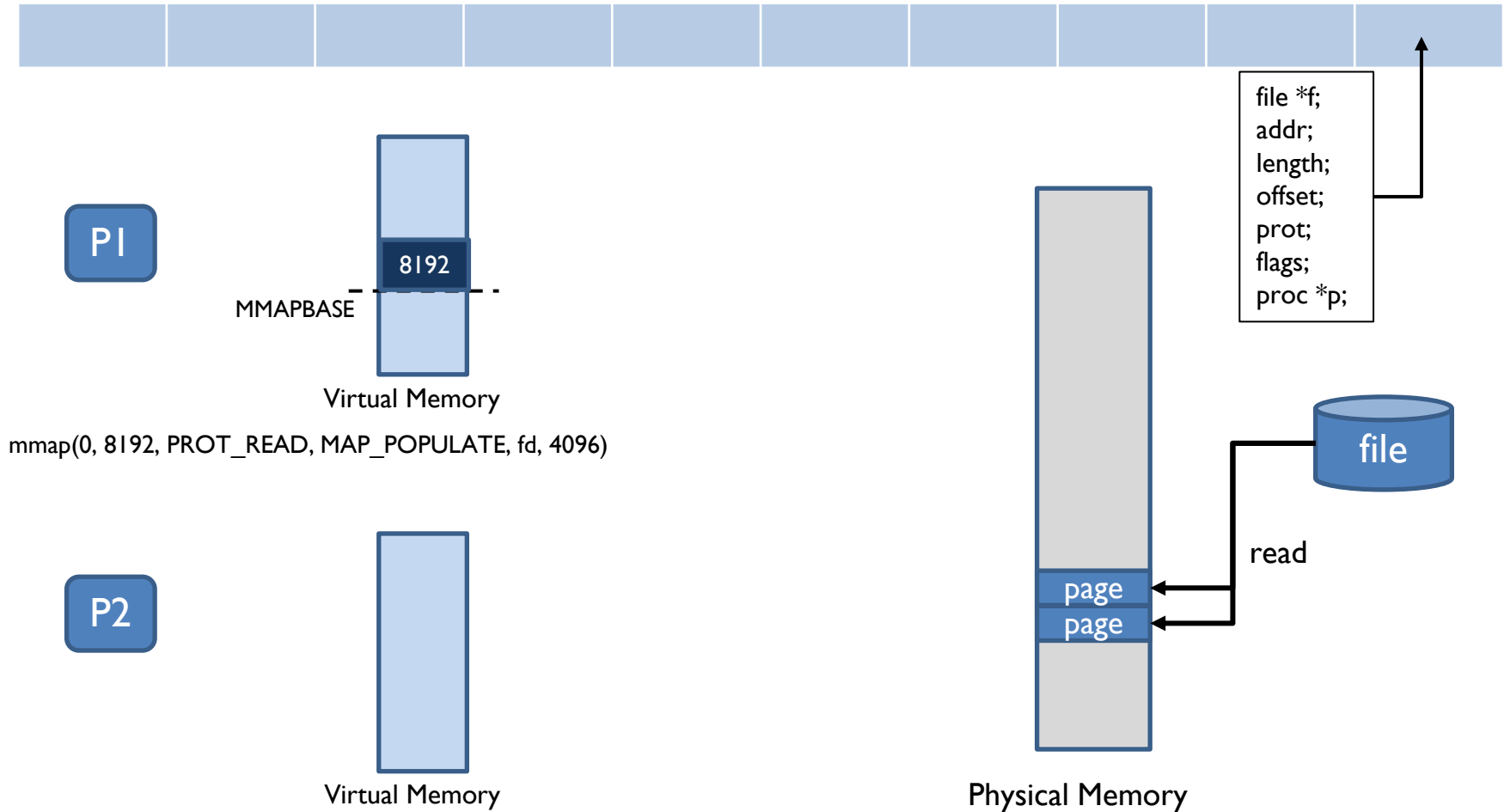
file \*f;  
addr;  
length;  
offset;  
prot;  
flags;  
proc \*p;

get physical page

# mmap\_area Array Example

In the case of populate...

struct mmap\_area array

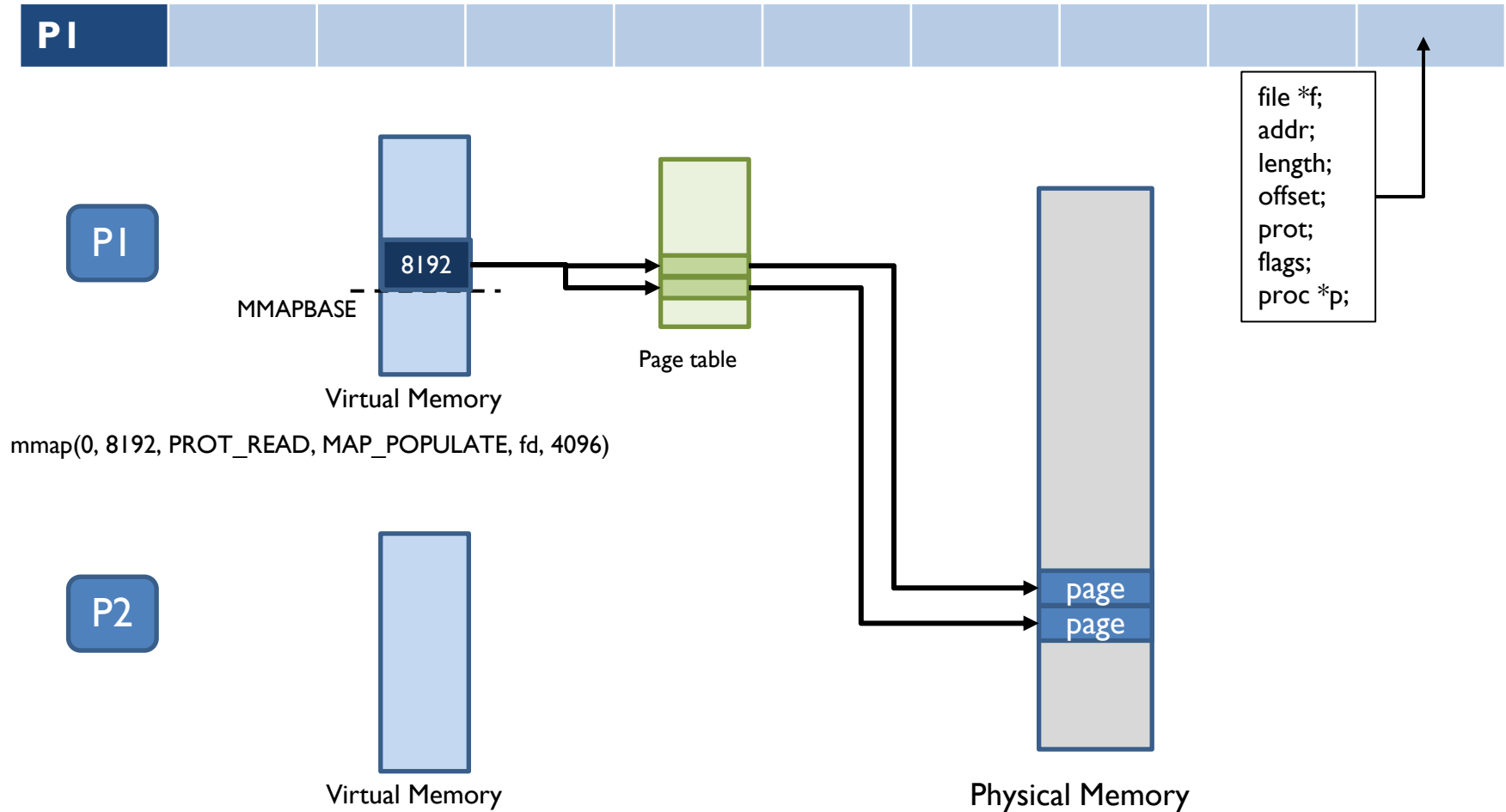




# mmap\_area Array Example

In the case of populate...

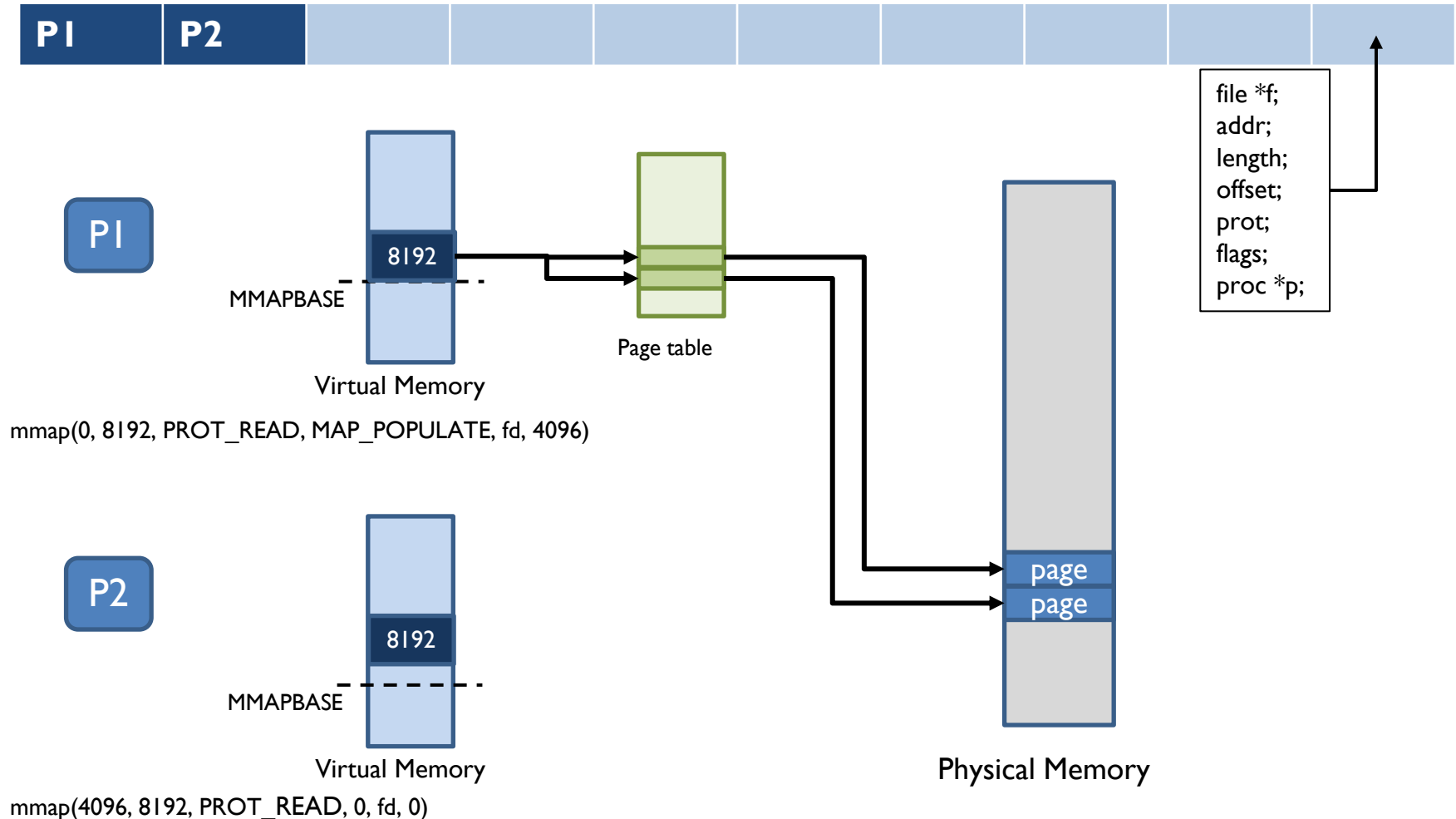
struct mmap\_area array



# mmap\_area Array Example

In the case of populate...

struct mmap\_area array



## 2. Page Fault Handler on xv6

- **Page fault handler is for dealing with access on mapping region with physical page & page table is not allocated**
  - **Succeed:** Physical pages and page table entries are created normally, and the process works without any problems
  - **Failed:** The process is terminated
1. When an access occurs (read/write), catch according page fault (interrupt 14, T\_PGFLT) in *traps.h*
  2. In page fault handler, determine fault address by reading CR2 register(using rcr2()) & access was read or write  
read: `tf->err&2 == 0` / write: `tf->err&2 == 1`
  3. Find according mapping region in `mmap_area`  
If faulted address has no corresponding `mmap_area`, return -1
  4. If fault was write while `mmap_area` is write prohibited, then return -1
  5. For only one page according to faulted address
    1. Allocate new physical page
    2. Fill new page with 0
    3. If it is file mapping, read file into the physical page with offset
    4. If it is anonymous mapping, just left the page which is filled with 0s
    5. Make page table & fill it properly (if it was PROT\_WRITE, PTE\_W should be 1 in PTE value)

# 3. munmap() system call on xv6

```
int munmap(uint addr)
```

- **Unmaps corresponding mapping area**
- **Return value: 1(succeed), -1(failed)**
  1. *addr* will be always given with the start address of mapping region, which is page aligned
  2. `munmap()` should remove corresponding `mmap_area` structure
    - If there is no `mmap_area` of process starting with the address, return -1
  3. If physical page is allocated & page table is constructed, should free physical page & page table
    - When freeing the physical page should fill with 1 and put it back to freelist
  4. If physical page is not allocated (page fault has not been occurred on that address), just remove `mmap_area` structure.
  5. Notice) In one `mmap_area`, situation of some of pages are allocated and some are not can happen.

## 4. freemem() system call on xv6

```
int freemem()
```

- syscall to return current number of free memory pages
1. When kernel frees (put page into free list), freemem should be increase
  2. When kernel allocates (takes page from free list and give it to process), freemem should decrease

# How to test

- Only README file will be used on testing filemap
- Do not modify README file
- Your test code will include below header
- Test case
  - Anonymous mapping with populate & without populate
  - File mapping with populate & without populate
  - Freemem test between each case
  - Comparison of file mapped contents before and after fork
    - Contents should be same
    - Freemem test after fork

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"
#include "memlayout.h"
#include "mmu.h"
#include "param.h"
#include "spinlock.h"
#include "sleeplock.h"
#include "fs.h"
#include "proc.h"
#include "syscall.h"
```

# FAQ

- Mmap address range
  - All address will not exceed unsigned int range (address range is not so large, no overflow)
- Page fault
  - If page fault occurs, allocate physical memory page of that virtual address page range
- Mmap return address
  - MMAPBASE + addr
- OOM control
  - Test code will not allocate many memory page, don't care
- Mmap bigger than file size
  - Mmap size will not big. don't care
- Child memory control
  - When fork occurs, you should copy parent's page table and map physical page same as parent
- Mmap range overlap
  - Don't care. Will not be tested
- Mmap/munmap page align
  - Address argument must be page aligned, if not, return 0

# FAQ

- Dirty page control
  - Mmapped page write will not be tested.
- Invalid file descriptor
  - On test, we will use only README file. Don't care
- File mapped contents print
  - On test, file mapped content will be tested like this

```
printf(1, "- fd data: %c %c %c\n", test3[0], test3[1], test3[2]);
```
- Test file size
  - The size of the README file is too small to test various sizes of mmap, but in the actual test, we will increase the size of the README file before proceeding with the test. You may also create a large file in a similar manner for experimentation, but when submitting, please submit the original README file.
- Function definition
  - Mmap, freemem, munmap function must be defined and implemented exactly as written in the synopsis.



# Submission

- This project is to implement three system calls and page fault handler
  - mmap() syscall
  - Page fault handler
  - munmap() syscall
  - freemem() syscall
- Use the *submit* & *check-submission* binary file in Ui Server
  - \$ ~swe3004/bin/submit pa3 xv6-public
  - **Make clean**
  - you can submit several times, and the submission history can be checked through check-submission
    - Only the last submission will be graded

# Submission

- PLEASE DO NOT COPY
  - We will run inspection program on all the submissions
  - Any unannounced penalty can be given to **both students**
    - 0 points / negative points / F grade ...
- Due date: 11/13(Wed.), 23:59:59 PM
  - -25% per day for delayed submission

# Questions

- If you have questions, please ask on icampus
  - Please use the discussion board
  - We don't reply messages
- You can also visit Corporate Collaboration Center #85533
  - Please e-mail TA before visiting
- Reading xv6 commentary will help you a lot
  - <http://csl.skku.edu/uploads/SSE3044S20/book-rev11.pdf>

# Appendix. Hint

- **File structures** corresponding to fd are contained in `proc->ofile[fd]`
  - File structure can be used to get file data and file protection
- Page table entry can be created using **mappages** as in `copyvm`
- **At fork time**, if the parent process has mmap areas the child process will also have mmap areas at the same address, so this needs to be processed
- Page fault invokes **the trap function** in `trap.c` after similar processing of system calls in Project 2.
  - Here, you can utilize `rcr2()` and `tf->err`