SWE3004 Operating Systems, Fall 2025

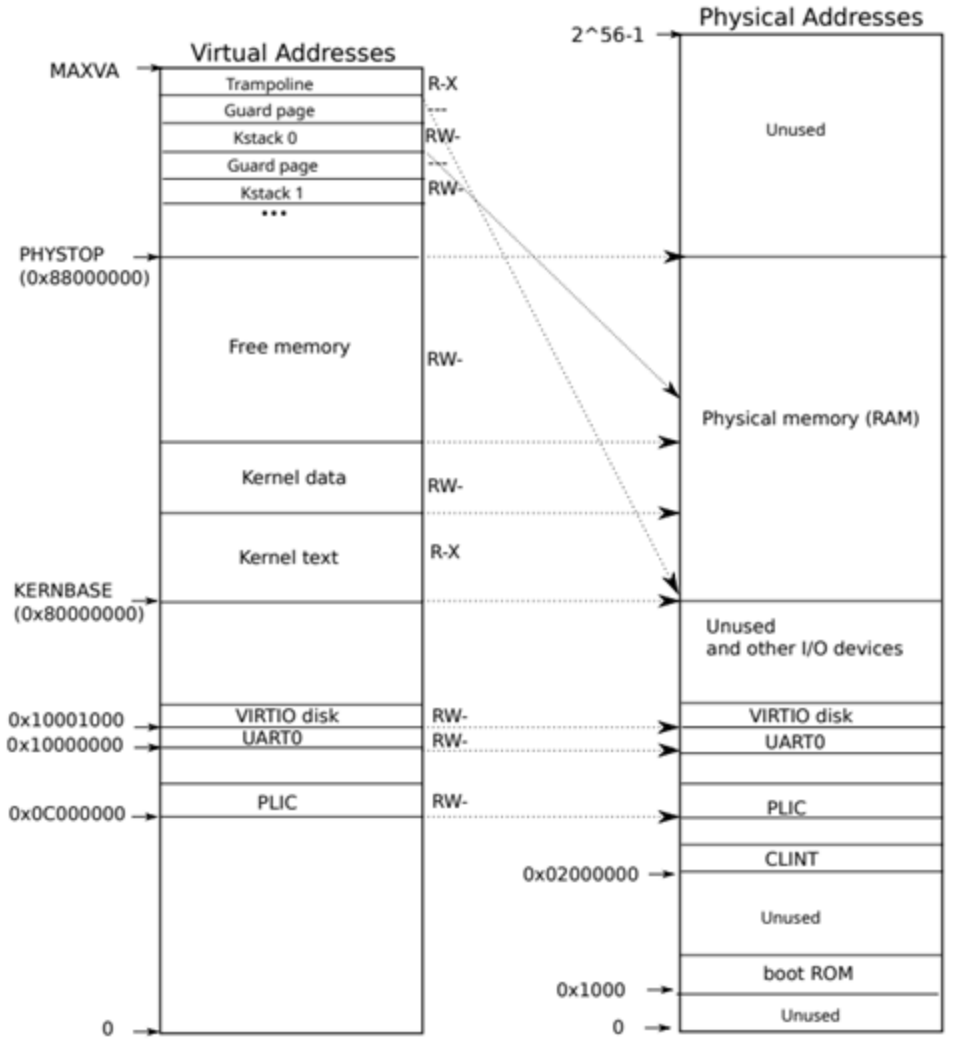
Project 3. Virtual Memory

TA)
Gwanjong Park
Yunseong Shin

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 /kernel/main.c

```
void
main()
{
  if(cpuid() == 0){
    consoleinit();
    printfinit();
    printf("\n");
    printf("xv6 kernel is booting\n");
    printf("\n");
    kinit();         // physical page allocator
    kvminit();       // create kernel page table
    kvminithart();   // turn on paging
    procinit();      // process table
    trapinit();      // trap vectors
    trapinithart();  // install kernel trap vector
    plicinit();      // set up interrupt controller
    plicinithart();  // ask PLIC for device interrupts
    binit();         // buffer cache
    iinit();         // inode table
    fileinit();      // file table
    virtio_disk_init(); // emulated hard disk
    userinit();      // first user process
    __sync_synchronize();
```

This function divides & manages physical memories with pages

How Physical Memories Initialized in xv6

- kinit(), freerange(), kfree() of /kernel/kalloc.c

```
void
kinit()
{
  initlock(&kmem.lock, "kmem");
  freerange(end, (void*)PHYSTOP);
}

void
freerange(void *pa_start, void *pa_end)
{
  char *p;
  p = (char*)PGROUNDUP((uint64)pa_start);
  for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
    kfree(p);
}
```

```
void
kfree(void *pa)
{
  struct run *r;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    panic("kfree");

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

  r = (struct run*)pa;

  acquire(&kmem.lock);
  r->next = kmem.freelist;
  kmem.freelist = r;
  release(&kmem.lock);
}
```

- kinit() arranges for memory (until PHYSTOP) to be allocatable (128MB)
- freerange() calls kfree() with page size unit (4KB)
- kfree() fills page with 1s, and put it into freelist (page pool)

How Physical Memories Initialized in xv6

- fork() of /kernel/proc.c / uvmcopy() of /kernel/vm.c

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

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

  // Copy user memory from parent to child.
  if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
  }
  np->sz = p->sz;

  // copy saved user registers.
  *(np->trapframe) = *(p->trapframe);

  // Cause fork to return 0 in the child.
  np->trapframe->a0 = 0;

  // increment reference counts on open file descriptors.
  for(i = 0; i < NOFILE; i++)
    if(p->ofile[i])
      np->ofile[i] = filedup(p->ofile[i]);
  np->cwd = idup(p->cwd);

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

  pid = np->pid;
```

- fork() creates a child with exactly the same memory contents as the parent
- allocproc() initializes basic kernel structure to create a new user-process.
- uvmcopy() copys parent's page table

```
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  pte_t *pte;
  uint64 pa, i;
  uint flags;
  char *mem;

  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walk(old, i, 0)) == 0)
      panic("uvmcopy: pte should exist");
    if((*pte & PTE_V) == 0)
      panic("uvmcopy: page not present");
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto err;
    memmove(mem, (char*)pa, PGSIZE);
    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
      kfree(mem);
      goto err;
    }
```

Project 3: Implement Simple mmap()

- **Goal: Implement three system calls and page fault handler on xv6**
- **What your code should handle**

1. mmap() system call
2. Page fault handler
3. munmap() system call
4. freemem() system call

1. mmap() system call on xv6

- Simple mmap() synopsis

```
uint64 mmap(uint64 addr, int length, int prot, int flags, int fd, int offset)
```

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

```
uint64 mmap(uint64 addr, int length, 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

```
uint64 mmap(uint64 addr, int length, 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

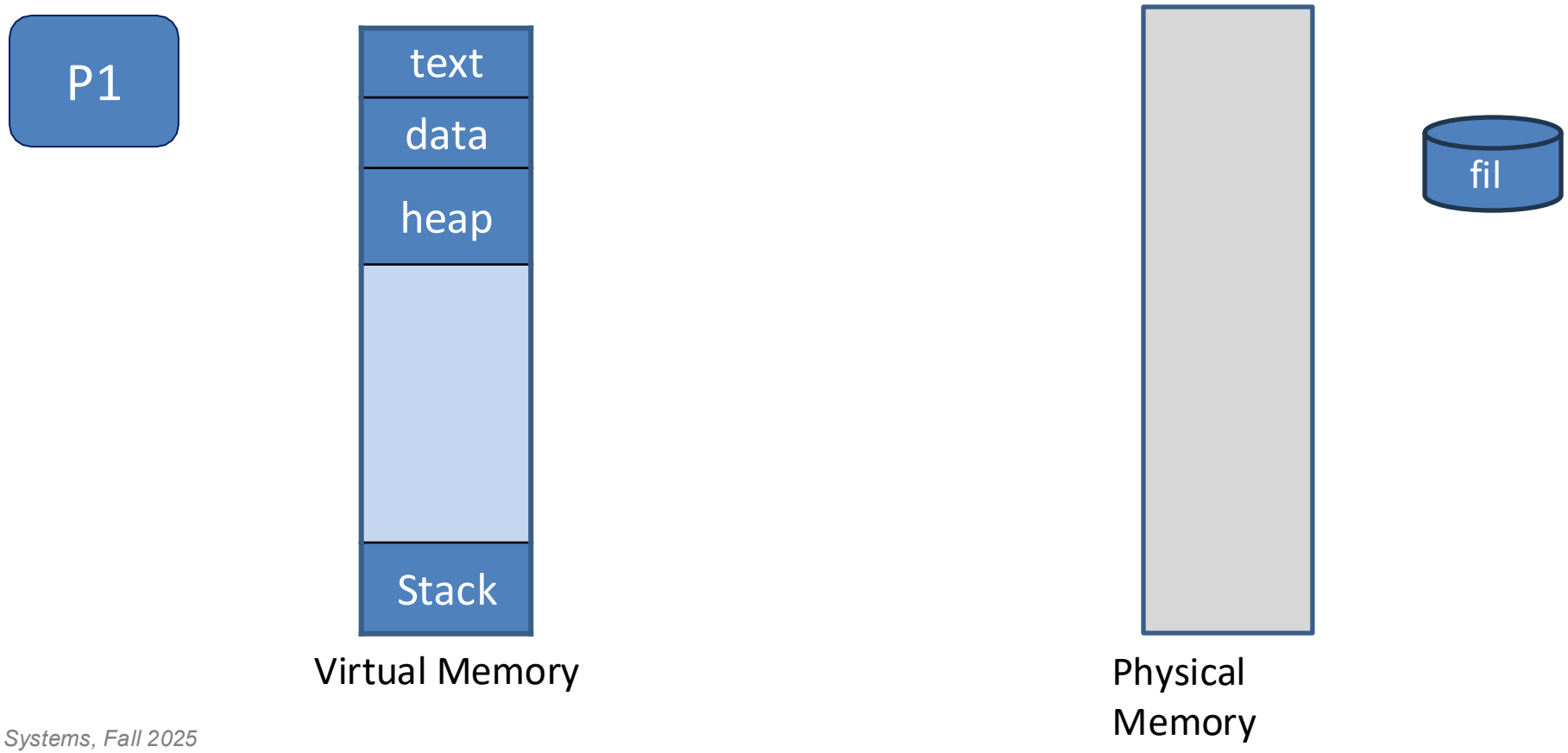
- It's not anonymous, but when the fd is -1
- The protection of the file and the prot of the parameter are different
- The situation in which the mapping area is overlapped is not considered
- If additional errors occur, we will let you know by writing notification

How file mmap() Works

1) Private file mapping wih **MAP_POPULATE**

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

  - mmap 2pages

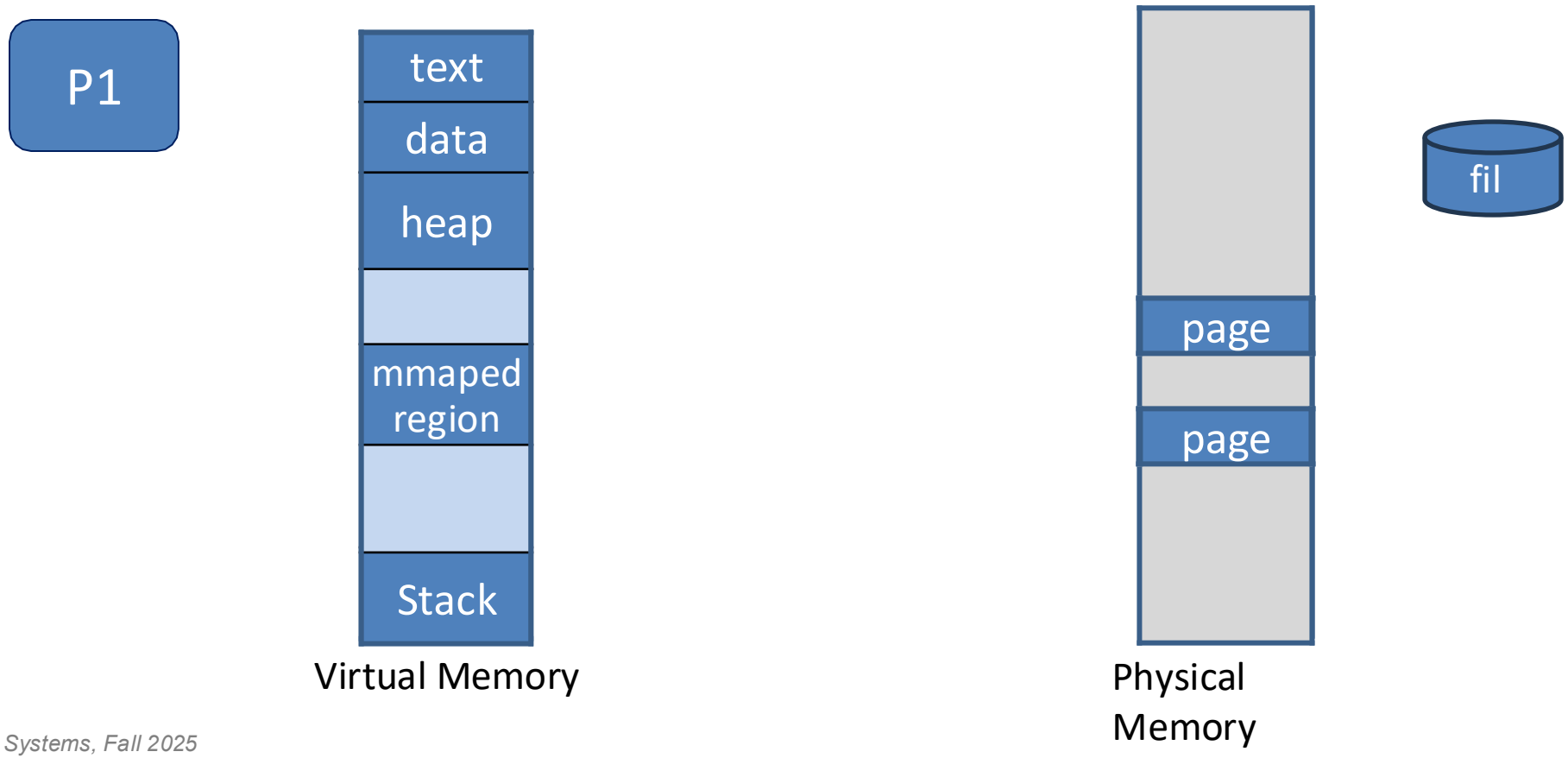


How file mmap() Works

1) Private file mapping wih **MAP_POPULATE**

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

  - mmap 2pages

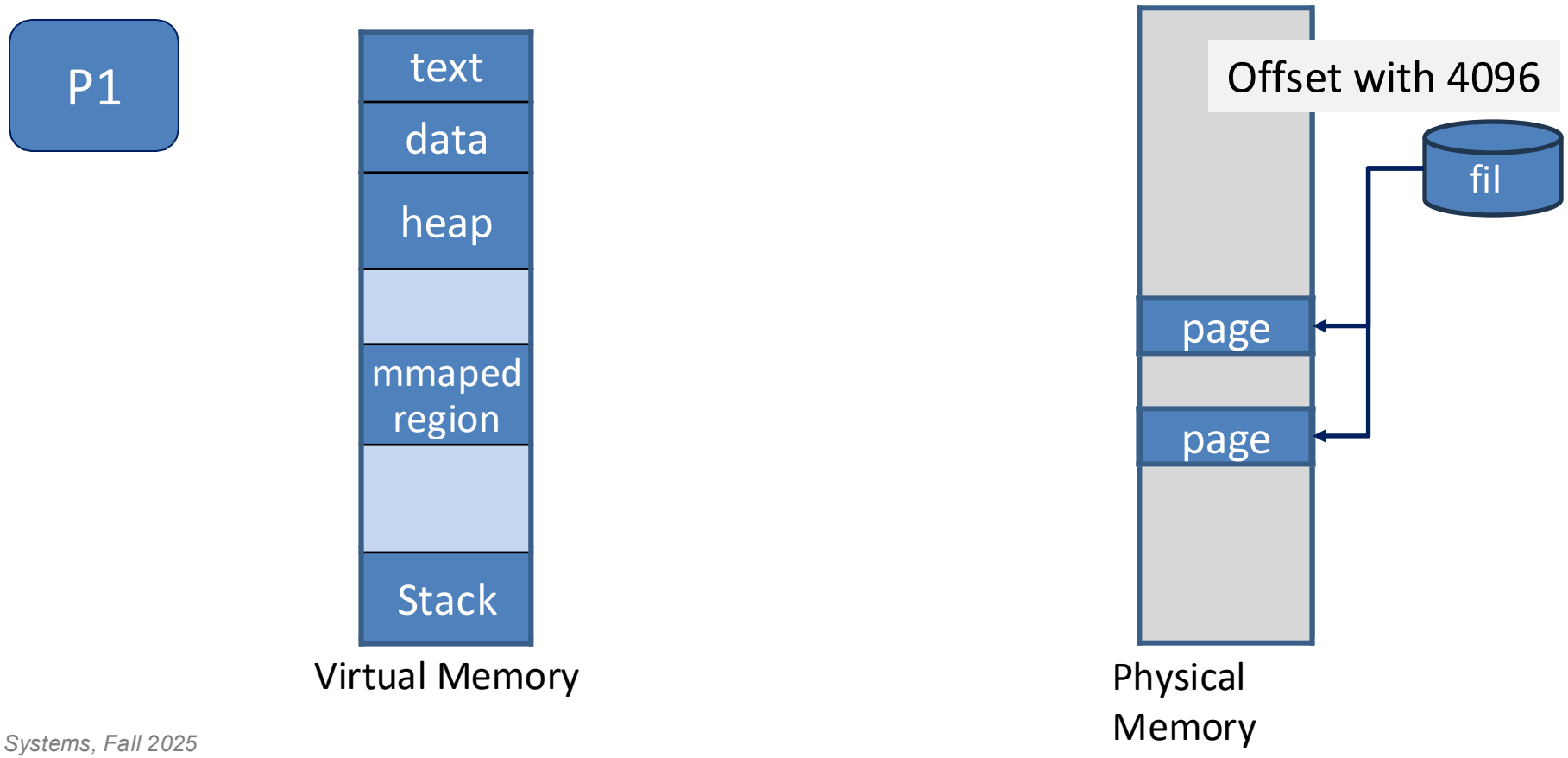


How file mmap() Works

1) Private file mapping wih **MAP_POPULATE**

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

  - mmap 2pages

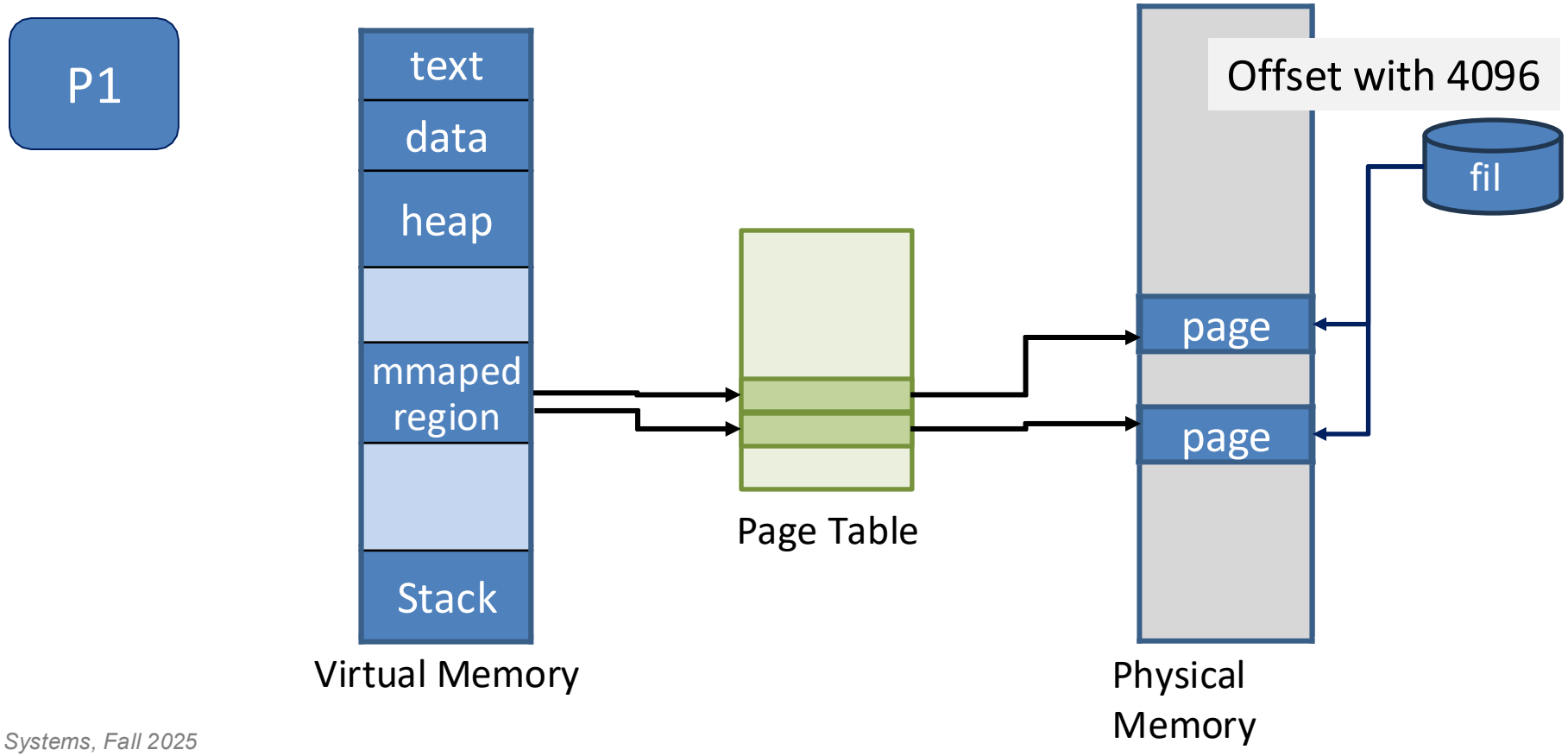


How file mmap() Works

1) Private file mapping wih **MAP_POPULATE**

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

  - mmap 2pages

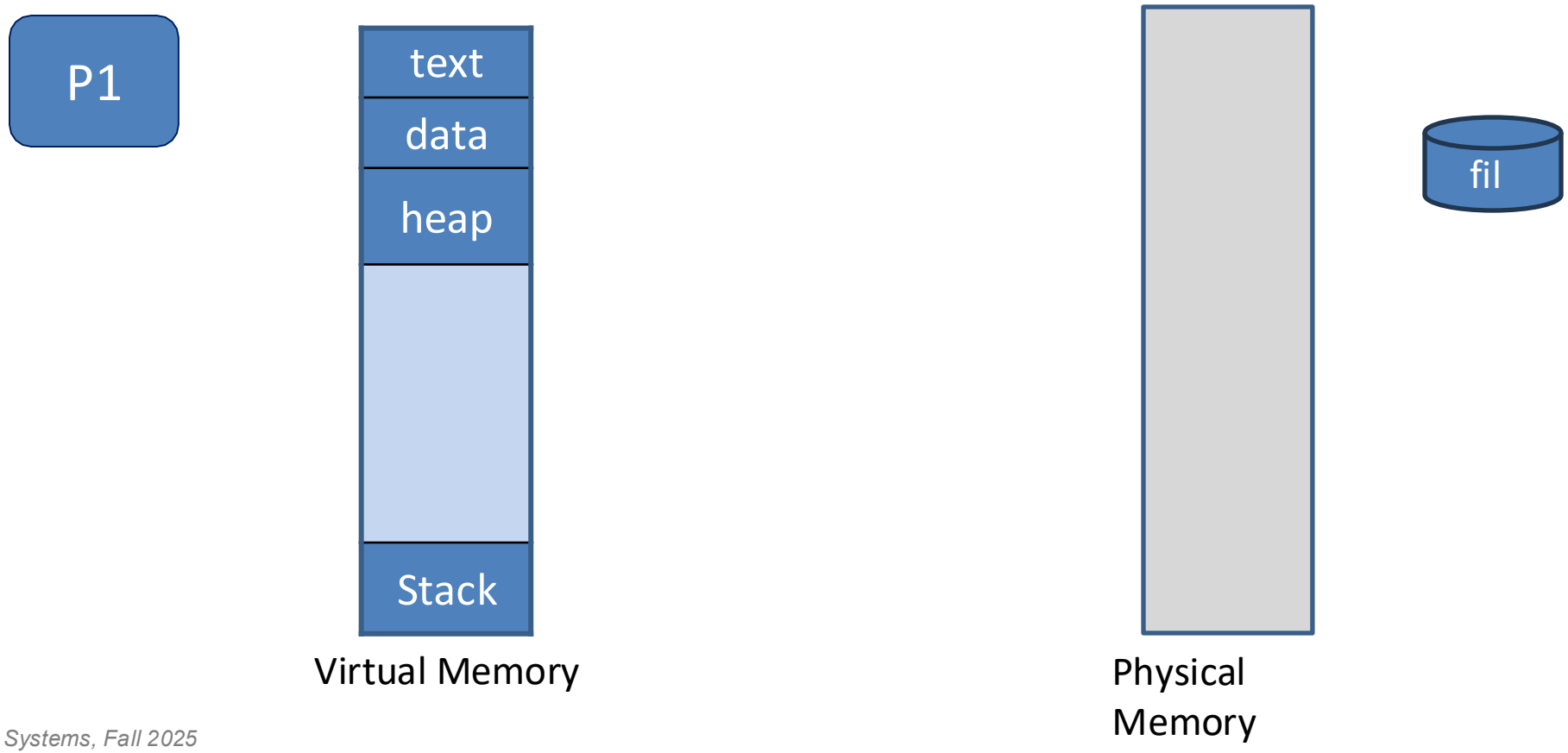


How file mmap() Works

2) Private file mapping **wihout MAP_POPULATE**

- mmap(0, 8192, PROT_READ, 0, fd, 4096)

  - mmap 2pages

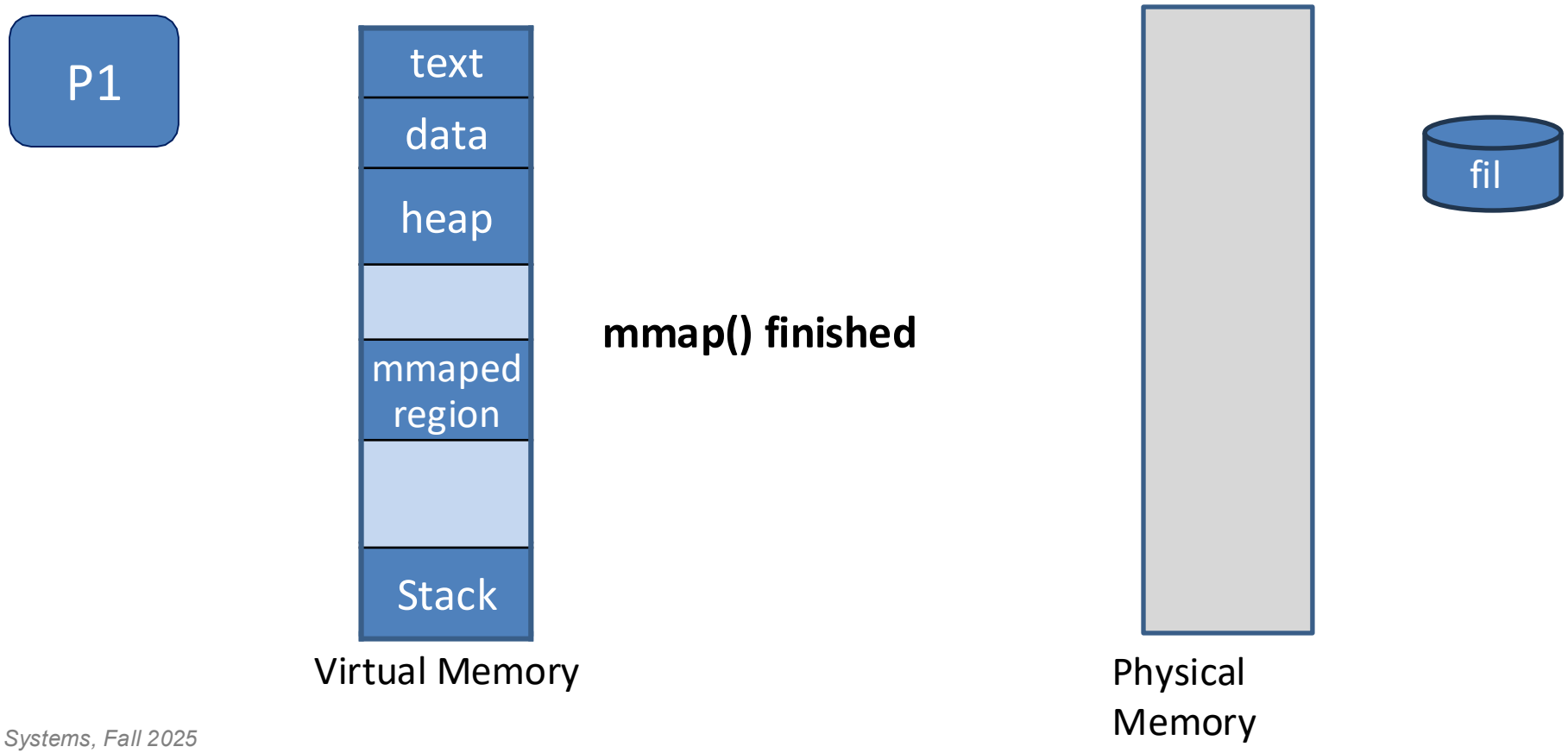


How file mmap() Works

2) Private file mapping **wihout MAP_POPULATE**

- mmap(0, 8192, PROT_READ, 0, fd, 4096)

  - mmap 2pages

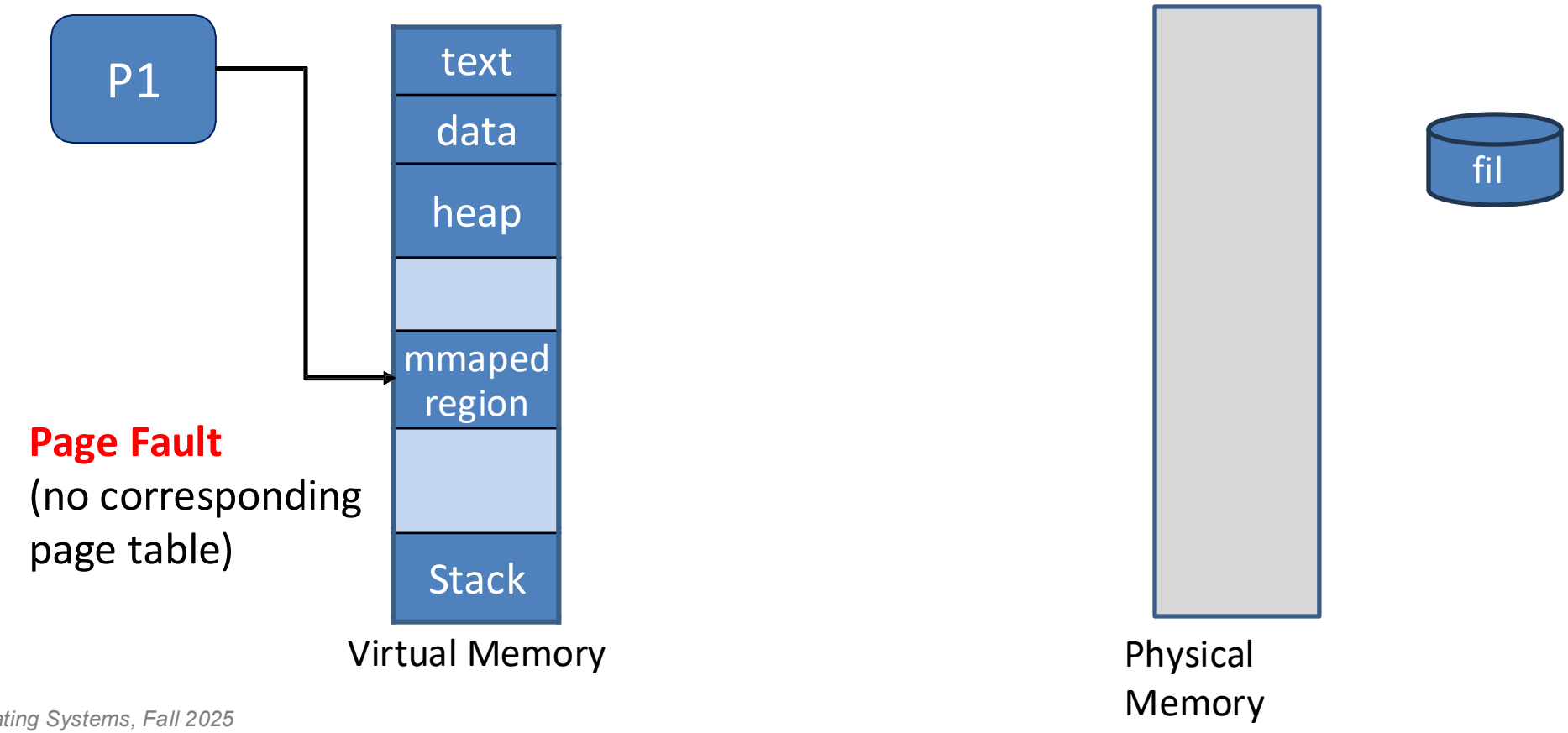


How file mmap() Works

2) Private file mapping **wihout MAP_POPULATE**

- mmap(0, 8192, PROT_READ, 0, fd, 4096)

  - mmap 2pages

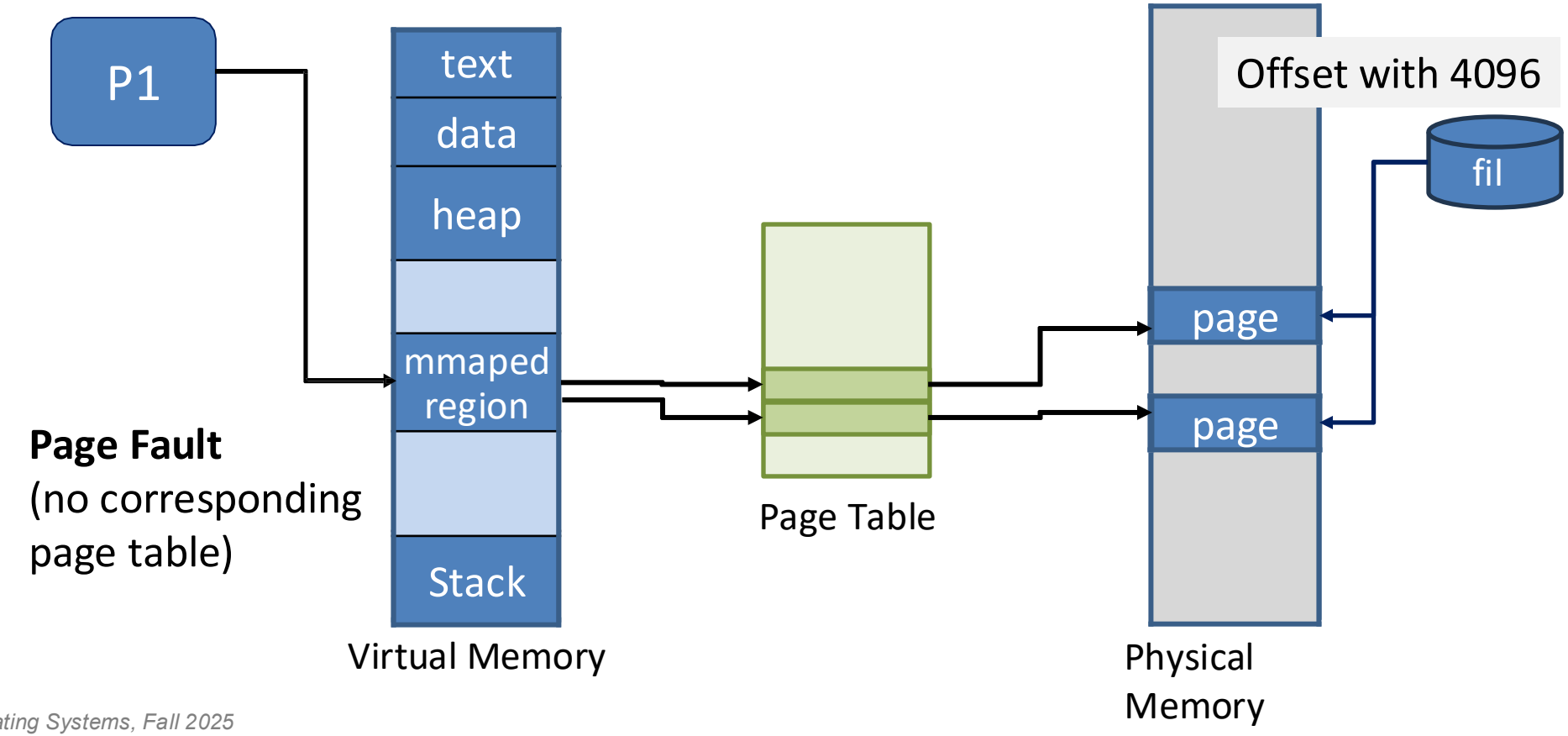


How file mmap() Works

2) Private file mapping **wihout MAP_POPULATE**

- mmap(0, 8192, PROT_READ, 0, fd, 4096)

  - mmap 2pages

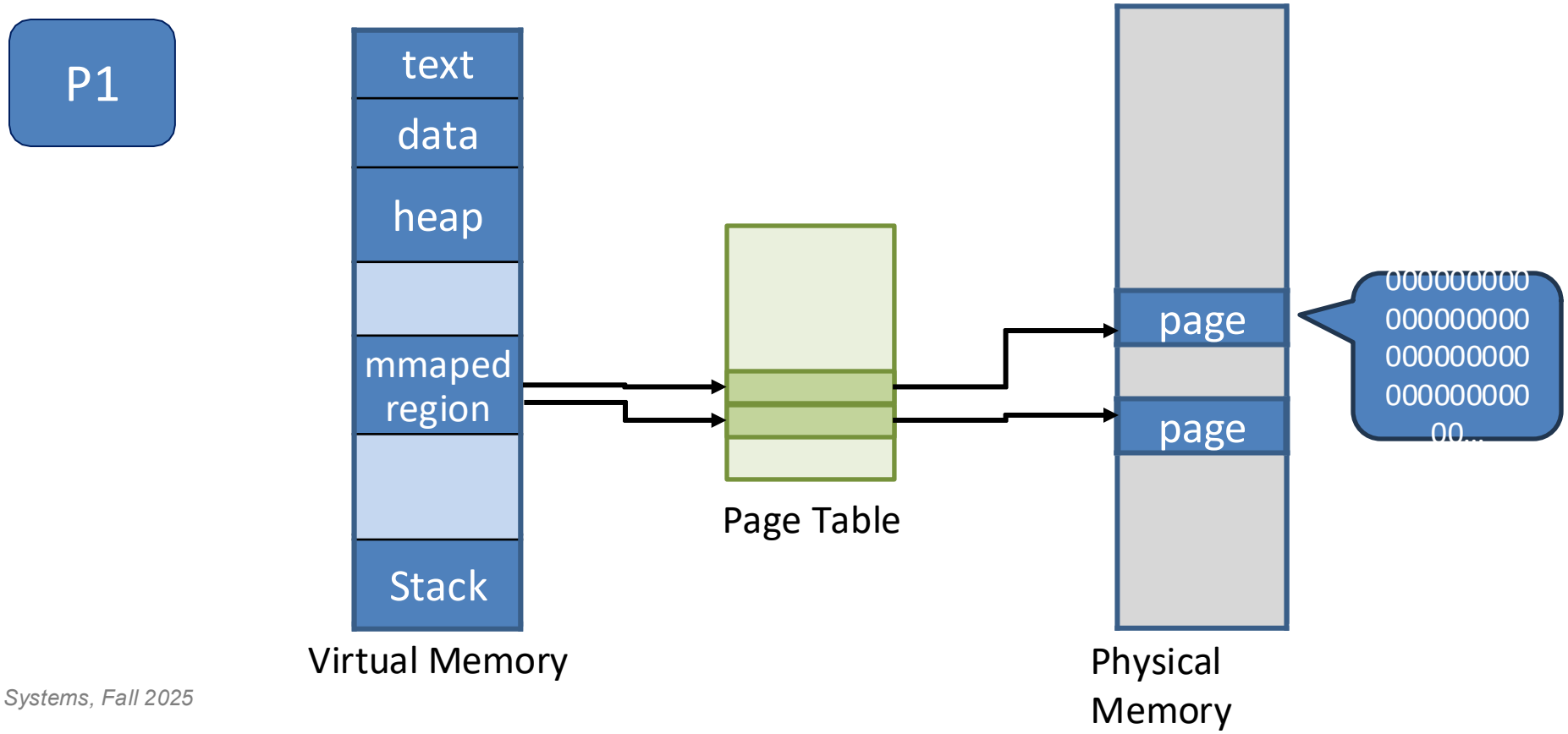


How file mmap() Works

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 0s



Implementation detail of mmap()

- **Parameters should be defined at /kernel/param.h**
  - PROT_READ 0x1
  - PROT_WRITE 0x2

  - MAP_ANONYMOUS 0x1
  - MAP_POPULATE 0x2

Implementation detail of mmap()

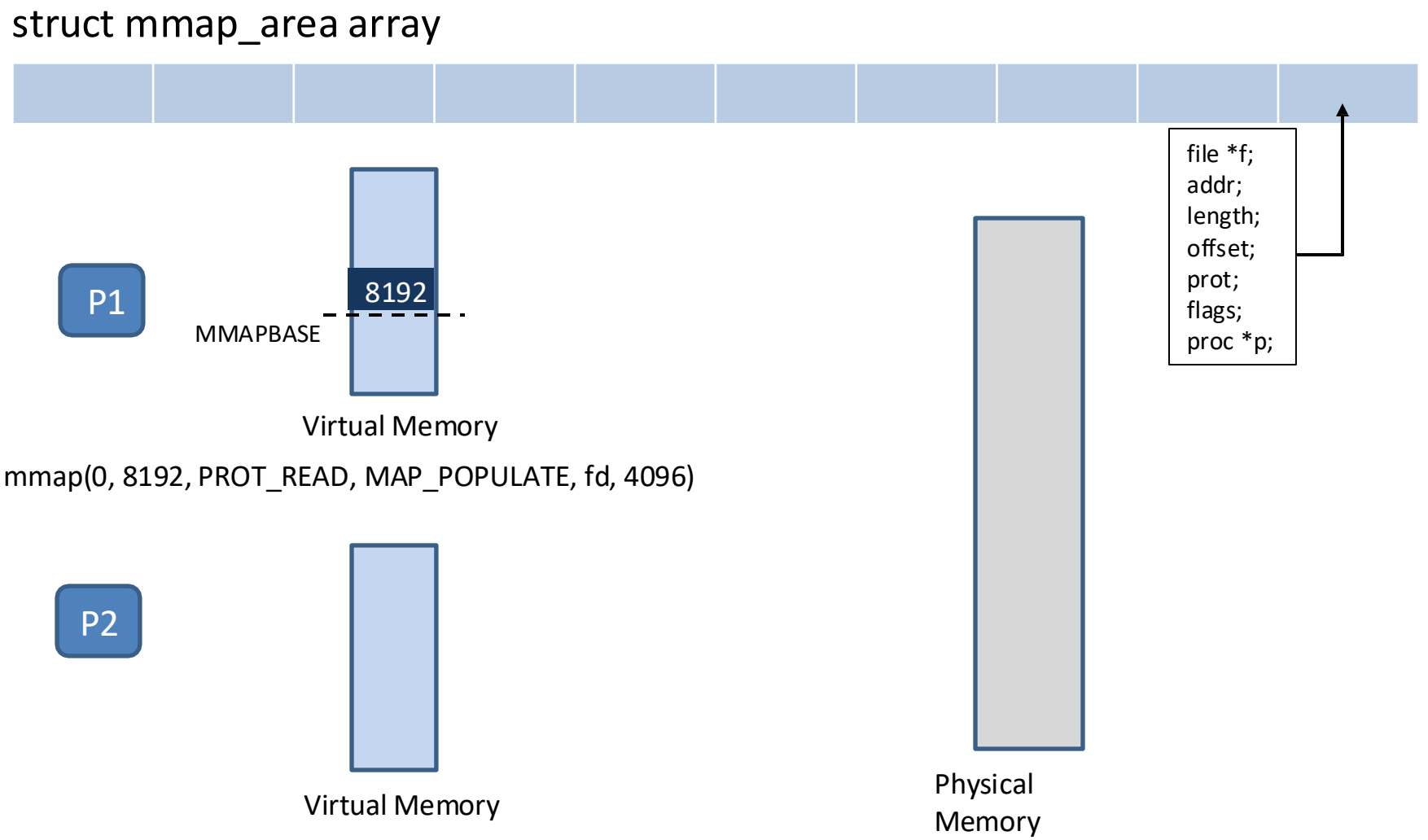
- struct mmap_area {

```
        struct file *f;
        uint64 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**.

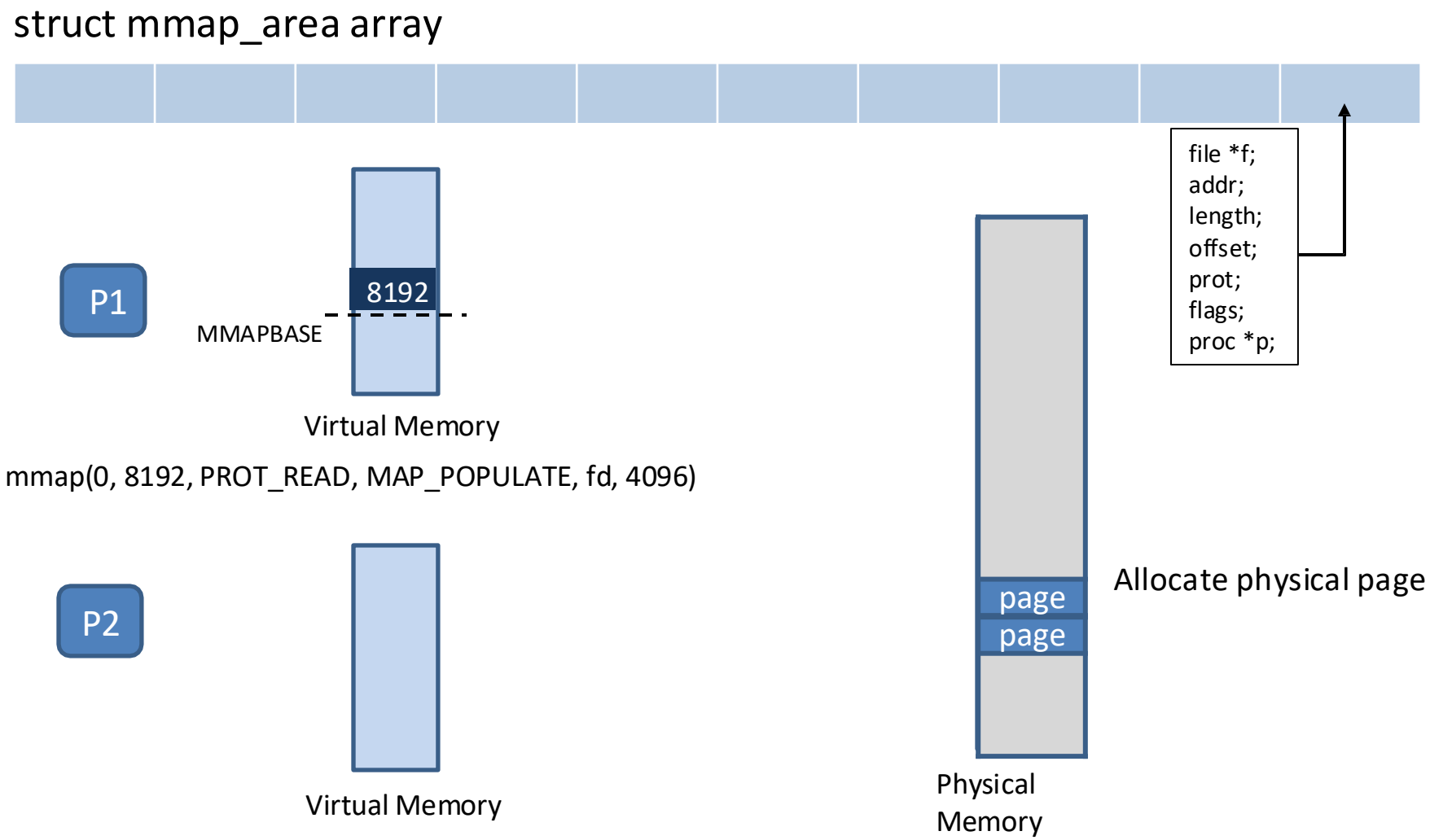
mmap_area Array Example

In the case of populate…



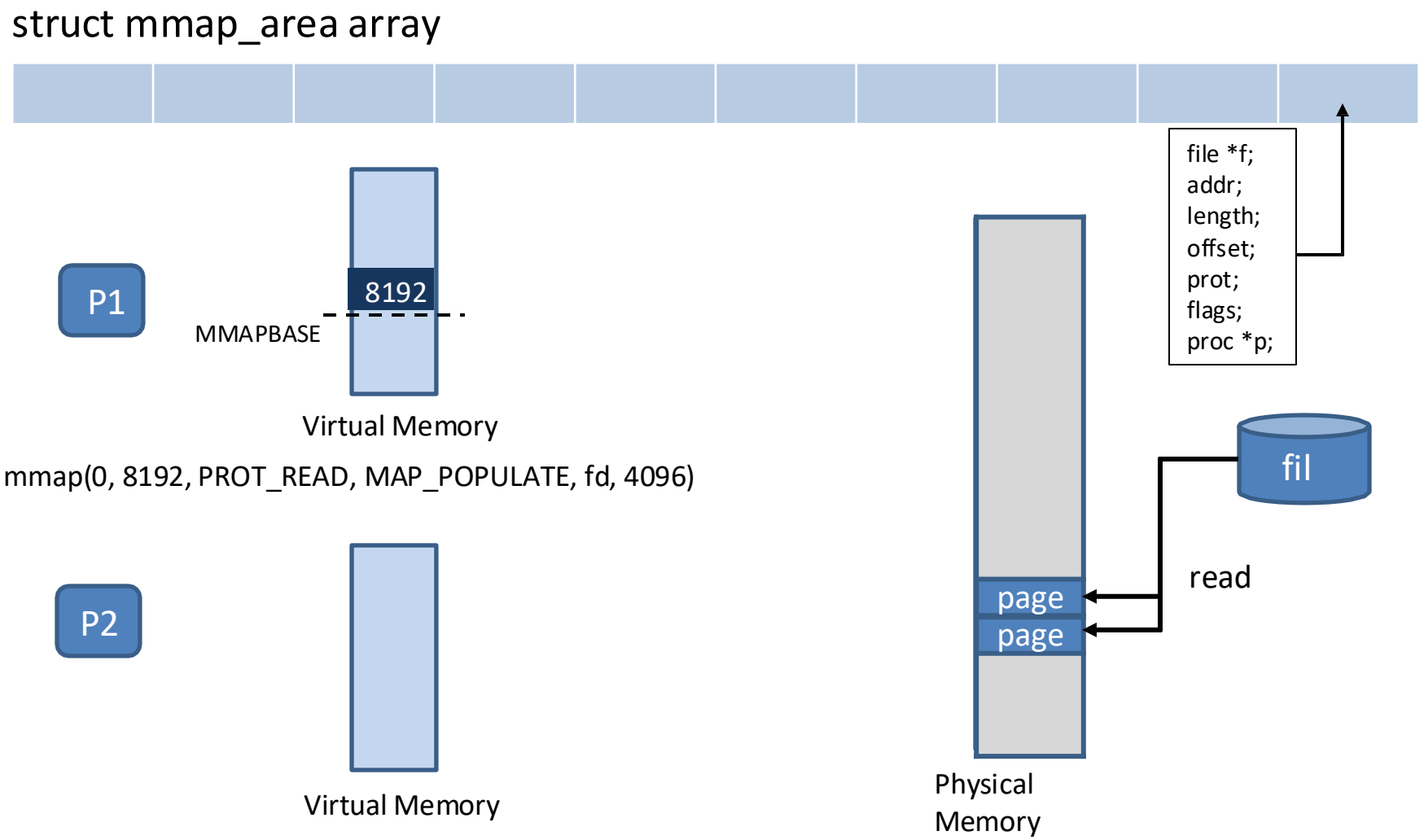
mmap_area Array Example

In the case of populate…



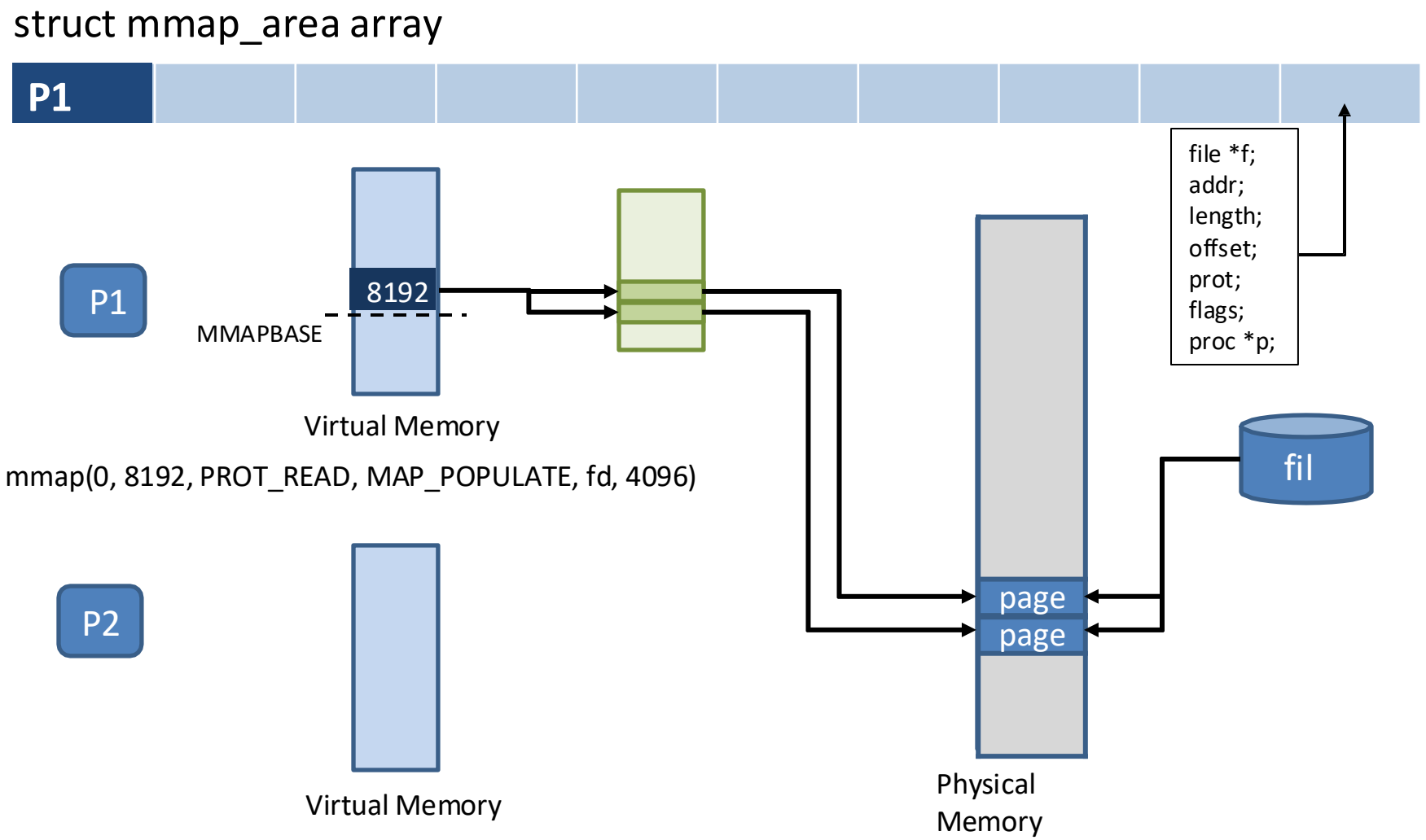
mmap_area Array Example

In the case of populate…



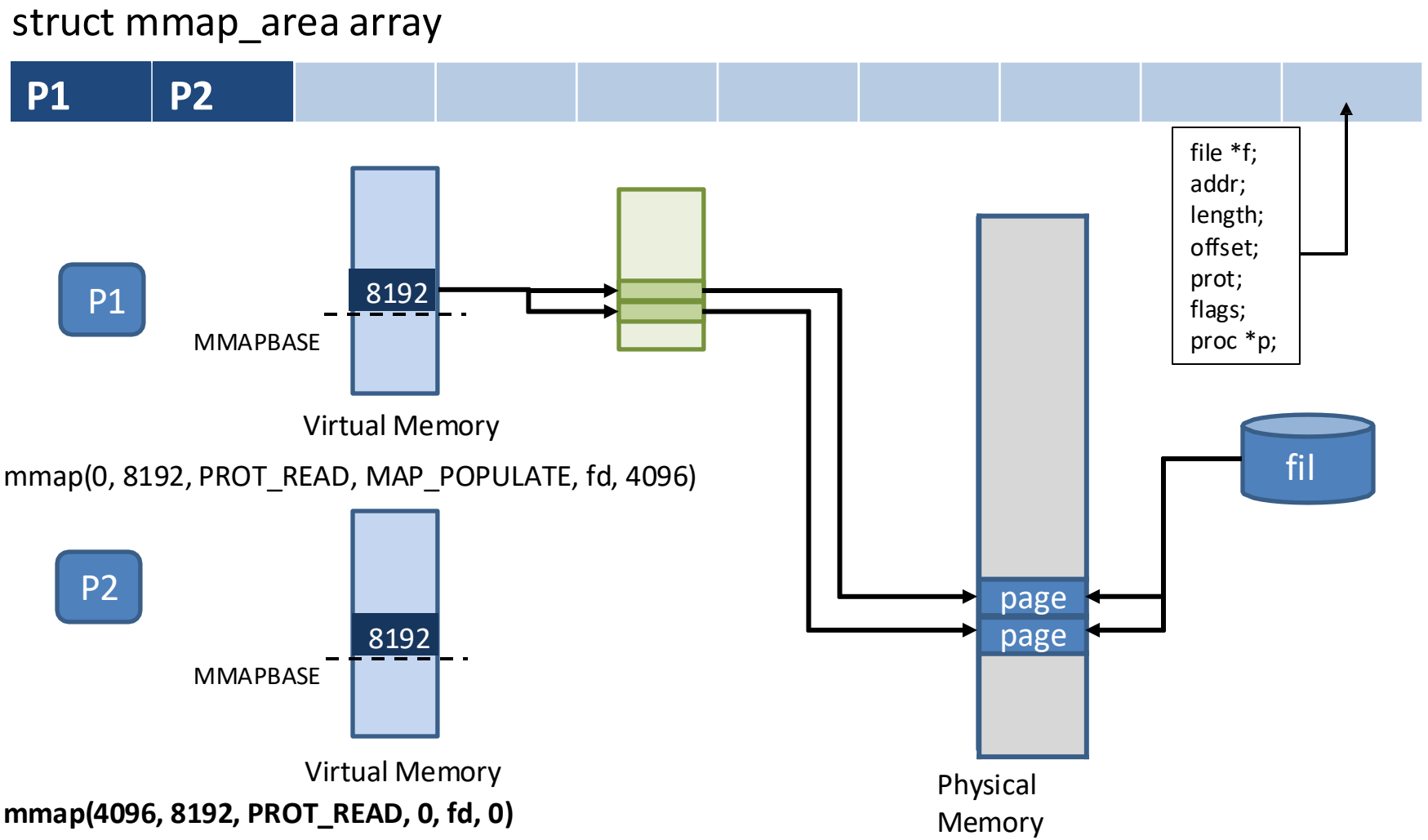
mmap_area Array Example

In the case of populate…



mmap_area Array Example

In the case of populate…



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 and return 1
- **Failed**: The process is terminated and return -1

1. When an access occurs (read/write), catch according page fault (interrupt 13 or 15) in ***traps.c***
2. In page fault handler, determine fault address by reading stval register(using r_stval()) & access was read or write by reading scause register (using r_scause())

   read: r_scause() == 13 / write: r_scause() == 15
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 0s
   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 set in PTE value)

3. munmap() system call on xv6

int munmap(uint64 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 1s 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 page is freed (put page into free list), freemem should be increase
2. When kernel page is allocated (take 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 should include below headers
- 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
    - Page Fault should be handled

```
#include "../kernel/types.h"
#include "../kernel/stat.h"
#include "user.h"
#include "../kernel/fcntl.h"
#include "../kernel/memlayout.h"
#include "../kernel/param.h"
#include "../kernel/spinlock.h"
#include "../kernel/sleeplock.h"
#include "../kernel/fs.h"
#include "../kernel/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. It will not be tested
- mmap()/munmap() page align
  - Address argument must be page aligned, if not, return 0

FAQ

- 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

```
ch = test3[0];
printf( "- fd data: ");
write(1, &ch, 1);
ch = test3[1];
write(1, &ch, 1);
ch = test3[2];
write(1, &ch, 1);
printf("\n");
```

- 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 Ye Server
  - $ make clean
  - $ ~swe3004/bin/submit pa3 xv6-riscv
  - 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: 11/11(Tue.), 23:59:59 PM
  - **-25%** per day for delayed submission

Questions

- If you have questions, please ask on i-campus
  - Please use the discussion board
  - Discussion board preferred over messages

- You can also visit Corporate Collaboration Center #85533
  - Please iCampus message TA before visiting

- Reading xv6 commentary will help you a lot

  https://pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.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 uvmcopy()
- 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 r_scause() and r_stval()