PROJECT

SUNGKYUNKWAN UNIVERSITY

SSE3044 Introduction to Operating Systems
Prof. Jinkyu Jeong

Project 3. Virtual Memory

TA)

송원석(wonsuk.song@csi.skku.edu)

진승우(seungwoo.jin@csi.skku.edu)

Project Plan

- Total 6 projects
  0. Booting xv6 operating system
  1. System call
  2. CPU scheduling
  3. Virtual memory
     - Memory mapped file
     - Demand paging
  4. Page replacement
  5. File systems

Project Objective

- Implement **mmap**() and **munmap**() system calls in xv6
  - File mapping and anonymous mapping
- Implement pre-paging and demand paging

- Codes you need to create or modify in xv6
  - **mmap**() sysccall
  - **munmap**() syscall
  - Page fault handler
  - Some extras

Memory Mapped File

- Map a portion of a file into a virtual address space
- Access file data using virtual memory access (e.g., load and store instructions) instead of using explicit file I/O operations (e.g., open(), read(), write() and close())

- **mmap**() supports two mappings
  - File-backed memory mapping → memory mapped file
  - Anonymous memory mapping

New System Calls

- The **mmap**()/**munmap**() interfaces you need to implement in xv6
  - void ***mmap**(int ***fd***, int ***offset***, int ***length***, int ***flags***)
  - int **munmap**(void* ***addr***, int ***length***)

- There is a slight difference from **mmap**() and **munmap**() used in Linux

mmap() System Call (1/2)

- void *__mmap__(int ***fd***, int ***offset***, int ***length***, int ***flags***)
  - Returns the start address of the new mapping
  - Returns MAP_FAILED if **mmap**() fails
    - #define MAP_FAILED ((void *) -1)
  - if ***fd*** is -1, the mapping is not file-backed, hence the memory within the mapping should be initialized with zero
  - if ***fd*** is specified, the mapping is file-backed and ***fd*** should be a valid open file descriptor.
    - ***offset*** is valid only when ***fd*** is specified. Hence, if ***fd*** is -1, ***offset*** is ignored.
  - ***flags*** can be any combination of the following flags
    - The supported flags are specified in the next page
  - ***lengths*** is the size of the new mapping and should be multiple of 4KB
  - From proc->sz, Investigate empty space and map it to the appropriate empty space

mmap() System Call (2/2)

- Flags to be supported
  - #define MAP_PROT_READ 0x00000001
    - Read permitted in the mapping
  - #define MAP_PROT_WRITE 0x00000002
    - Write permitted in the mapping
  - #define MAP_POPULATE 0x00000010
    - Without this flag, demand paging should be supported
    - With this flag, all page table mappings should be made before mmap() returns.
    - This will help to reduce blocking on page faults later

munmap() System Call

- int **munmap**(void* ***addr***, int ***length***)
  - Unmap the memory from ***addr*** to ***addr*** + ***length***
  - Returns 0 on success and -1 on failure
  - ***addr*** should be multiple of 4KB
  - ***length*** should be multiple of 4KB

  - If the specified mapping is file-backed, write back the modified contents within the mapping to file

  - If physical page is allocated & page table is constructed, should free physical page
    - You don't need to free the page table, just initialized pte value to 0

  - If there are no mappings in the specified address range, then **munmap**() has no effect

Pre-Paging

- **mmap**() with MAP_POPULATE
  1. Allocate pages
  2. Initialize each page
     1. If mapping is file-backed, read file data
     2. Otherwise, reset page with zero
  3. Setup PTEs
     1. Allocating a page table page may be also necessary
     2. Set proper protection bits and PFN bits
  4. Return the starting virtual address of mapping area

Demand Paging (1/2)

- **mmap**() without MAP_POPULATE
  1. Just record its mapping area
  2. Return the starting virtual address of mapping area
  3. If access the corresponding mapping area, do something to make page fault handler handles page faults properly
     - This operation may also be necessary in pre-paging
     - Specified in the next page

Demand Paging (2/2)

- Page fault handler
    1. Check whether the faulted virtual address is within a valid mapping
        - If the address is not within any valid mappings, follow the original path that kills the process.
    2. Allocate a page
    3. Initialize a page
        1. If mapping is file-backed, read file data
        2. Otherwise, reset page with zero
    4. Setup a PTE
        1. Allocating a page table page may be also necessary
        2. Set proper protection bits and PFN bits
    5. Return

Several Considerations and Assumptions

- fork()
  - When a process forks, mappings made by mmap() are not cloned
    - This is different from the original fork/mmap semantics, but this will simplify your tasks
- close()
  - Assume that a file descriptor is not closed until its mapping is unmapped
- exit()
  - When a process exits, all mappings should be unmapped before termination
  - Pages mapped in PTEs within all the mappings should also be freed
- File mapping is not shared by multiple processes
  - Only a single process can mmap a file

Useful xv6 Functions

- File read / write
  - int fileread(struct file ***f**, char ***addr**, int ***n***)
  - int filewrite(struct file ***f**, char ***addr**, int ***n***)

- Page allocation & page free
  - char* kalloc(void)
  - void kfree(char ***v**)

- Mapping into page table & find address of pte
  - int mappages(pde_t ***pgdir**, void ***va**, uint ***size***, uint ***pa***, int ***perm***)
  - pte_t* walkpgdir(pde_t ***pgdir**, const void ***va**, int ***alloc***)

- And, reading xv6 commentary will help you a lot
  - http://csl.skku.edu/uploads/SSE3044S20/book-rev11.pdf

Files to be Modified

- trap.c
  - Catch page fault (14, T_PGFLT) and calls page fault handler

- sysproc.c, proc.c ...
  - System call declaration and implementation or modification
  - Ex) **mmap**(), **munmap**(), exit()…

- file.c (optional)
  - If you have any ideas in file read/write

- You can also modify other files freely!

Hints (1/2)

- Where the page fault handler should be placed
  - A page fault handler should be called inside **trap.c**
  - Trap number of page fault is **T_PGFLT**
- A fault can occur in user virtual memory or kernel virtual memory
  - Your page fault handler should only handle faults within user virtual memory
  - **KERNBASE** is the boundary between user and kernel virtual memories
- How to know faulted virtual address
  - CR2 register has the faulted virtual address when page fault occurs
- How to know access type (read or write)
  - Bit 1 in **tf->err** is set if access is write
- How to flush TLB
  - Reload the CR3 register (ex. **lcr3**(**V2P**(**pgdir**))
  - Think about when TLB flush is necessary. You may need to refer to Intel architecture software developer's manual (Vol 3A, 4.10.4.2)

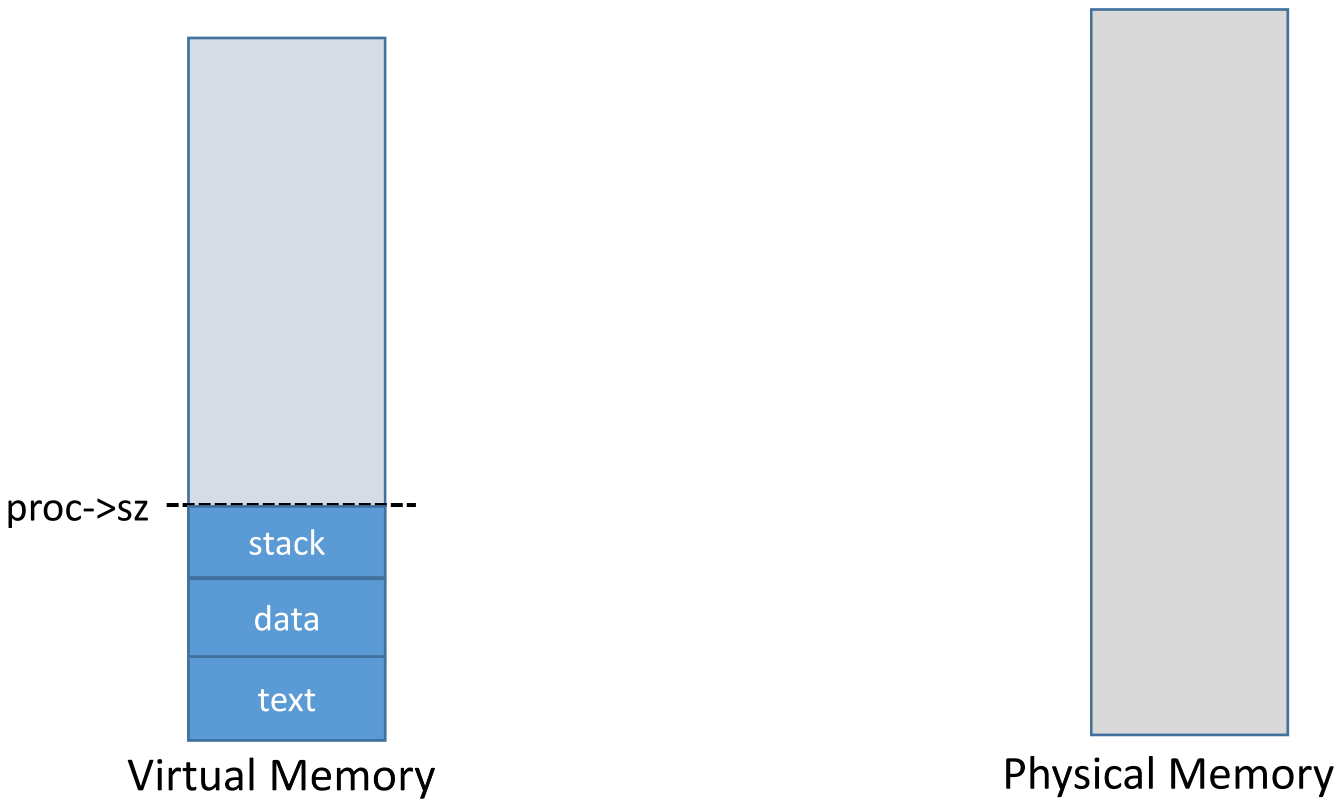
Hints (2/2)

- Page fault can occur because PTE is invalid or PTE has no proper protection bits
  - Ex) write page fault can occur if PTE has only read permission.

- It would be convenient to have a structure variable that manages the mapped region
  - Ex) construct variables using **mmap**() parameters

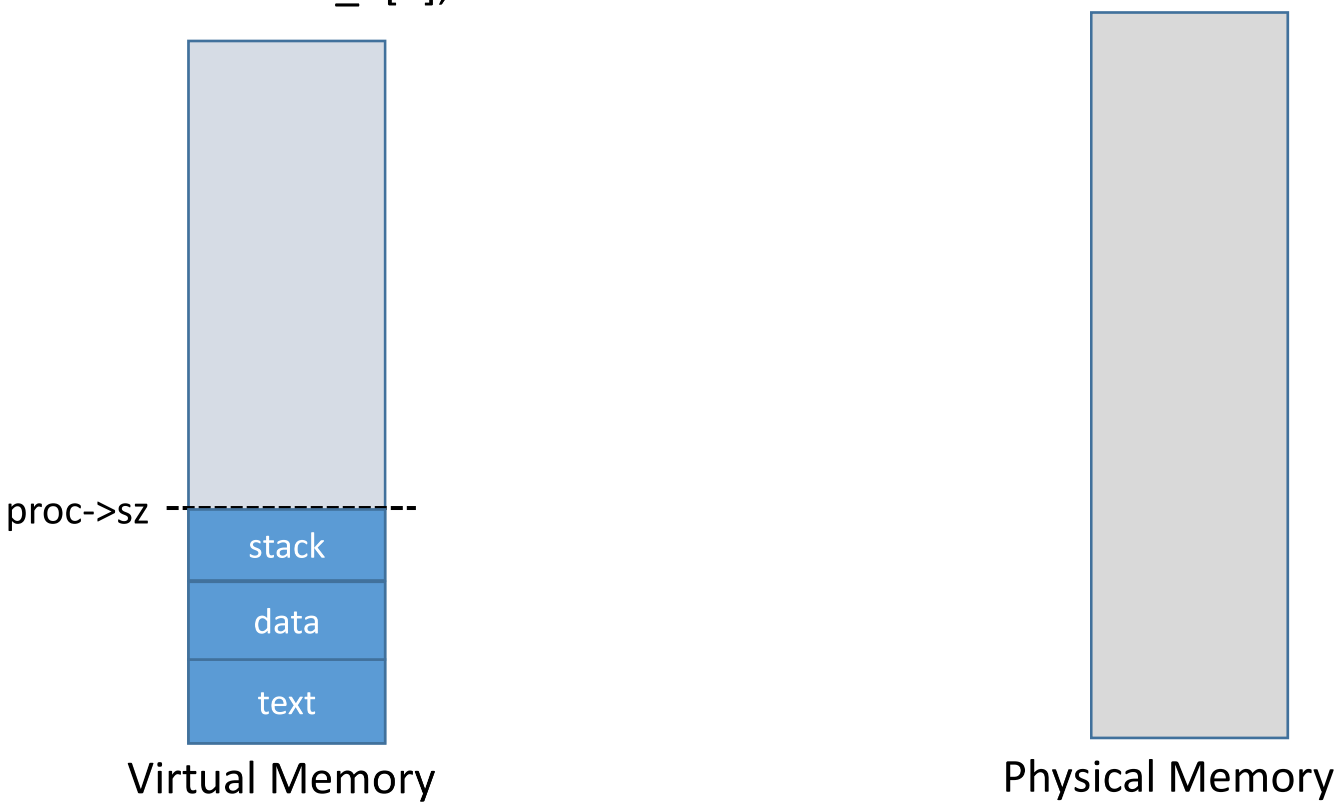
- xv6 has no dynamic memory allocator like malloc() in C
  - If you need dynamic memory allocation, use **kalloc**() which returns 4KB memory
  - Before using **kalloc**(), please consider whether your required memory can be allocated statically.
  - Memory leak should NOT occur. Hence, you need to free memory whenever the memory is no longer necessary.

mmap(), munmap() Example (1/2)



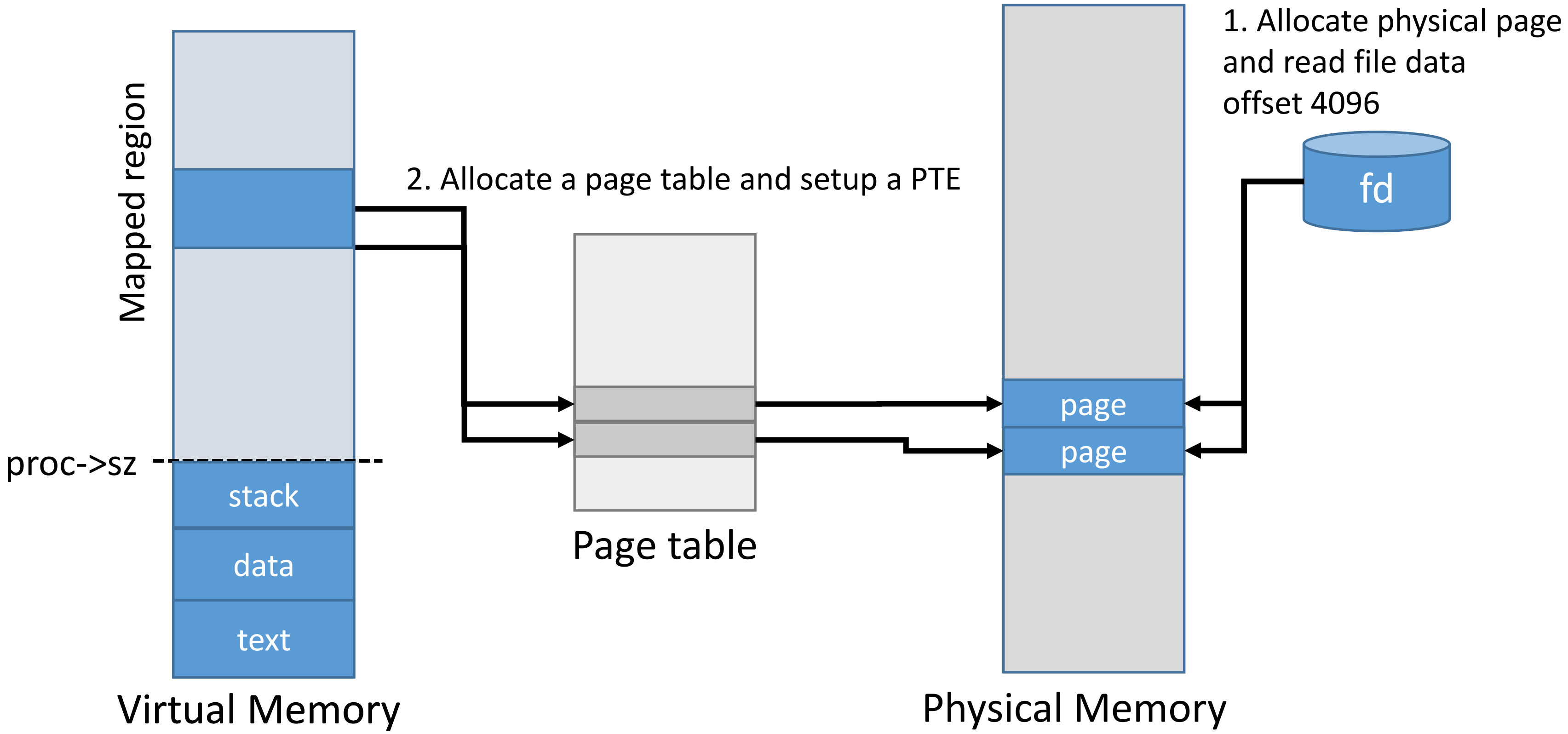
mmap(), munmap() Example (1/2)

1. char * addr_1 = **mmap**(fd, 4096, 8192, MAP_PROT_READ | MAP_POPULATE);
2. char a = addr_1[0];
3. **munmap**(addr_1, 4096);
4. char b = addr_1[1];



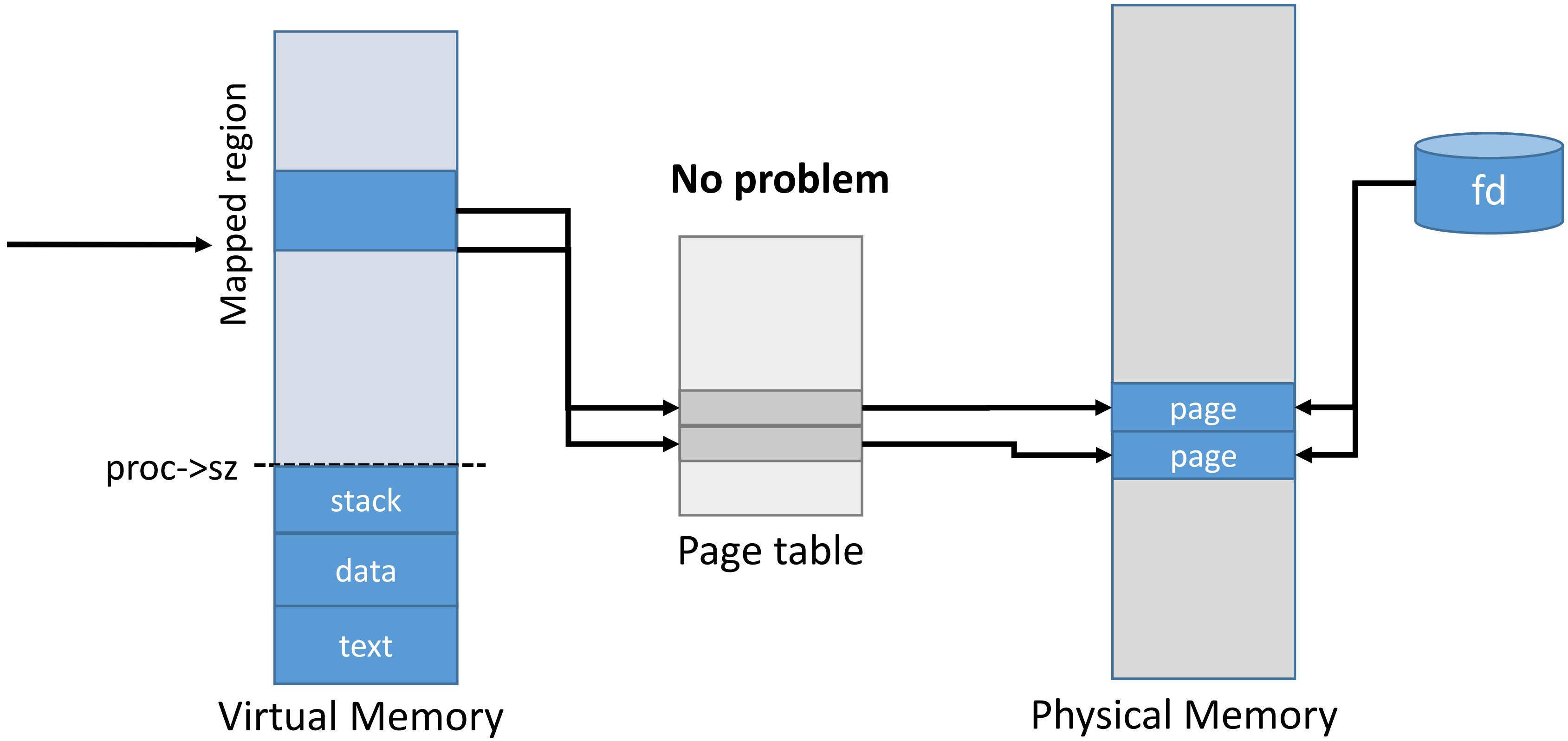
mmap(), munmap() Example (1/2)

1. char * addr_1 = **mmap**(fd, 4096, 8192, MAP_PROT_READ | MAP_POPULATE);
2. char a = addr_1[0];
3. **munmap**(addr_1, 4096);
4. char b = addr_1[1];



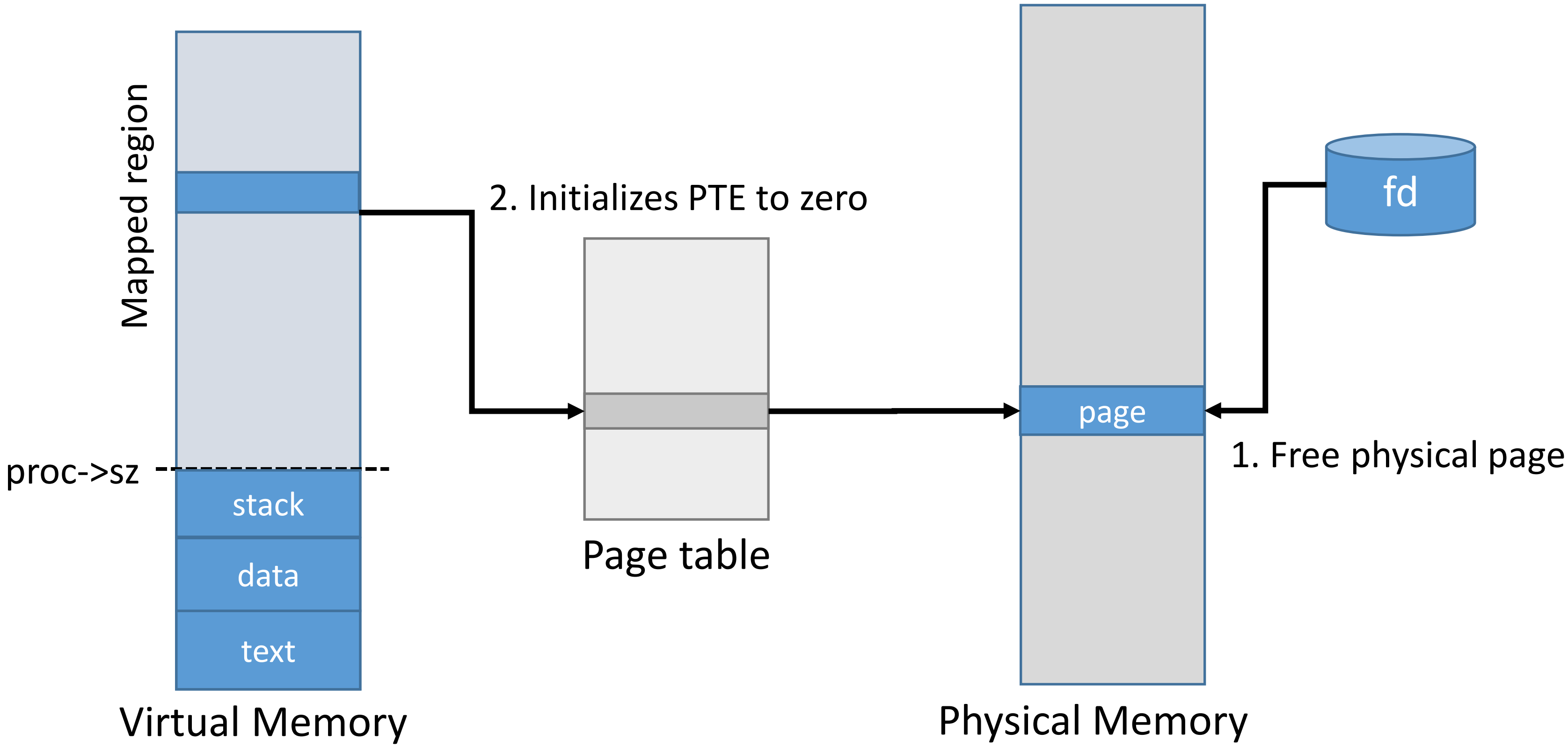
mmap(), munmap() Example (1/2)

1. char * addr_1 = **mmap**(fd, 4096, 8192, MAP_PROT_READ | MAP_POPULATE);
2. char a = addr_1[0];
3. **munmap**(addr_1, 4096);
4. char b = addr_1[1];



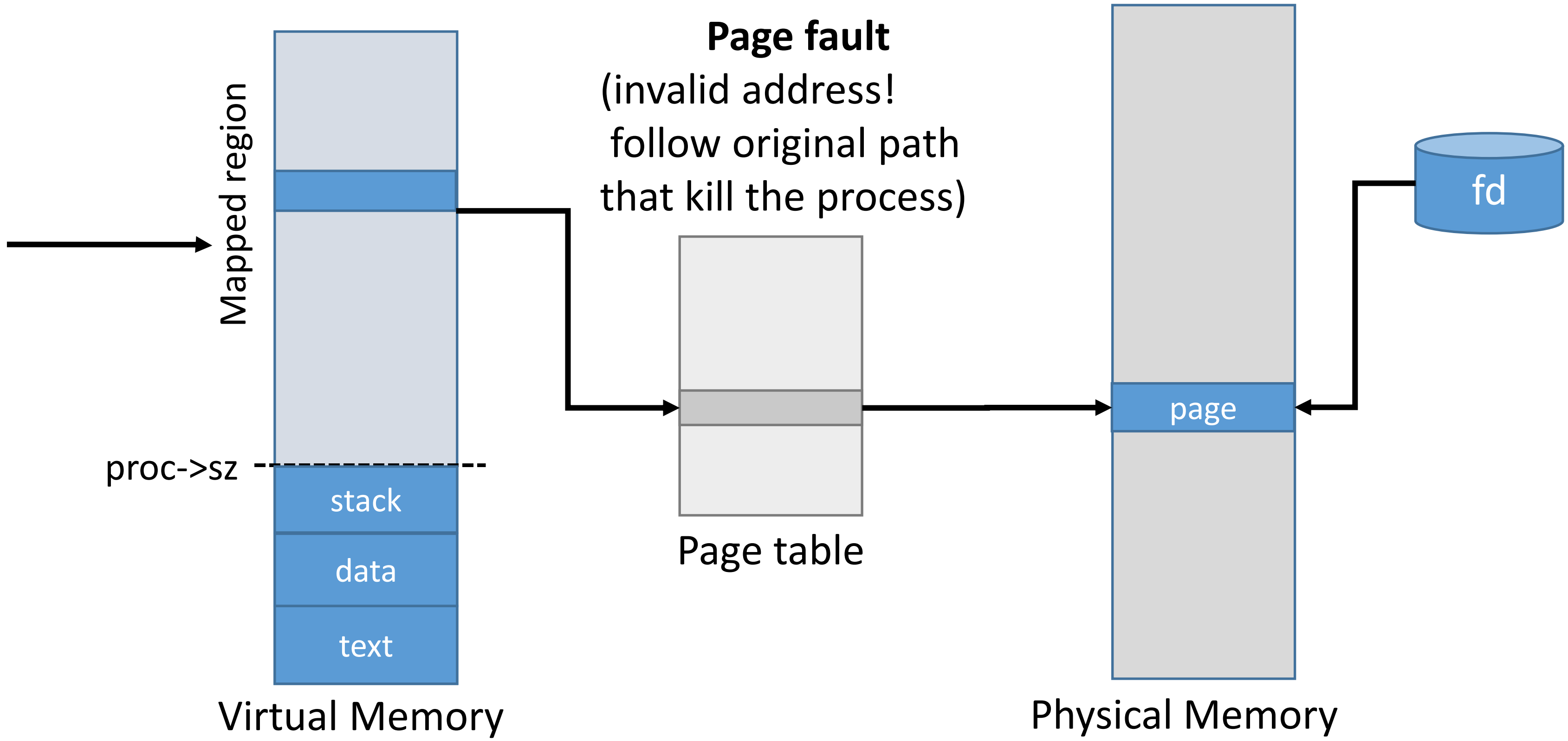
mmap(), munmap() Example (1/2)

1. char * addr_1 = **mmap**(fd, 4096, 8192, MAP_PROT_READ | MAP_POPULATE);
2. char a = addr_1[0];
3. **munmap**(addr_1, 4096);
4. char b = addr_1[1];

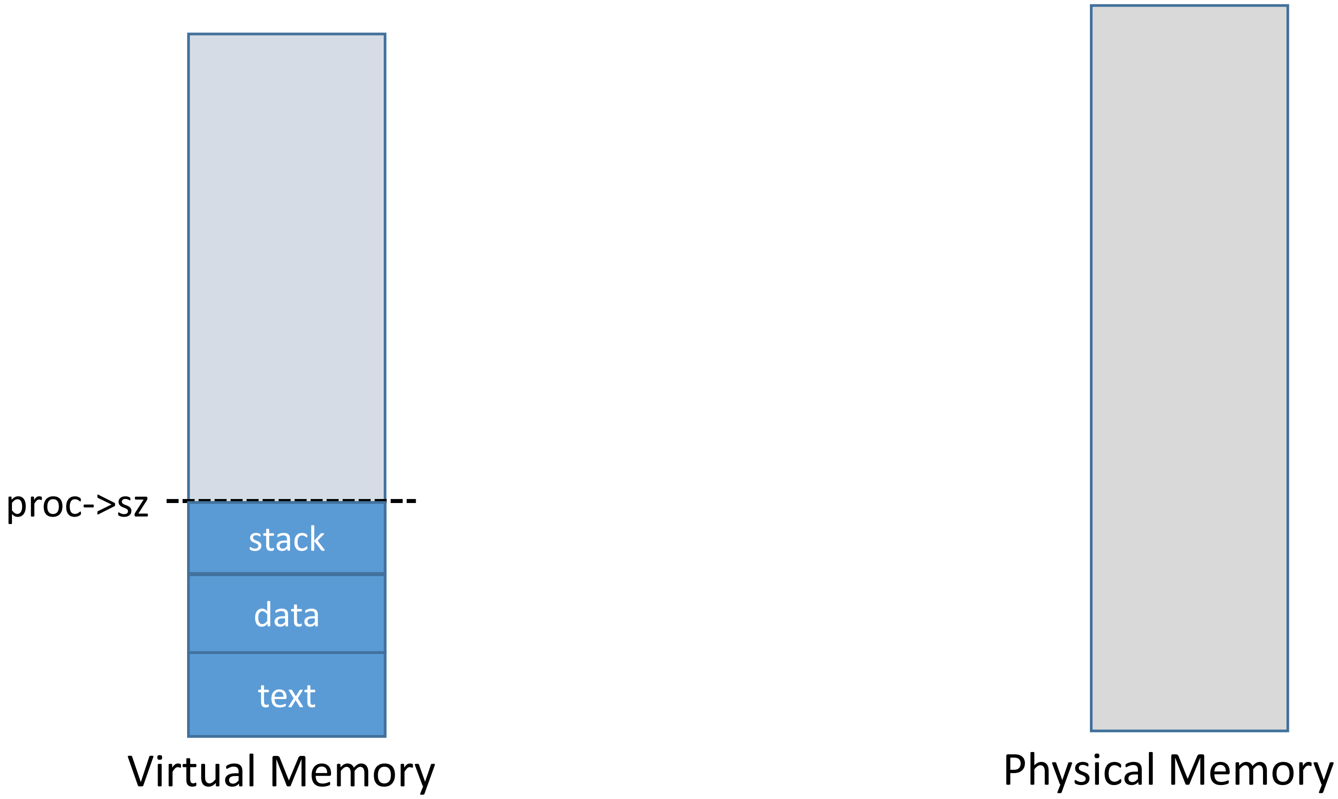


mmap(), munmap() Example (1/2)

1. char * addr_1 = **mmap**(fd, 4096, 8192, MAP_PROT_READ | MAP_POPULATE);
2. char a = addr_1[0];
3. **munmap**(addr_1, 4096);
4. char b = addr_1[1];

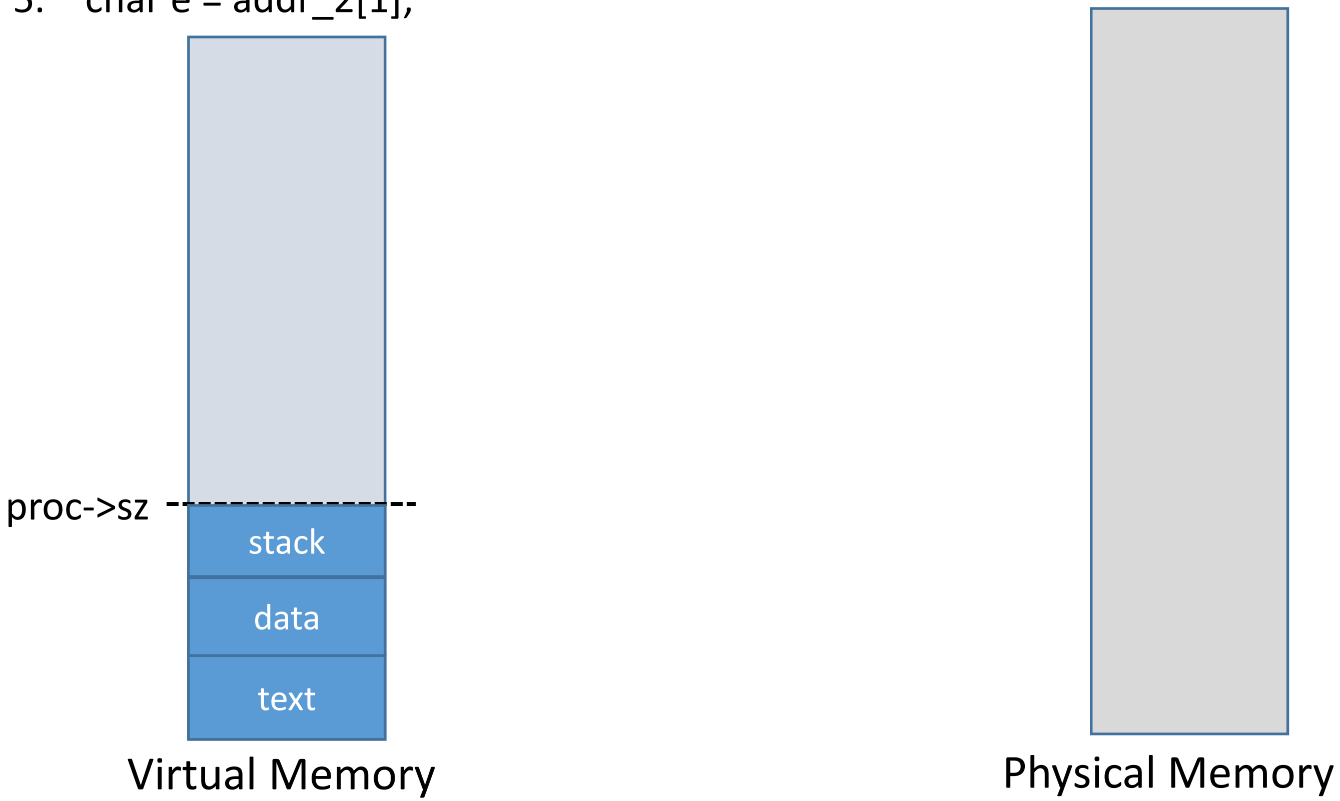


mmap(), munmap() Example (2/2)



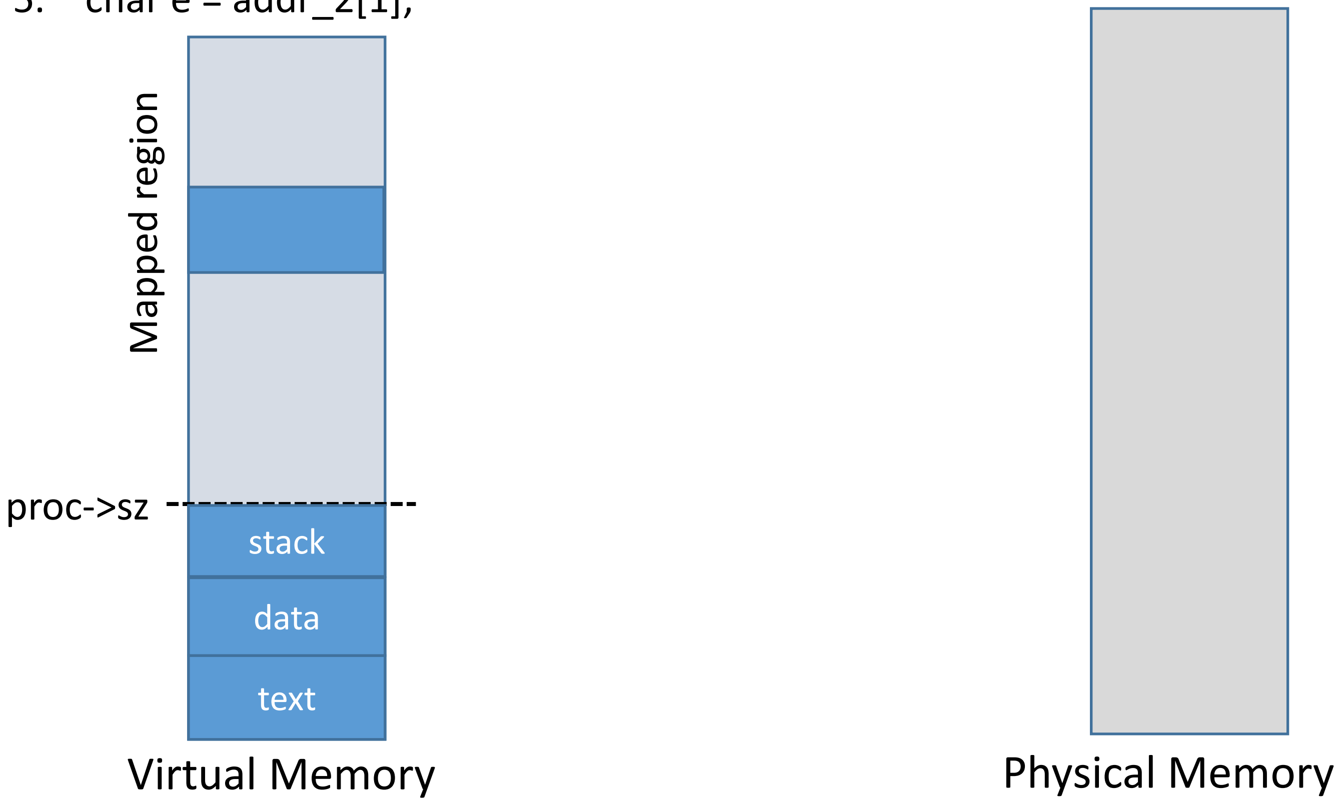
mmap(), munmap() Example (2/2)

1. char * addr_2 = **mmap**(-1, 0, 8192, MAP_PROT_READ);
2. char c = addr_2[0];
3. char d = addr_2[4096];
4. **munmap**(addr_2, 8192);
5. char e = addr_2[1];



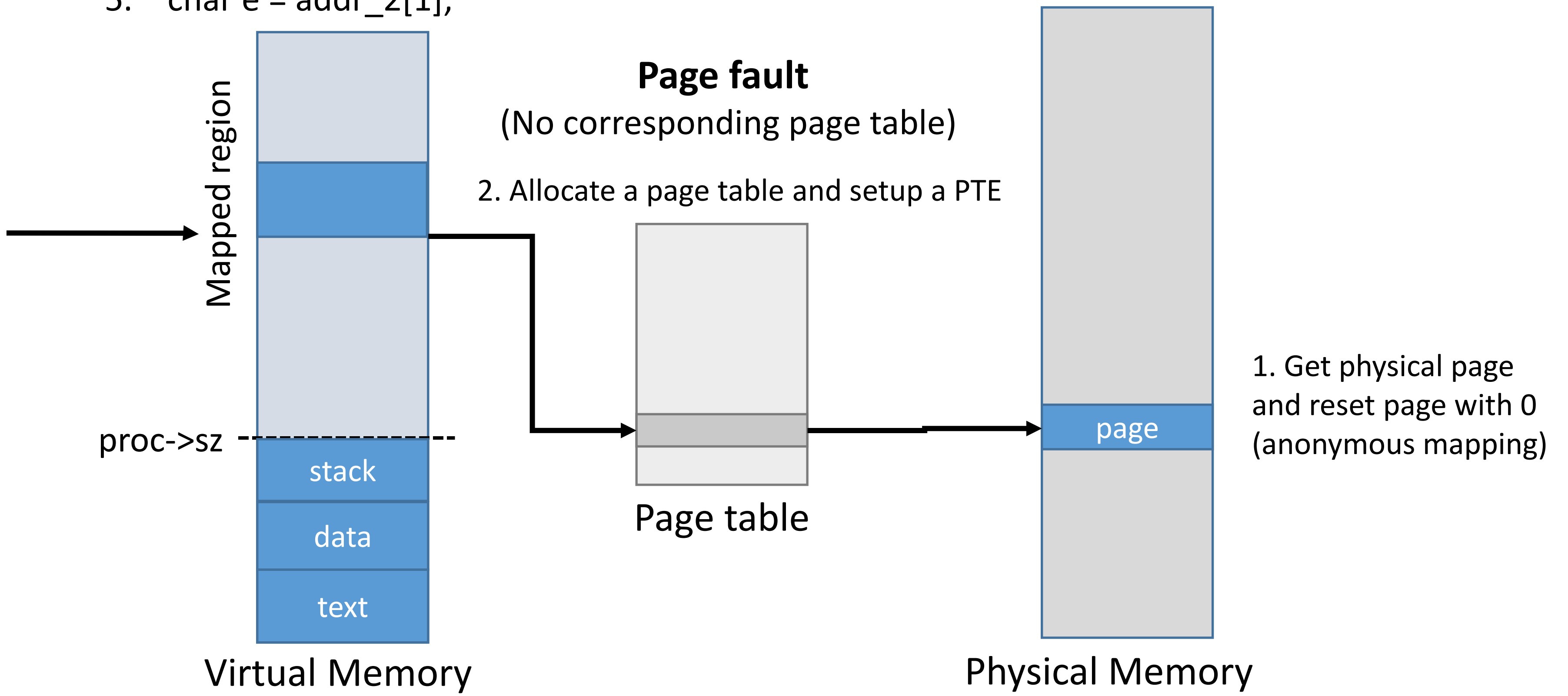
mmap(), munmap() Example (2/2)

1. char * addr_2 = **mmap**(-1, 0, 8192, MAP_PROT_READ);
2. char c = addr_2[0];
3. char d = addr_2[4096];
4. **munmap**(addr_2, 8192);
5. char e = addr_2[1];



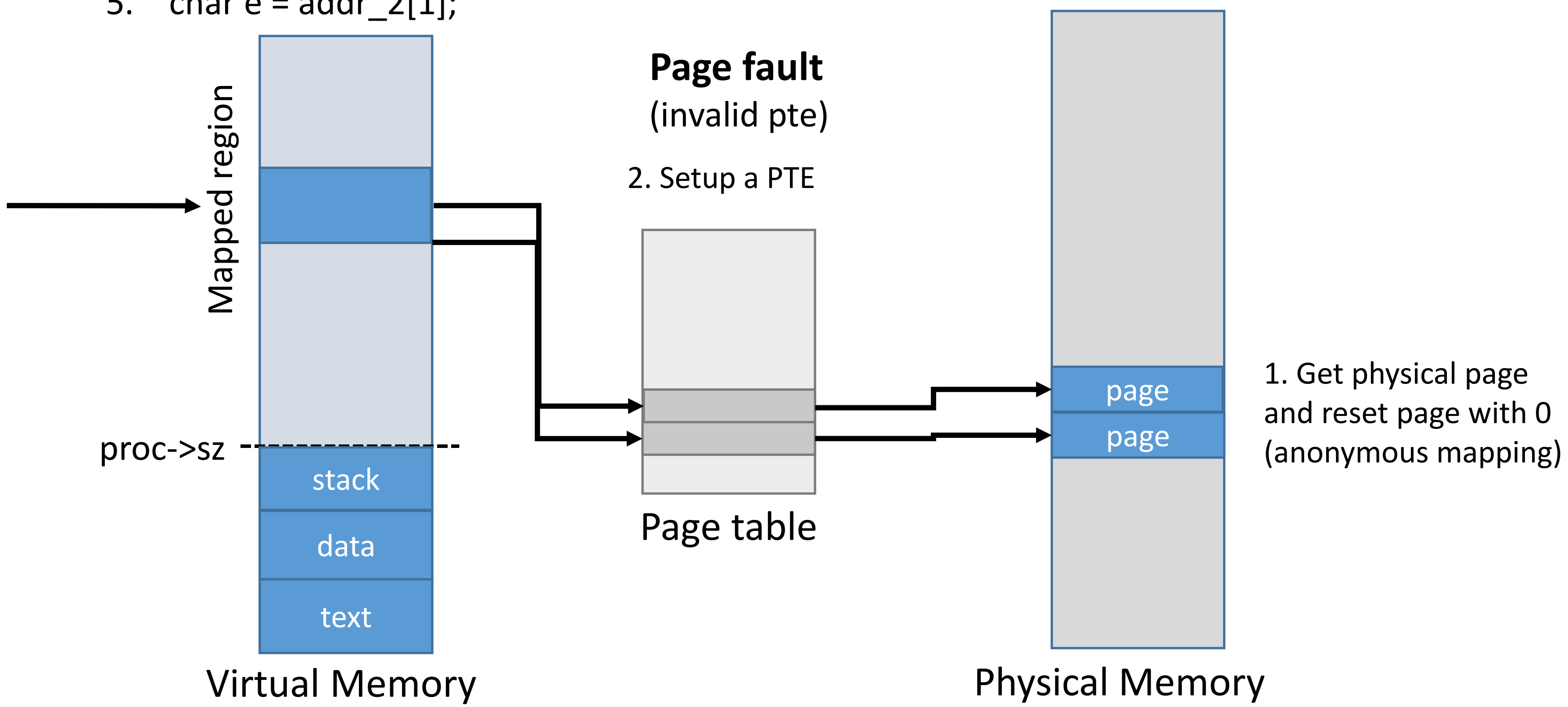
mmap(), munmap() Example (2/2)

1. char * addr_2 = **mmap**(-1, 0, 8192, MAP_PROT_READ);
2. char c = addr_2[0];
3. char d = addr_2[4096];
4. **munmap**(addr_2, 8192);
5. char e = addr_2[1];



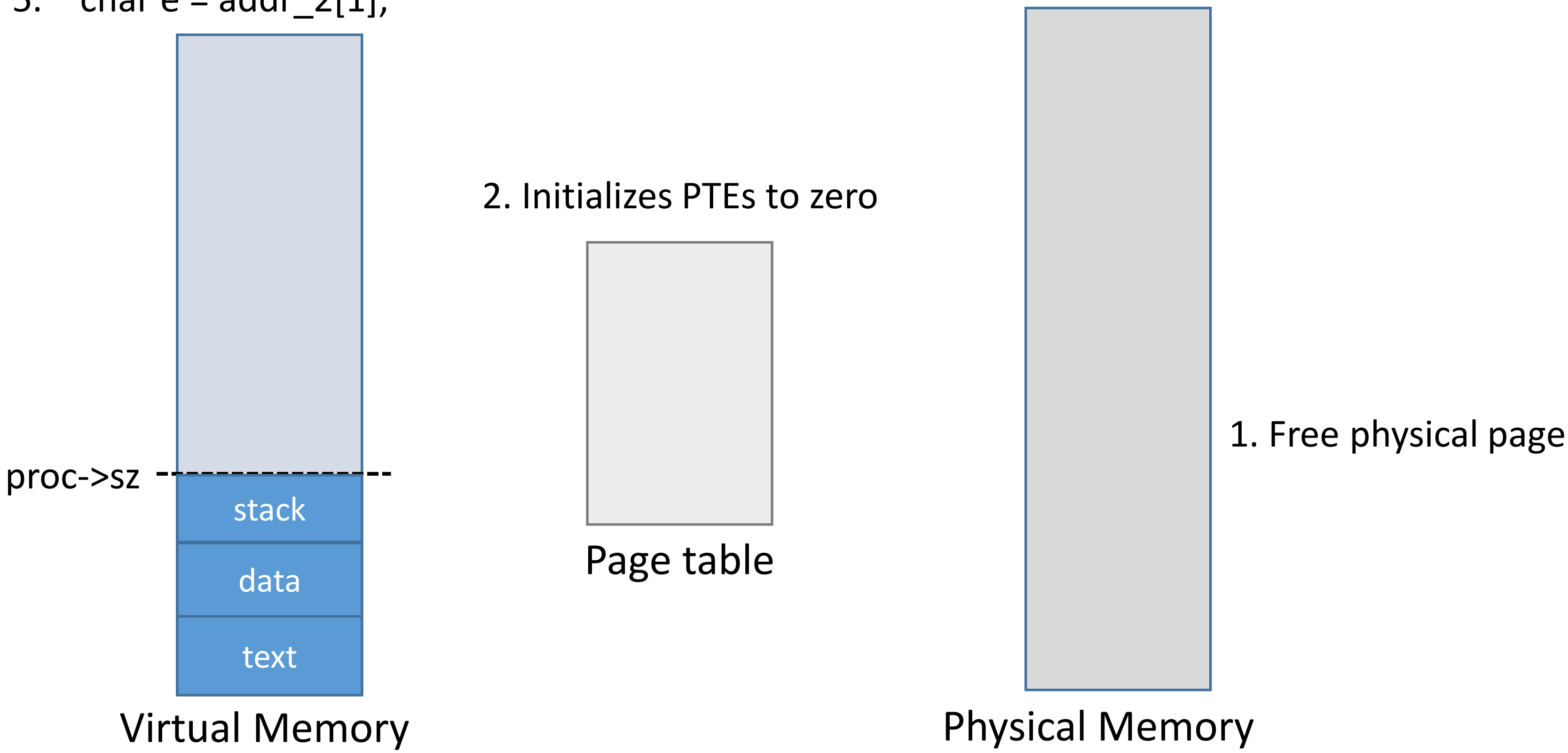
mmap(), munmap() Example (2/2)

1. char * addr_2 = **mmap**(-1, 0, 8192, MAP_PROT_READ);
2. char c = addr_2[0];
3. char d = addr_2[4096];
4. **munmap**(addr_2, 8192);
5. char e = addr_2[1];



mmap(), munmap() Example (2/2)

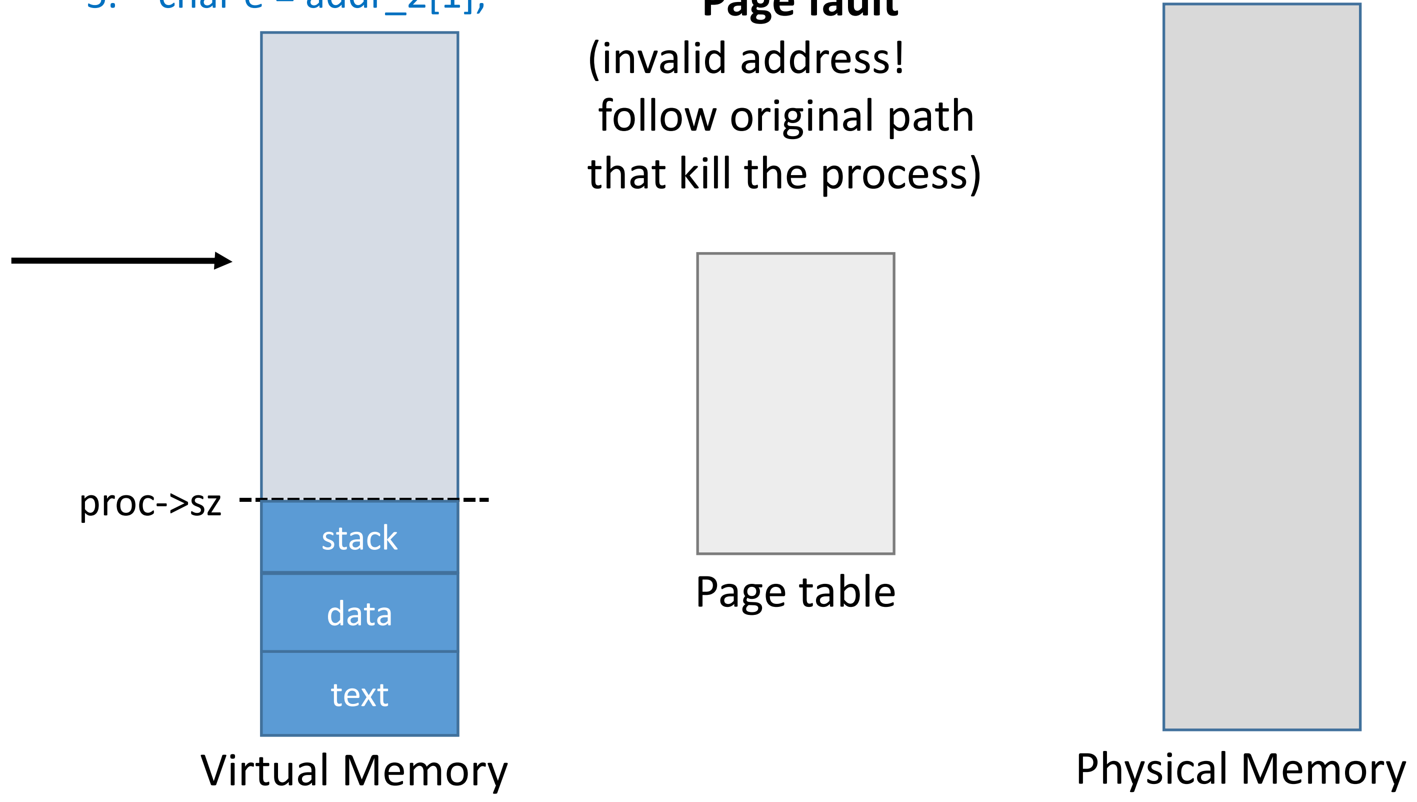
1. char * addr_2 = **mmap**(-1, 0, 8192, MAP_PROT_READ);
2. char c = addr_2[0];
3. char d = addr_2[4096];
4. **munmap**(addr_2, 8192);
5. char e = addr_2[1];



mmap(), munmap() Example (2/2)

1. char * addr_2 = **mmap**(-1, 0, 8192, MAP_PROT_READ);
2. char c = addr_2[0];
3. char d = addr_2[4096];
4. **munmap**(addr_2, 8192);
5. char e = addr_2[1];

Page fault
(invalid address!
follow original path
that kill the process)



How to Test?

- To check that **mmap**() and **munmap**() are implemented properly,
  - Expected output (mmap_test.c)

```
mmap test
-x86 or non-ELF machines (like OS X, even on x86), you
will need to install a cross-compiler gcc suite capable of producing
x86 ELF binaries (see https://pdos.csail.mit.edu/6.828/).
Then run "make TOOLPREFIX=i386-jos-elf-". Now install the QEMU PC
simulator and run "make qemu".

============file mmap end==========



sY
============anonymous mmap end==========
```

  - Simple testcase using **mmap**() and **munmap**()
  - File-backed mapping and anonymous mapping

- Please test the many cases by changing the code a lot
  - But you should insert exit() function at the end
- Check that address is unable to access after **munmap**()

Submission

- Begin with clean xv6 code
- Compress your source code and upload on i-Campus
- How to compress your project
  - If you insert the user program, Modify the 'EXTRA' in Makefile
  - make dist
  - make tar
  - Then, tar.gz file will be generated automatically
  - Rename to studentID-project3.tar.gz

- Submit a report together, the file format of the report is limited to pdf
  - There is no limit to the format of the contents
  - But, include your description of your code and execution screen

Submission

- File format
  - StudentID-project3.tar.gz
  - StudentID-project3.pdf

- PLEASE DO NOT COPY
  - YOU WILL GET F GRADE IF YOU COPIED

- Due date: 5/12(Tue.), 23:59:59 PM
  - -25% per day for delayed submission

Questions

- If you have questions, please ask in Piazza

- You can also visit Semiconductor Building #400509
  - Please e-mail TA before visiting

- Reading xv6 commentary will help you a lot
  - http://csl.skku.edu/uploads/SSE3044S20/book-rev11.pdf