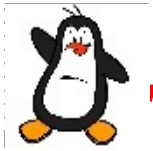
Lecture # 21
Memory Mappings

Course: SYSTEM PROGRAMMING

Instructor: Arif Butt

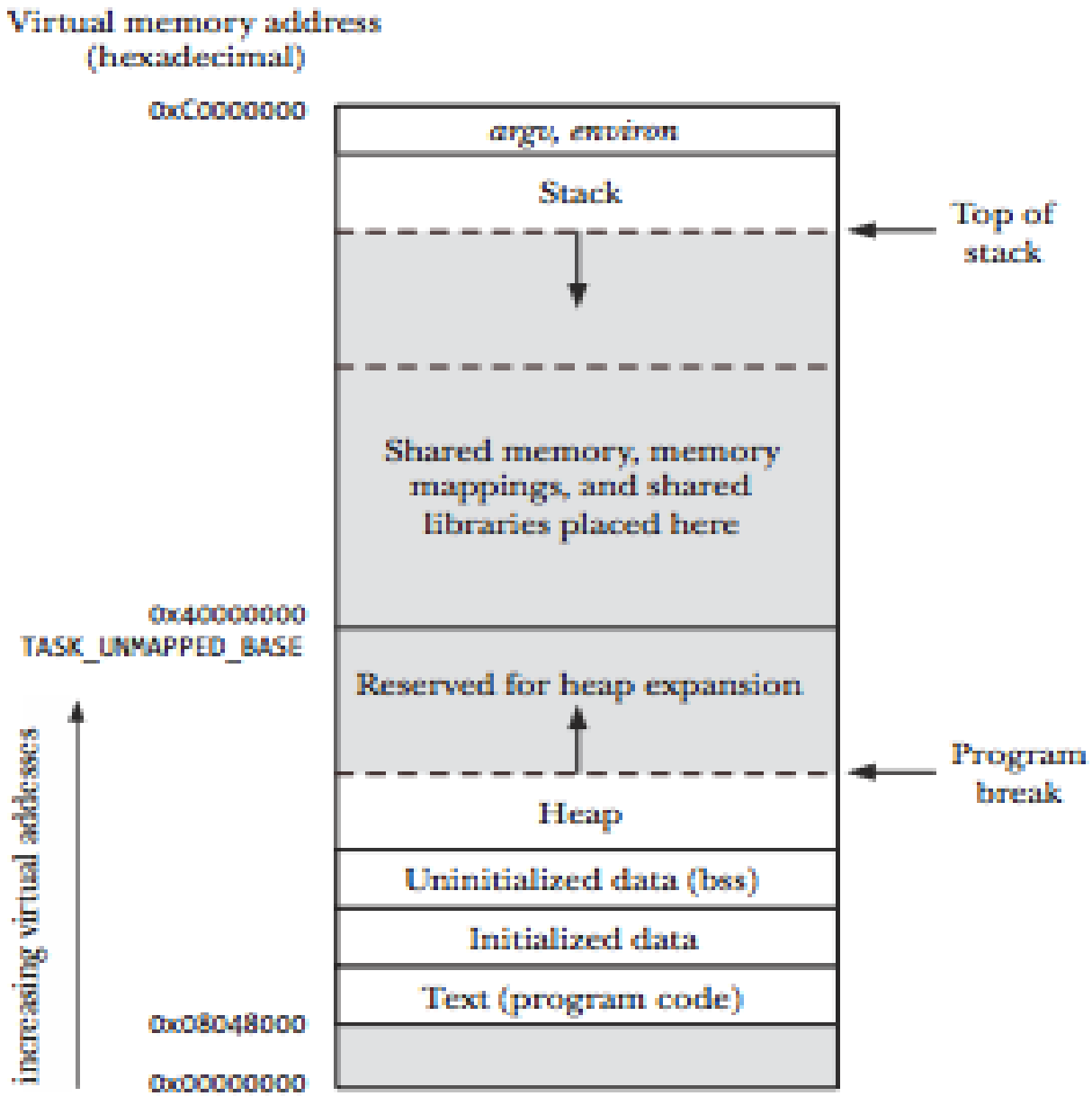
Punjab University College of Information Technology (PUCIT)
University of the Punjab

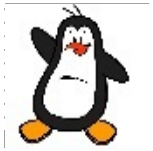


Today's Agenda

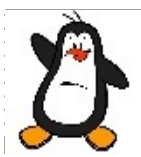
- Introduction to Memory Mappings
- Types of Memory Mappings
  - Private File Mapping
  - Shared File Mapping
  - Private Anonymous Mapping
  - Shared Anonymous Mapping
- Memory Mapped I/O

Location of Memory Mappings in Virtual Memory

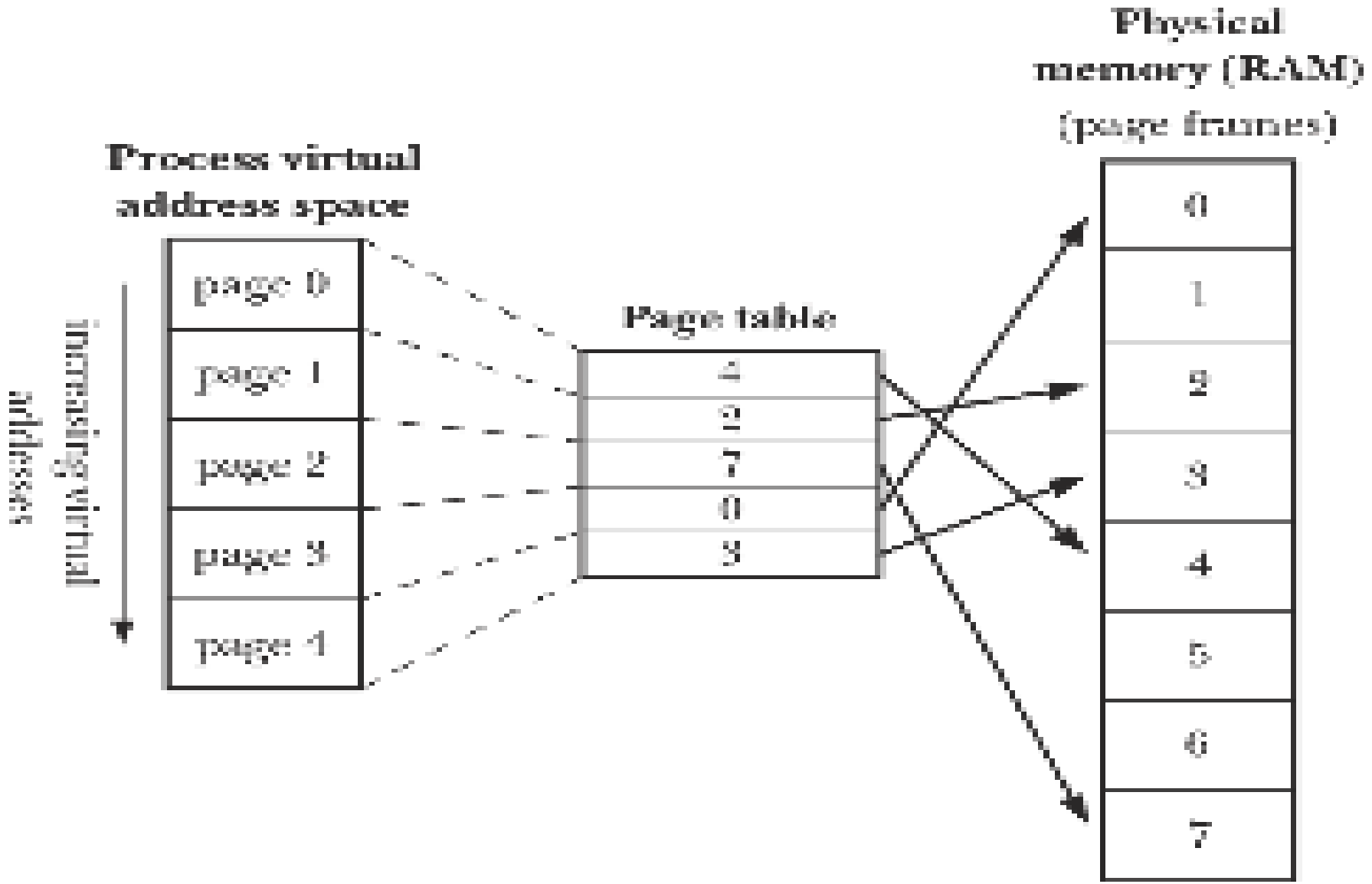




Memory Mappings

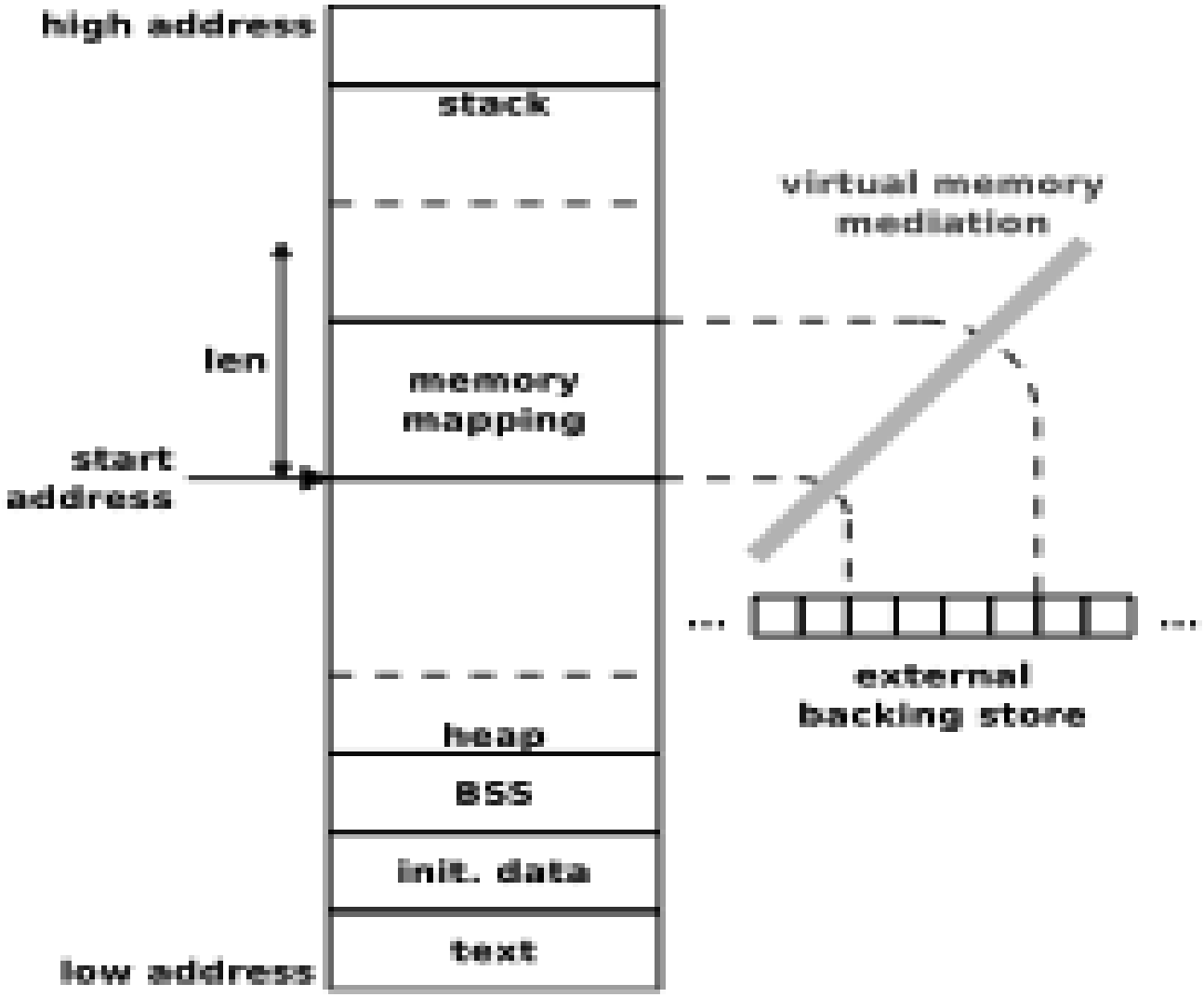


Virtual Memory Revisited



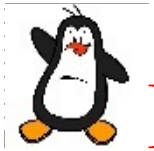
Memory Mapping

- A UNIX memory mapping is a virtual memory area that has an extra backing store layer, which points to an external page store.
- Processes can manipulate their memory mappings —request new mappings, resize or delete existing mappings, flush them to their backing store, etc.
- A memory mapping can be thought of as a dynamically allocated memory with peculiar read/write rules.



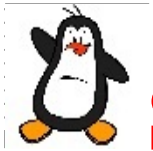
Memory Mapping Types

- Memory mappings can be of two different types, depending on the ultimate page backing store.
  - A **file mapping** maps a memory region to a region of a file
    - backing store = file
    - as long as the mapping is established, the content of the file can be read from or written to using direct memory access ("as if they were variables")
  - An **anonymous mappings** maps a memory region to a fresh "virtual" memory area filled with 0
    - backing store = zero-ed memory area



Having Mapped Pages in Common

- Thanks to virtual memory management, different processes can have mapped pages in common.
- More precisely, mapped pages in different processes can refer to physical memory pages that have the same backing store.
- This can happen in two ways:
  - through fork, as memory mappings are inherited by children
  - when multiple processes map the same region of a file



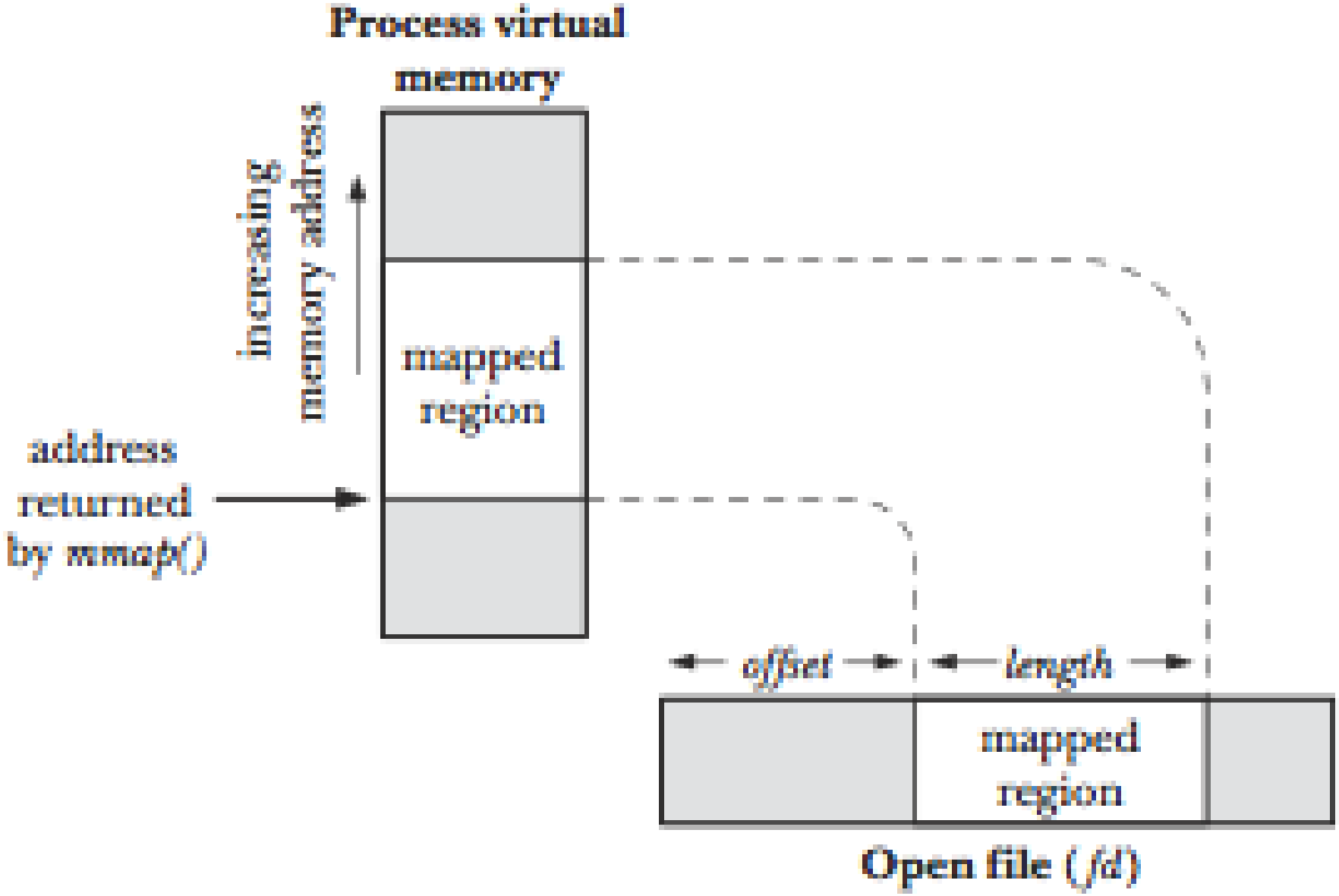
Shared vs Private Mapping

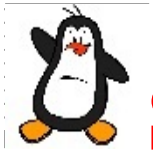
- With mapped pages in common, the involved processes might see changes performed by others to mapped pages in common, depending on whether the mapping is:
  - **Private mapping**. Modifications are not visible to other processes. Pages are initially the same, but modification are not shared, as it happens with copy-on-write memory after fork. Private mappings are also known as copy-on-write mappings.
  - **Shared mapping.** In this case modifications to mapped pages in common are visible to all involved processes, i.e. pages are not copied-on-write

Private File Mapping

- The contents of the mapping are initialized from a file region.
- Multiple processes mapping the same file initially share the same physical pages of memory
- Whenever a process tries to write, copy-on-write technique is employed, so that changes to the mapping by one process are invisible to other processes.
- **The main use of this type of mapping is initializing a process's text and initialized data segments from the corresponding parts of a binary executable file or a shared library file.**

Private File Mapping

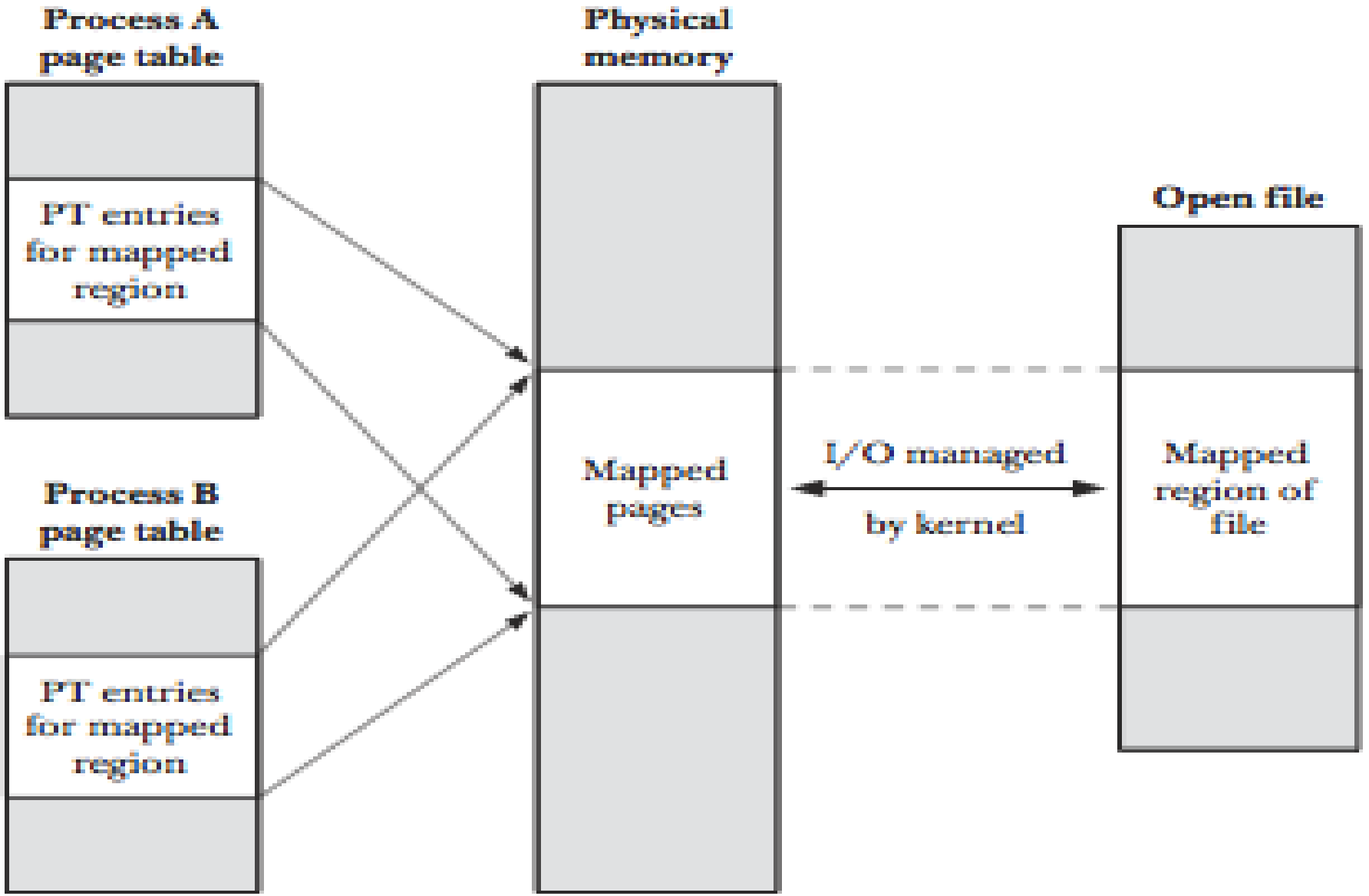


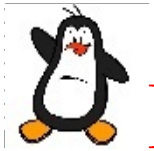


Shared File Mapping

- The contents of the mapping are initialized from a file region.
- Multiple processes mapping the same region of a file share the same physical pages of memory.
- Whenever a process tries to write, the modifications to the contents of the mapping are carried through to the file.
- **Main use of this type of mapping is:**
  - Memory-mapped I/O.
  - IPC in a manner similar to System V shared memory segments between related or unrelated processes

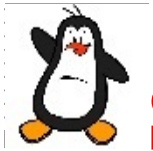
Shared File Mapping





Private Anonymous Mapping

- An anonymous mapping is one that doesn't have a corresponding file.
- Each call to `mmap()` to create a anonymous mapping yields a new mapping that is distinct from (i.e., does not share physical pages with) other anonymous mappings created by the same (or a different) process.
- If this Anonymous mapping is **private**, then although a child process inherits its parent's mappings, but the pages of the mapping are copy-on-write. Thus ensuring that, after the `fork()`, the parent and child don't see changes made to the mapping by the other process.
- The primary purpose of private anonymous mappings is to allocate new (zero-filled) memory for a process (e.g., `malloc()` employs `mmap()` for this purpose when allocating large blocks of memory).



Shared Anonymous Mapping

- An anonymous mapping is one that doesn't have a corresponding file.
- Each call to `mmap()` to create a anonymous mapping yields a new mapping that is distinct from (i.e., does not share physical pages with) other anonymous mappings created by the same (or a different) process.
- If this Anonymous mapping is **shared**, then although a child process inherits its parent's mappings, but the pages of the mapping are NOT copy-on-write. Thus ensuring that, after the `fork()`, the parent and child see changes made to the mapping by the other process.
- The primary purpose of shared anonymous mappings is to allow IPC in a manner similar to SystemV shared memory segments, but only between related processes.

mmap() System Call

```
void *mmap(void * addr , size_t len , int prot,
           int flags, int fd , off_t offset);
```

The mmap syscall is used to request the creation of memory mappings in the address space of the calling process. It returns the starting address of the mapping if OK, and MAP_FAILED on error.

- The **addr** argument indicates the virtual address at which the mapping is to be located. Preferably, we should specify addr as NULL, so that the kernel chooses a suitable address for the mapping that doesn't conflict with any existing mapping.
- The **len** argument specifies the size of the mapping in bytes. If len is not a multiple of virtual memory page size, kernel creates mappings rounded up to the next multiple of the page size. To map an entire file, we put len as size of the file.

mmap() System Call

```
void *mmap(void * addr , size_t len , int prot,
           int flags, int fd , off_t offset)
```

- The **prot** argument is a bit mask specifying the protection to be placed on the mapping.
- It can be either PROT_NONE (pages may not be accessed at all)
  (This is used to put memory fences around memory areas that we do not want to be trespassed inadvertently
- It can be a combination (ORing) of any of the other three flags
  - PROT_READ (pages may be read)
  - PROT_WRITE (pages may be written)
  - PROT_EXEC (pages may be executed)

mmap() System Call

```
void *mmap(void * addr , size_t len , int prot,
           int flags, int fd , off_t offset)
```

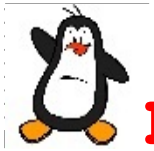
The **flags** argument is a bit mask of options controlling various aspects of the mapping operation. Exactly one of the following values must be included in this mask:

- **MAP_PRIVATE**: Any modifications into the mapped region cause a copy of the mapped file to be created. These modifications affect the copy, not the original file. Moreover, these modifications are not visible to other processes employing the same mapping, and, in the case of a file mapping, are not carried through to the underlying file.
- **MAP_SHARED:** Modifications into the mapped region are visible to other processes mapping the same region, and, in the case of a file mapping, are carried through to the underlying file. However, updates to the file are not guaranteed to be immediate. (see **`msync()`**)

mmap() System Call

```
void *mmap(void * addr , size_t len , int prot,
           int flags, int fd , off_t offset)
```

- The last two arguments, fd and offset, are used with file mappings
- The **fd** argument is a file descriptor identifying the file to be mapped.
- The **offset** argument specifies the starting point of the mapping in the file, and must be a multiple of the system page size. To map the entire file, we would specify offset as 0 and len as the size of the file.
- They are ignored for anonymous mappings. We normally put -1 for fd and a zero for offset for anonymous mapping).



mmap() System Call

```
void *mmap(void * addr , size_t len , int prot,
           int flags, int fd , off_t offset)
```

- A code snippet that privately maps a file in a process virtual address space is shown below:

```
int fd = open("myfile.txt", O_RDWR);
addr = mmap(NULL,len,PROT_READ | PROT_WRITE, MAP_PRIVATE,fd,0);
if (addr == MAP_FAILED)
   perror("mmap");
```

msync() System Call

```
int msync(void *addr , size_t len ,  int flag)
```

- The **msync** function causes the changes in part of all of the memory segment to be written back to (or read from) the mapped file. The part of the file that corresponds the memory area starting at **addr** and having length **len** is updated
- **flags** parameter controls how update should be performed

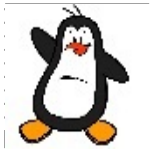
Flag	Work
MS_ASYNC	Performs asynchronous writes
MS_SYNC	Performs synchronous writes
MS_INVALIDATE	Read data back in from the file

munmap() System Call

```
int munmap(void * addr, size_t len)
```

- This simply unmaps the region pointed to by **addr** (returned from mmap) with length **len**.
- Normally, we unmap an entire mapping. Thus, we specify addr as the address returned by a previous call to `mmap()`, and specify the same length value as was used in the `mmap()` call.
- Any further references to addresses within the address range will generate invalid memory references
- Closing the file **fd** does not unmap the region
- The region is also automatically unmapped when a process terminates or performs an **exec**

```
if(munmap(addr, len) == -1)
    perror("mmap");
```



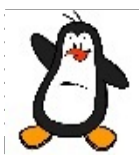
Memory Mapped I/O

Memory Mapped I/O

- An example of Shared File Mapping.
- Memory mapped I/O lets us map a file on disk into buffer in memory so that, when we fetch bytes from the buffer, the corresponding bytes of the file are read
- Similarly, when we store data in the buffer, the corresponding bytes are automatically written to the file on disk
- This lets us perform I/O without using read or write. Moreover, memory mapped files can be very useful, especially on systems that don't support shared memory segments

Example: Getting IDs

```
//sysprogramming/mmap/mmap2.c
$ a.out   file1.txt   7

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

int fd = open(argv[1], O_RDWR);

struct stat sbuff;

stat(argv[1], &sbuff);

int offset = atoi(argv[2]);

char* data = mmap(NULL, sbuff.st_size, PROT_READ
                | PROT_WRITE, MAP_SHARED, fd, 0);

printf("Byte at offset %d is %c \n", offset,
         data[offset]);

printf("\nComplete data in the file is:\n
         %s\n",data);
```



Things To Do

If you have problems visit me in counseling hours. . . .