Memory Mapping

Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

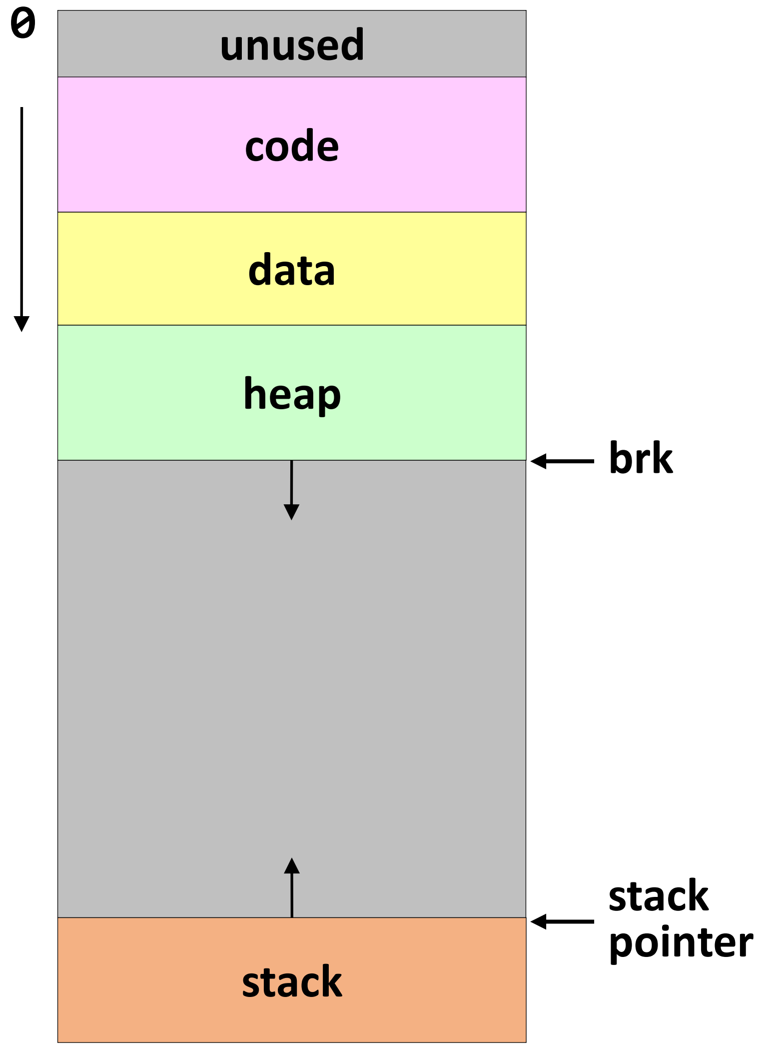
Seoul National University

Spring 2024

Virtual Memory Area

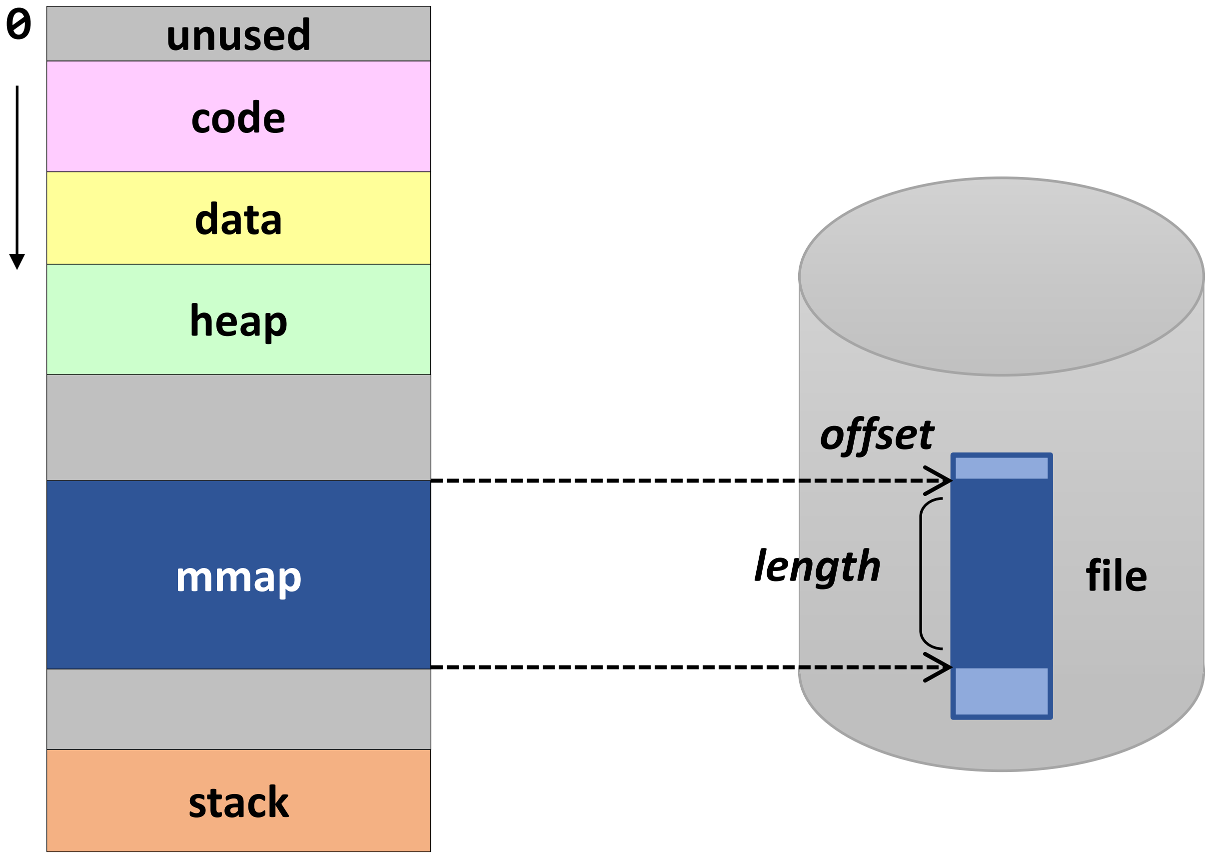
- Virtual address space is a resource
  - Every memory area should be allocated in the virtual address space
  - If you run out of the virtual address space, you can not access any more memory
    (even if you have space in the physical memory)

- Some of memory areas are backed by files and some aren't



Memory Mapping

- A dynamically allocated virtual memory area that has a backing store
  - File
  - Device memory
  - Shared memory
  - None



File vs. Anonymous Mapping

- File mapping (memory-mapped file)
  - Backing store: regular file
  - Maps a memory region to a file region
  - The content of the file can be read from or written to using load/store instructions

- Anonymous mapping
  - Virtual address space not backed by a file
  - Maps a memory region to a memory area filled with 0
  - Zero-page mapping

Shared vs. Private Mapping

- Several processes can map the same backing store in their own virtual address space

- Shared mapping
  - Modifications to shared pages are visible to all involved processes
- Private mapping
  - Modifications are not visible to other processes
  - Copy-on-write

	File mapping	Anonymous mapping
Private	Private file mapping	Private anonymous mapping
Shared	Shared file mapping	Shared anonymous mapping

mmap()

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

- Creates a new mapping in the virtual address space of the calling process
  - `addr`: the starting address for the new mapping (should be aligned to page boundary)
    - If NULL, the kernel chooses the address
    - Otherwise, the kernel takes it as a hint about where to place the mapping
  - `length`: the length of the mapping
  - `prot`: protection info. (`PROT_EXEC`, `PROT_READ`, `PROT_WRITE`, `PROT_NONE`)
  - `flags`: mapping flags (`MAP_PRIVATE`, `MAP_SHARED`, `MAP_ANONYMOUS`, …)
  - `fd`, `offset`: file descriptor & file offset (used for file mapping)

Memory-Mapped File: Example

- Allows processes to perform file I/O using memory references
  - Instead of `open()`, `read()`, `write()`, `close()`, etc.
  - Map a file to a virtual memory region

```
#include <sys/mman.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    int fd = open("/bin/ls", O_RDONLY);
    char *p = (char *) mmap(0, 4096, PROT_READ, MAP_SHARED, fd, 0);
    printf("0x%02x 0x%02x 0x%02x 0x%02x\n", *p, *(p+1), *(p+2), *(p+3));
    close(fd);
}
```

Memory-Mapped File

- Implementation
  - Initially, all pages in mapped region are marked as invalid
  - OS reads a page from file whenever invalid page is accessed
  - PTEs map virtual addresses to page frames holding file data
  - <Virtual address base + *n*> refers to `offset` + *n* in file

- Writes to the memory-mapped area
  - If MAP_SHARED,
    OS writes to a page and it is written to the file when evicted from physical memory
  - If MAP_PRIVATE,
    OS creates a private copy and then write data to the page. (a.k.a. Copy-On-Write)
    File is not modified.

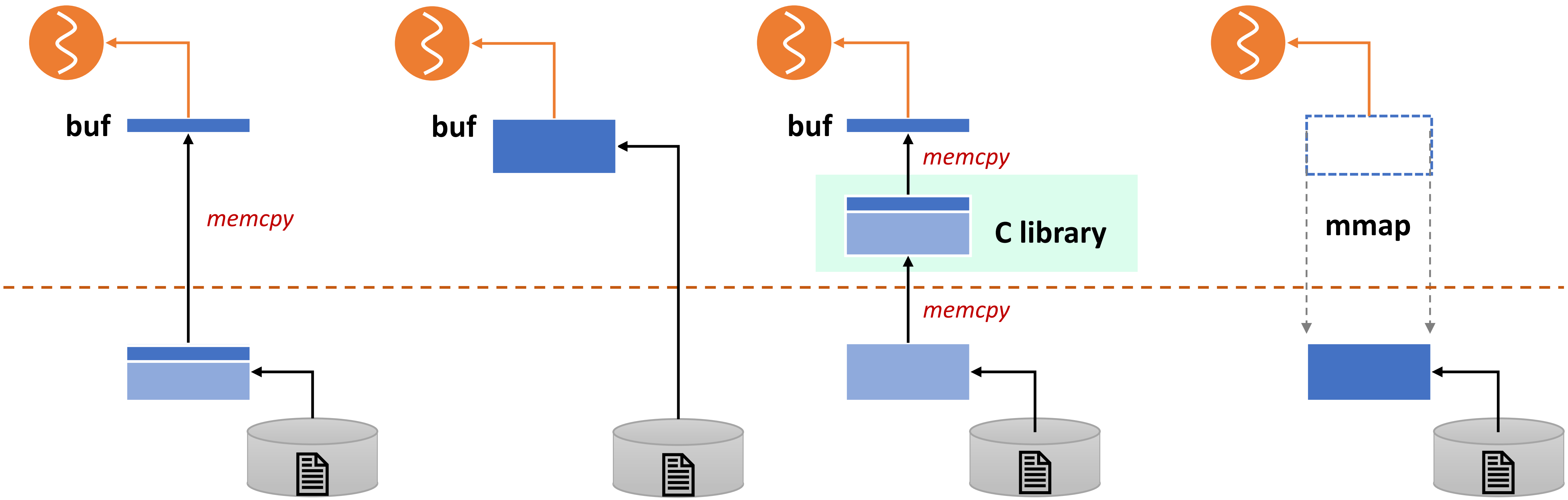
File I/O Comparisons

```c
char buf[1024];
int fd = open("a",...);
read(fd, buf, 1024);
```

```c
char buf[4096];
int fd = open("a",...,
              O_DIRECT);
read(fd, buf, 4096);
```

```c
char buf[1024];
FILE *fp = fopen("a","r");
fgets(buf, 1024, fp);
```

```c
int fd = open("a",...);
char *p = mmap(0,.., fd, 0);
```



Summary: Memory-Mapped File

- Pros
  - Uniform access for files and memory (just use pointers)
  - Less memory copying
  - Several processes can map the same file allowing the pages in memory to be shared

- Cons
  - Process has less control over data movement
  - Does not generalize to streamed I/O (pipes, sockets, etc.)

Shared Memory: Example

- Allows (unrelated) processes to share data using direct memory reference

```
#include <sys/mman.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    int fd = shm_open(“/shm1”, O_CREAT | O_EXCL | O_RDWR, 0600);
    ftruncate(fd, 4096);      // set shmem size
    int *p = (int *) mmap(0, 4096, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    for (int i = 0; i < 1024; i++)  p[i] = i;
    close(fd);
}
```

Shared Memory

- Implementation
  - Have PTEs in both tables map to the same physical frame
  - Each PTE can have different protection values
  - Must update both PTEs when a page becomes invalid

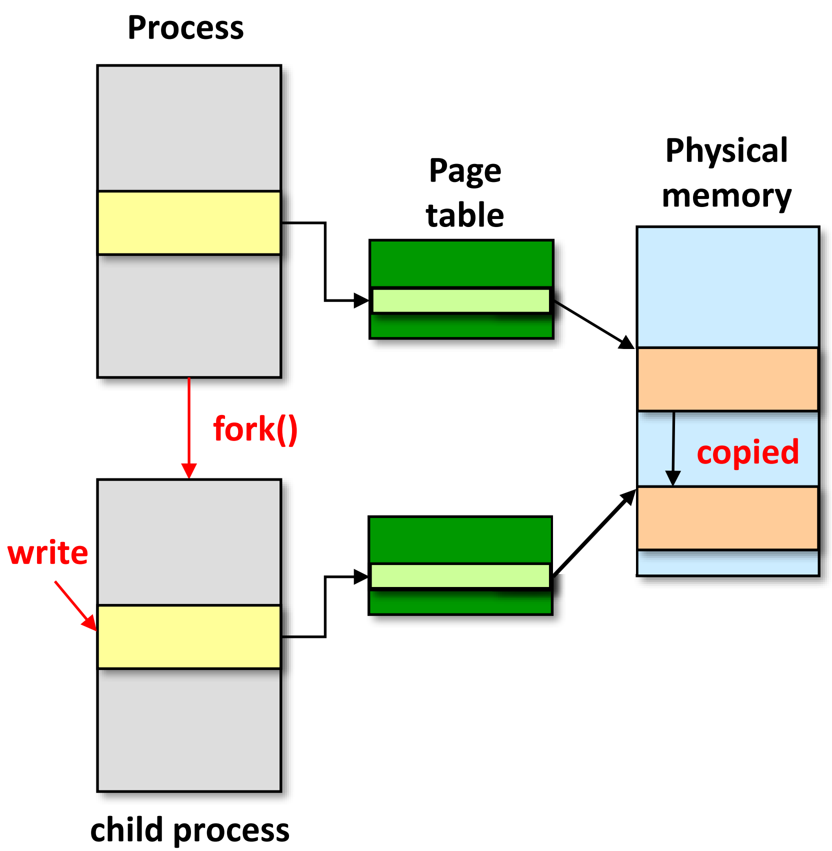
- Mapping shared memory in the virtual address space
  - At the different address: flexible (no address space conflicts), but pointers inside the shared memory are invalid
  - At the same address: less flexible, but shared pointers are valid

Copy-on-Write

- Defers memory copies as long as possible, hoping to avoid them altogether
- Implementation
  - Instead of copying pages, create shared mappings to the same page frames in physical memory
  - Shared pages are protected as read-only
  - When data is written to these pages, OS allocates new space in physical memory and directs the write to it
- Usage
  - `fork()`
  - Allocating data and heap pages, etc.

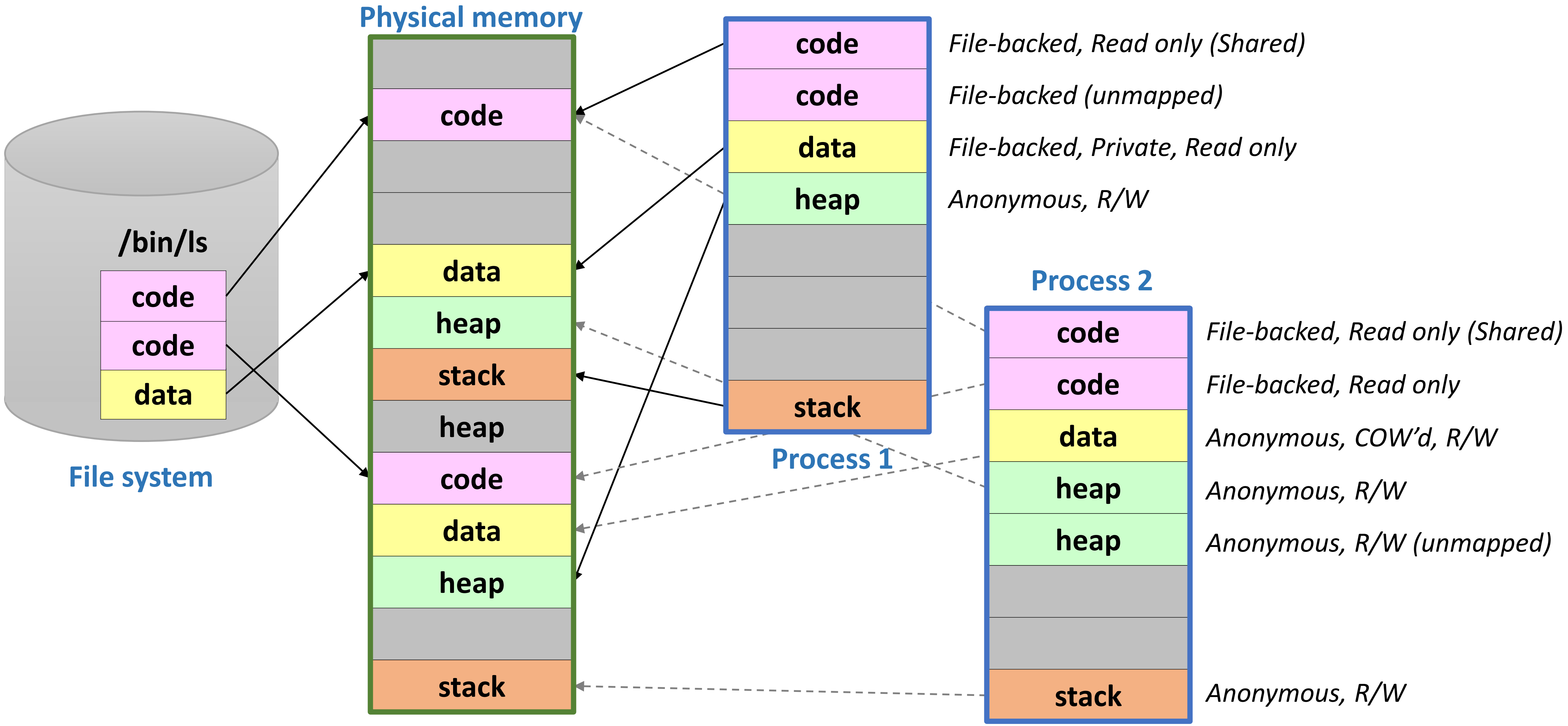
Copy-on-Write during fork()

- COW ensures that both processes do not see each other's changes
  - Instead of copying all pages, create shared mappings of parent pages in the child address space
  - Shared pages are protected as read-only
  - Reads happen as usual
  - Writes generate a protection fault and OS copies the page, changes page mapping, and restarts write instruction



- Efficient when the child process calls `exec()` immediately after `fork()`

Summary


Physical memory
code
data
heap
stack
heap
code
data
heap
stack
/bin/ls
code
code
data
File system
code
code
data
heap
stack
Process 1
File-backed, Read only (Shared)
File-backed (unmapped)
File-backed, Private, Read only
Anonymous, R/W
Process 2
code
code
data
heap
heap
stack
File-backed, Read only (Shared)
File-backed, Read only
Anonymous, COW'd, R/W
Anonymous, R/W
Anonymous, R/W (unmapped)
Anonymous, R/W