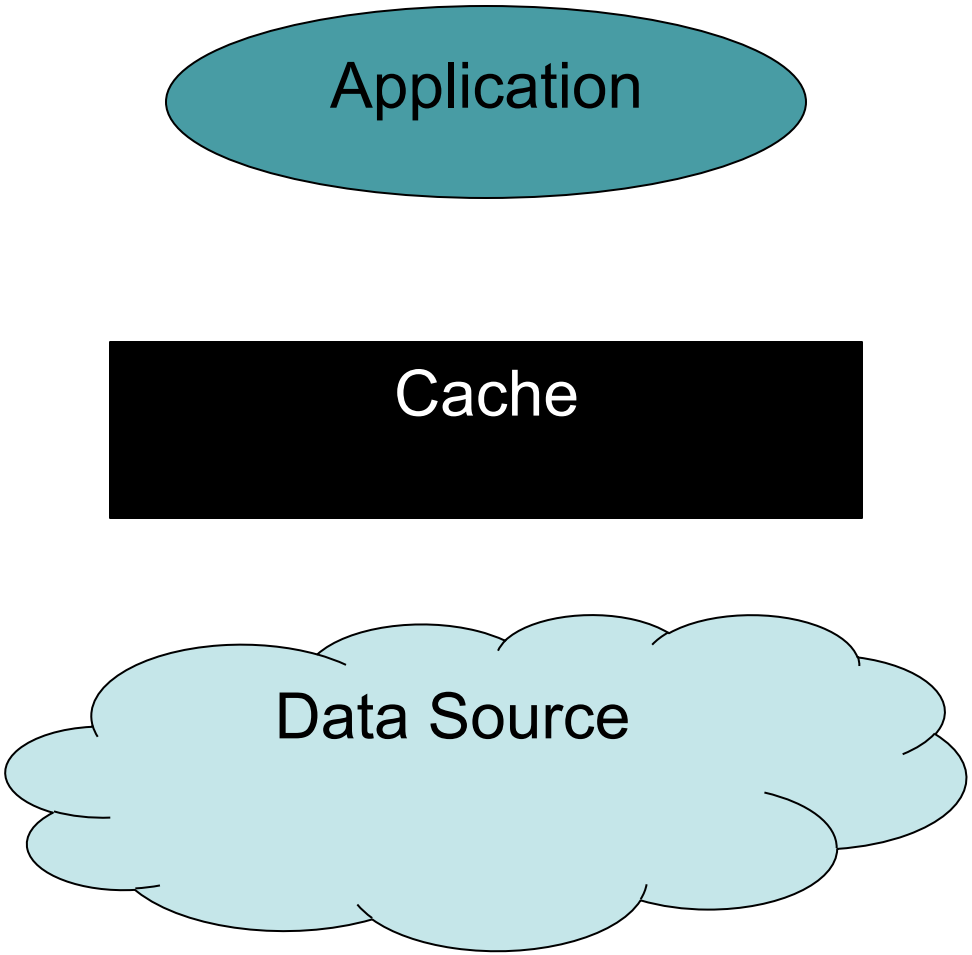
Storage 5: Data copying and Mmap

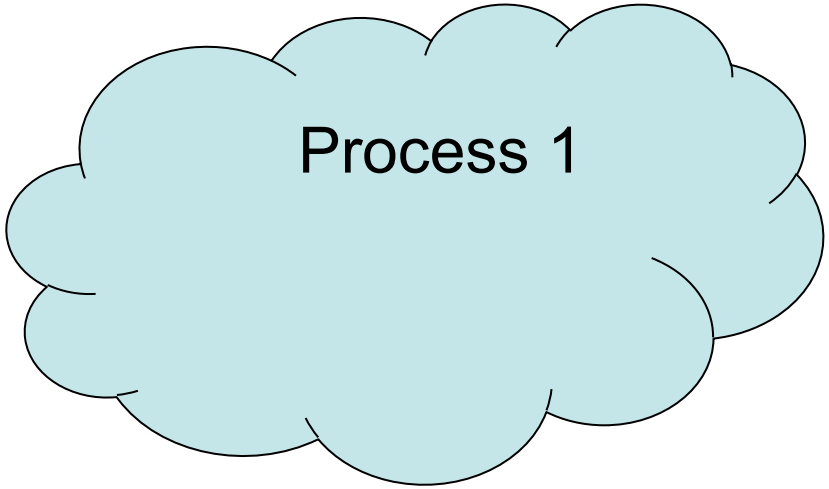
- Learning Objectives
  - Examine relationship between caching and copying
  - Use `mmap` (and it's associated calls).
  - Discuss the pros and cons of using `mmap`.

Caches and Copying

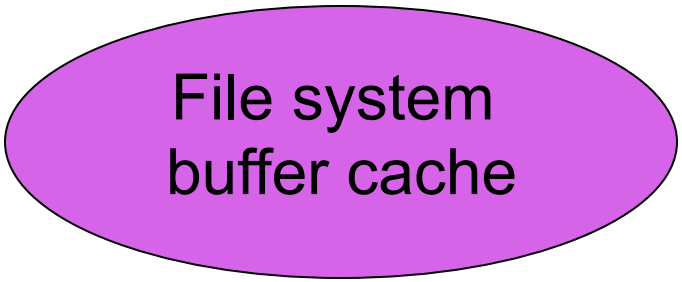


Copies using open/close/read/write

User programs

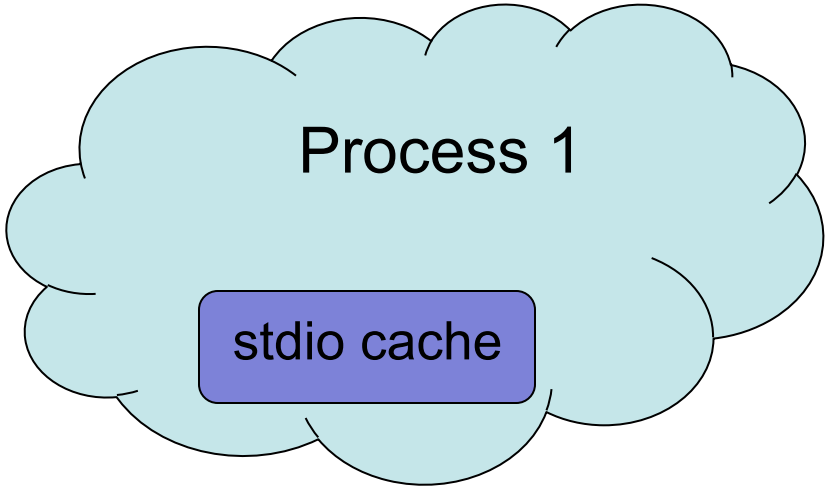


Kernel

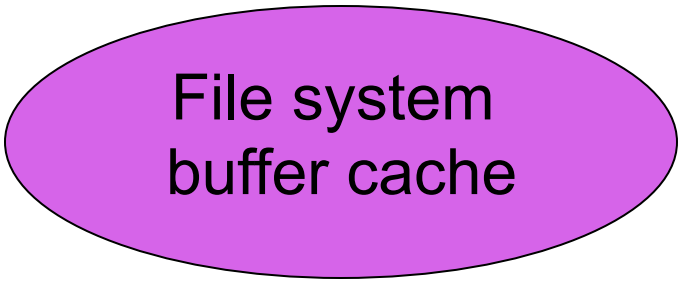


Copying with standard IO

User programs



Kernel

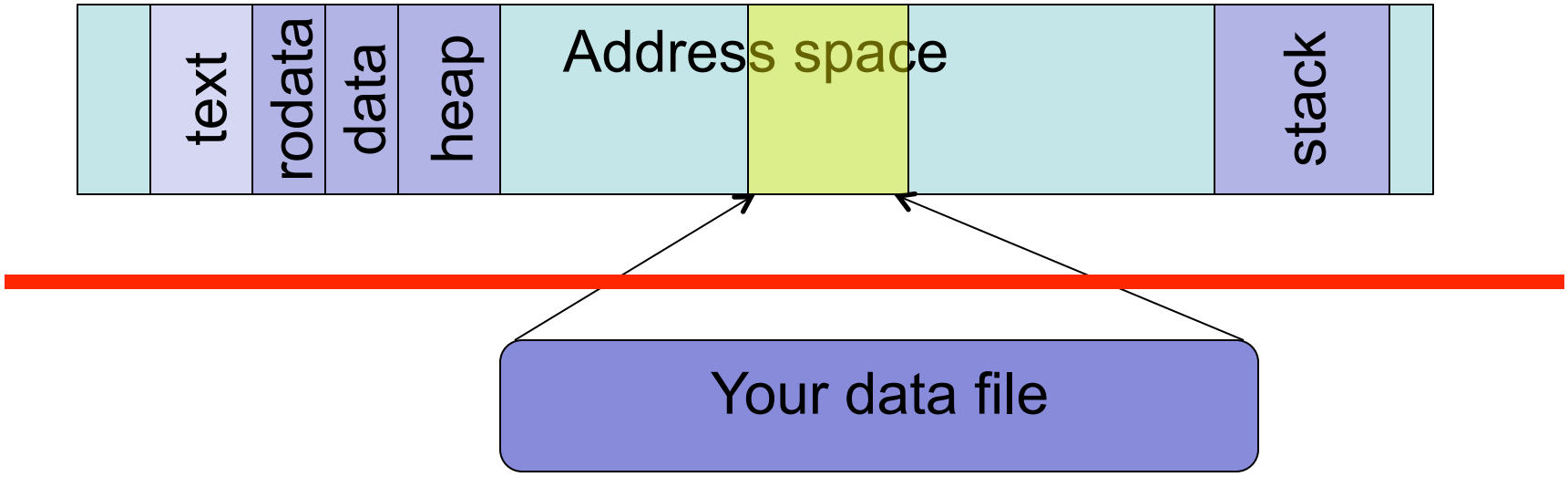


mmap

- A system call that "maps a file into a process's addess space."

Mmap

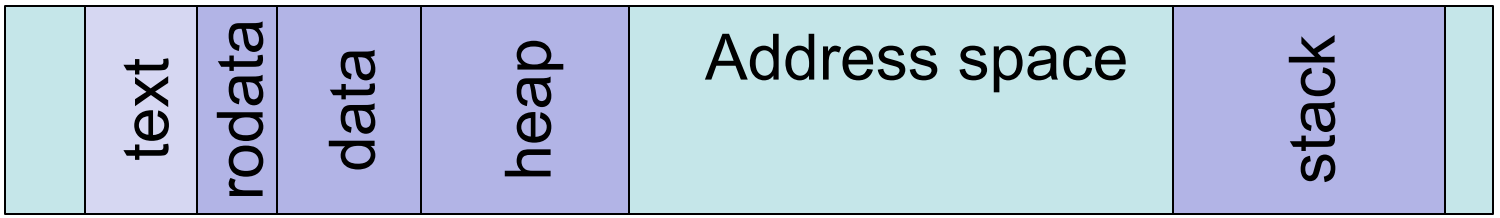
- A system call that "maps a file into a process's addess space."



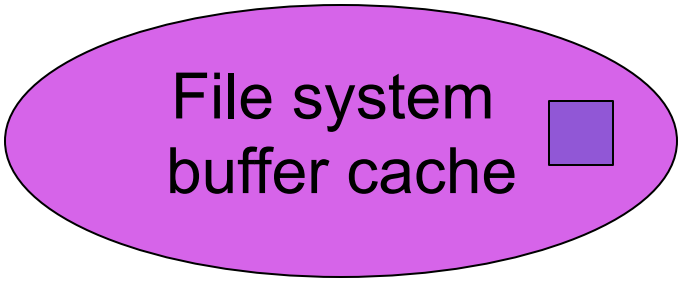
- Access data in the file directly by addresses (as you would in the buffer that you pass to read).

Compare `mmap` to `read`: `read`

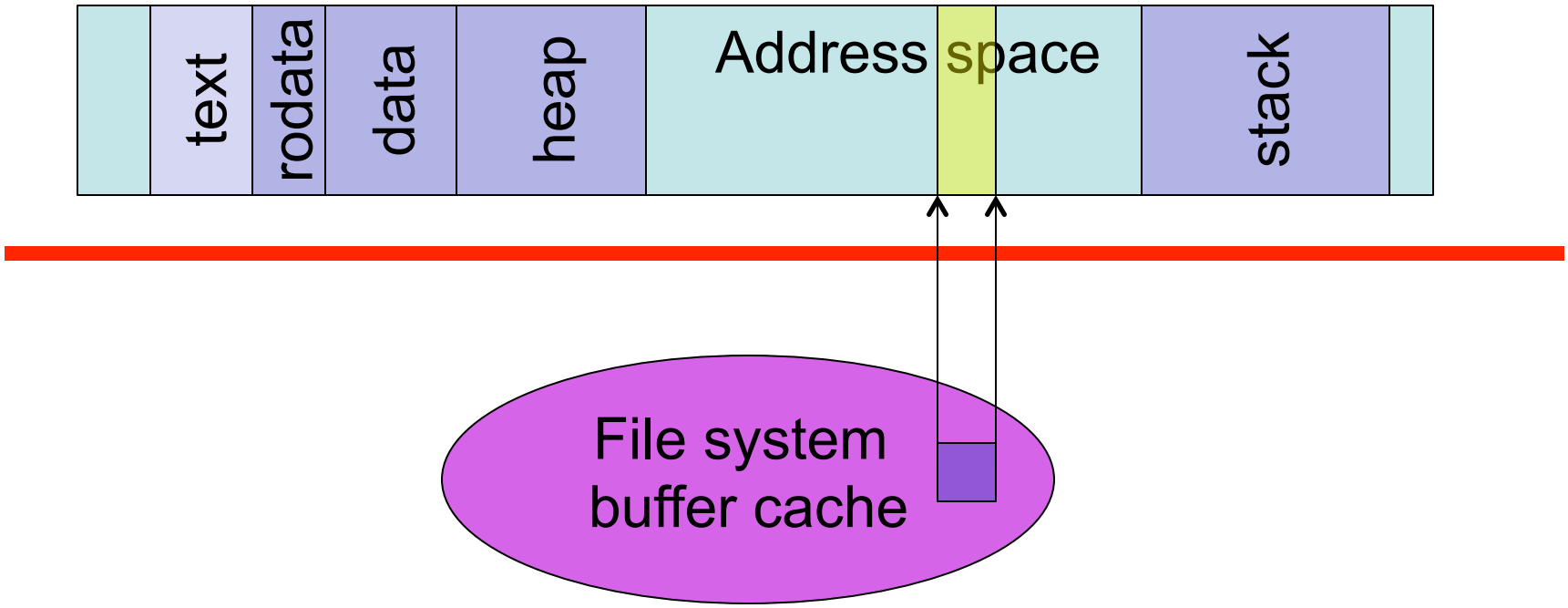
```
char buf1[4096];
int main(int argc, char *argv[]) {
        char buf2[4096];
        char *buf3 = malloc(4096);
        …
}
```



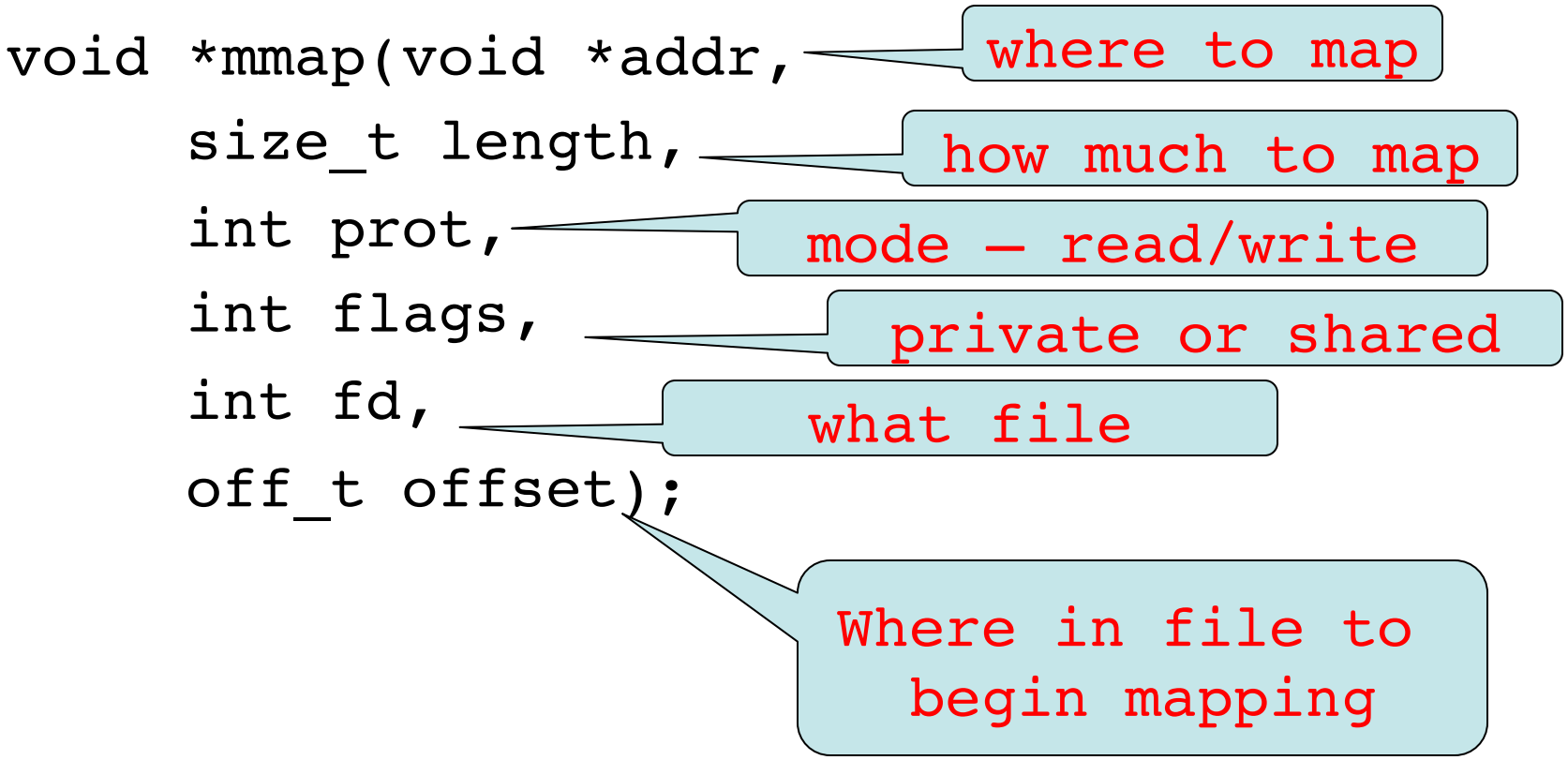

```
read(fd, buf1, 4096);    read(fd, buf3, 4096);    read(fd, buf2, 4096);
```



Compare `mmap` to `read`: `mmap`



mmap details



`mmap` Features and Limitations

- If you don't care where the file is placed, you can set `addr` to 0.
- `offset` (where in file mapping begins) must be a multiple of the pagesize (typically 4 KB).
- If `length` (how much data to map) is not a multiple of page size, then the mapping will be rounded up to the next pagesize boundary and bytes between length and the mapping will be 0-filled.
- Doesn't really work for growing files.
- You have no control about when changes get written back to the file (you can force them, but you can't prevent them).

Screen Capture

- Let's look at mmap.c
- Predict what will happen.
- Verify prediction.
- Lather rinse repeat with stdio.c/syscall.c

Other `mmap`-related calls.

- `int munmap(void *addr, size_t length);`
  - Delete the mapping starting at `addr`

- `int msync (void *addr, size_t length, int flags);`
  - Flushes changes made to the in-memory copy of the file to be reflected back to the disk (persistent store).

Why `mmap`?

- Mmap is sometimes the most efficient way to access data; why?
  - Fewer system calls: you make only one system call to map the file; then the rest of the processing that the system has to do is a side-effect of touching memory.
  - Fewer copies: both standard IO and read/write copy the data out of the operating system into a user buffer. `mmap` brings the data into memory and lets your application access that data directly.

Why not `mmap`?

- Why do we ever use read/write if mmap is so great?
  - Can't really grow files easily using `mmap`, so it's not great for creating a new files.
  - Although `msync` lets you force data to persistent storage, the application has no control over when data may be flushed back to persistent storage, so it is difficult to maintain on-disk data consistency using `mmap` in the presence of updates.
  - Requires block-alignment – not great for small files.
  - Doesn't work on all file types* (just regular files).

Different Types of Files

- When we say file, almost everyone thinks of plain old regular files – things in which we store data and read/write to/from persistent storage.
- However, UNIX systems (e.g., OSX, Linux, etc) use files and the file system interfaces as an abstraction for lots of things:
  - The terminal window (called tty)
  - Pipes
  - Directories
  - Devices
  - On Linux, even processes have a representation in the file system (/proc).