# IPC III: mmap, fds

CS 241

Nov. 1, 2013

# File-backed mmap

```
void main() {
  /* For a file-backed mmap, we need a fd... */
  int fd = open("apple.txt", O_RDWR);

  /* We also to know the length of the file... */
  // Seek to the end of the file and record how
  // far into the file we went (eg: length of file)
  int file_len = fseek(fd, 0, SEEK_END);

  // Rewind the file
  fseek(fd, 0, SEEK_SET);

  /* Create the mmap */
  void *ptr = mmap(NULL,
                   file_len,
                   PROT_READ | PROT_WRITE,
                   MAP_SHARED,
                   fd,
                   0);
```

# Anonymous mmap

- An anonymous mmap is effectively a malloc() that survives a fork().

- Same system call, with three differences:
  - **offset** field is ignored
  - **fd** must be -1
  - **flags** must contain MAP_ANONYMOUS

# Anonymous mmap

```
void main() {
  /* Simply create the mmap */
```

```
void main() {

}
```

```
void main() {









}
```

# Files on Linux

- In the beginning of this semester, we saw:

```
open() / read() / write() / close()
```

- There is also:

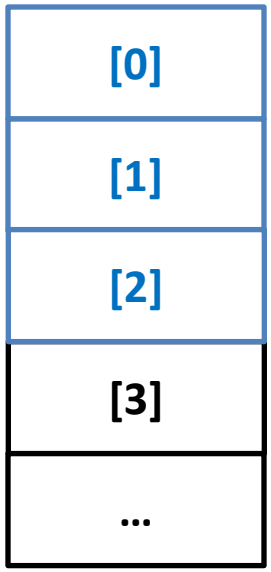
```
fopen() / fread() / fwrite() / fclose()
  fprintf() / fscanf() / getline()
```

# File Descriptor

- A **file descriptor** is a single, universal interface on Linux that works for every stream-based interface.
  - 
  - 
  - 

# File Descriptor Table

- Each process maintains its own **file descriptor table** that maps a fd to the underlying **stream**.



# Every entry has some properties...

- Every **stream** in Linux has at least one property:
  - 
- Some streams have other properties:
  - 
  - 
  - 

# File Descriptor vs (FILE *)

- In Windows and other OSs, file descriptors works slightly differently.

- The “C library” uses a (**FILE ***) as the data structure to store system-specific file information.
  - FILE * ➔ fd
    - **`int fileno(FILE *stream)`**
  - fd ➔ FILE *
    - **`FILE *fdopen(int fd)`**

# I/O Multiplexing

- By default: **read()** / **fread()** are blocking calls.
  - …if no data is available, the process will be moved to the BLOCKED state until data is available.

- In order to read() from multiple files in one thread at one time, **I/O multiplexing** is required.
  - **epoll()**: *monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready".*

# epoll() Overview

- Usage of epoll():
  - *Create an epoll instance via **epoll_create()***
  - *Register each file descriptor to watch via **epoll_ctl()***
  - *Use **epoll_wait()** to block until an fd is ready*
  - *(Replaces both select() and poll() system calls.)*