Unix Inter-Process Communication (IPC)

W4118 Operating Systems I

columbia-os.github.io

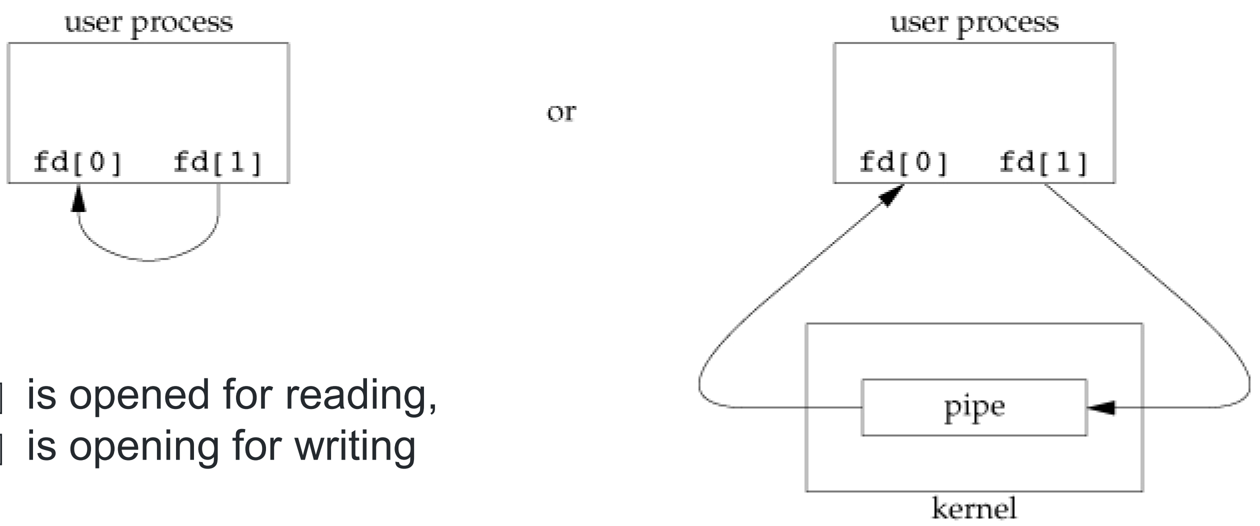
Credits to Jae

Unnamed Pipes

```
#include <unistd.h>
```

```
int pipe(int fd[2]); // Returns: 0 if OK, -1 on error
```

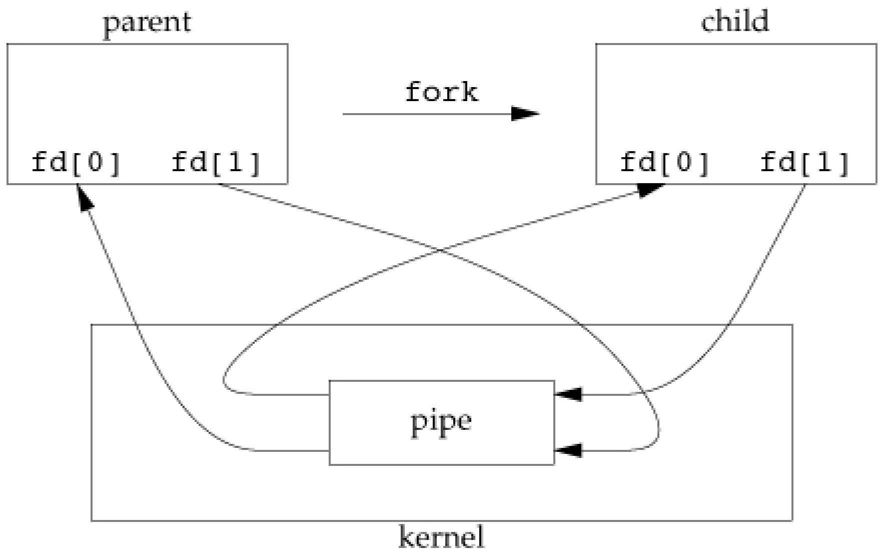
After calling `pipe()`



`fd[0]` is opened for reading,
`fd[1]` is opening for writing

Unnamed Pipes – Parent & Child

`pipe()` and then **`fork()`**:



`connect2.c` example

!!! pipes are only half-duplex !!!
(one-way communication)

Q: What happens if parent read writes to fd[1] and reads from fd[0]?

Q: How can unrelated process communicate with each other?

Named Pipe

```
#include <sys/stat.h>
```

```
int mkfifo(const char *path, mode_t mode); // Returns: 0 if OK, -1 on error
```

- `mkfifo()`: create a new named pipe on the filesystem
- Use file I/O syscalls to interact with special pipe file
- Shares semantics with unnamed pipe – still half-duplex

Semaphores

Definition: Integer value mainly manipulated by two methods

- **Increment:** increase the value of the integer
  - `sem_post()`
- **Decrement:** wait until value > 0, then decrease the integer value
  - `sem_wait()`
  - **Blocking semantics**: unlike increment, decrement blocks until value is positive

Semaphore Semantics

Initial value affects semaphore semantics:

- **Binary semaphore** (a.k.a. lock): initial value is 1. Protects one resource.
    - Before acquiring the resource, run `sem_wait()`, value -> 0
    - Use resource
    - Run `sem_post()` to release the resource, value -> 1
- **Counting semaphore:** initial value is N > 1. Protects N resources.
    - Before acquiring the resource, run `sem_wait()`, value -> value - 1
    - Use resource
    - Run `sem_post()` to release the resource, value -> value + 1
- **Ordering semaphore:**

```
sem = 0  // initial value is 0
```

```
P1: 1 -> 2 -> sem_wait() -> 4 -> 5
```

```
P2: A -> B -> C -> D -> sem_post()
```

Semaphore POSIX API

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
        // Returns: 0 if OK, -1 on error
int sem_destroy(sem_t *sem);
        // Returns: 0 if OK, -1 on error
```

Q: If semaphore is to be shared by related processes, where should semaphore be declared?

Semaphore POSIX API

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
        // Returns: 0 if OK, -1 on error
int sem_destroy(sem_t *sem);
        // Returns: 0 if OK, -1 on error
```

Q: If semaphore is to be shared by related processes, where should semaphore be declared?

1. Shared memory, see `mmap()` in a bit
2. Named semaphore

Named Semaphores

Similar semantics to named pipes

On Linux, named semaphores are stored in the filesystem under `/dev/shm`

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag, ...
                /* mode_t mode, unsigned int value  */ );
        // Returns: Pointer to semaphore if OK, SEM_FAILED on error
int sem_close(sem_t *sem);
        // Returns: 0 if OK, -1 on error
int sem_unlink(const char *name);
        // Returns: 0 if OK, -1 on error
```

Decrement/Increment Semaphore Options

- `sem_trywait()` does NOT block, returns immediately if semaphore value is 0.
- `sem_wait()` blocks until semaphore value is positive
  - Sets `errno` to `EINTR` if interrupted by a signal
- `sem_timedwait()` blocks until it times out or semaphore value is positive, whichever happens first
  - Can `sem_timedwait()` be safely implemented using `SIGALRM`?
- `sem_post()` does not block

File I/O syscalls are kind of annoying

- Editing/accessing different parts of the files: have to keep calling `lseek()`
- Reading from the file requires `read()` to copy contents out of kernel to userspace buffer
- Writing to the file requires `write()` to copy contents out of userspace buffer into kernel

What is the alternative?

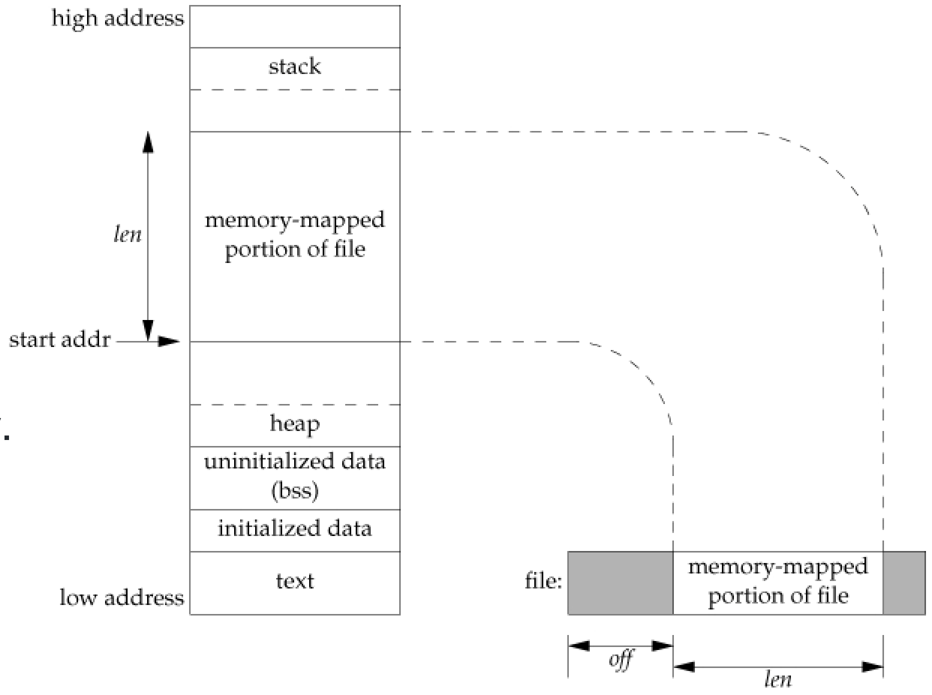
Memory-mapped I/O

Map region of your file into your virtual address space!!!

Updates to the memory-mapped region go to memory first, then (eventually) flushed to disk

Private mapping: changes are not flushed to disk and are not seen by other processes that map the same region.

Shared mapping: reference the same memory. Processes with shared mappings see each other's updates



mmap()

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

void *mmap(void *addr, size_t len, int prot, int flag, int fd, off_t off);

// Returns: starting address of mapped region if OK, MAP_FAILED on error
```

- `void *addr`: Virtual address to place the mapping at. Prefer to pass `NULL` and let `mmap()` decide for you (address is the return value).
- `int prot`: Protection of the mapped region (read, write, exec, none)
- `int flag`: Visibility (shared/private) + other modifiers
- `int fd`: file descriptor attached to file we want to map

Anonymous mappings

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

```
void *mmap(void *addr, size_t len, int prot, int flag, int fd, off_t off);
```

specify `fd = -1` and `flag = MAP_ANON | …`

`mmap()` more powerful than `malloc()` (and does something different)

- `MAP_PRIVATE`: child gets its own independent copy of the mapping (like `malloc()`)
- `MAP_SHARED`: child shares memory mapping with parent, both see each other’s updates

`counter.c` example