CSCB09: Software Tools and Systems Programming

Bianca Schroeder
bianca@cs.toronto.edu
IC 460

Short announcement

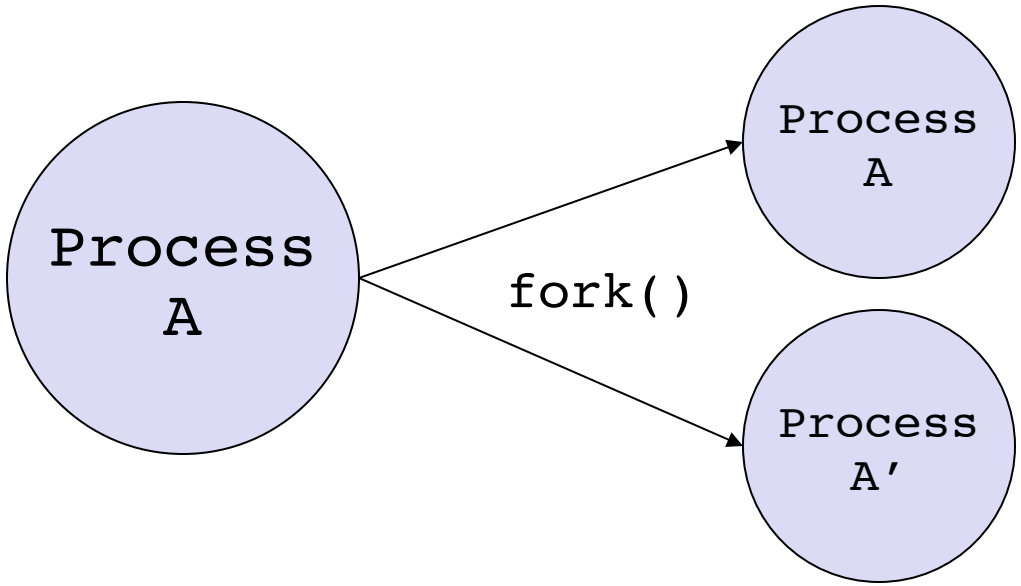
- I will be travelling next week. This wasn't planned, but my grad student is affected by the Trump travel ban and cannot go to present our paper.
- One of my grad students will teach the class (covering signals).

Processes can create other processes

- The fork system call creates a child process:

```
#include <unistd.h>
pid_t fork(void);
```

- Fork creates a **duplicate** of the currently running program.
- Both processes run concurrently and independently.
- After fork() both execute the next instruction after fork.



waitpid

- What if a process wants to wait for a particular child (rather than any child)
- What if a process does not want to block when no child has terminated?

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- First parameter specifies PID of child to wait for
- If options is 0, waitpid blocks (just like wait)
- If options is WNOHANG, it immediately returns 0 instead of blocking when no terminated child

How does a child become a zombie?

- When a child terminates, but its parent process is not `wait`ing for it
- The child (its exit code) is kept around as a zombie until parent collects its exit code through `wait`
  - or until parent terminates
- Shows up as `Z` in `ps`

How does a child become an orphan?

- If the parent process terminates before the child
- Who is now the new parent?
  - Orphans get adopted by the `init` process
  - `init` is the first process started during booting
  - It's the root of the process hierarchy
  - `init` has a PID of 1
  - The PPID of orphans is 1

Wait a second ….

- Fork creates a duplicate of the current process
- How do we actually create a new process that runs a different program???

exec()

- exec() is not one specific function, but a family of functions:

```
execl(char *path, char *arg0, …, (char *)NULL);
execv(char *path, char *argv[]);
```

```
execlp(char *file, char *arg0, …,(char *)NULL);
execvp(char *file, char *argv[]);
```

- First parameter: name of executable; then commandline parameters for executable; these are passed as argv[0], argv[1], …, to the main program of the executable.

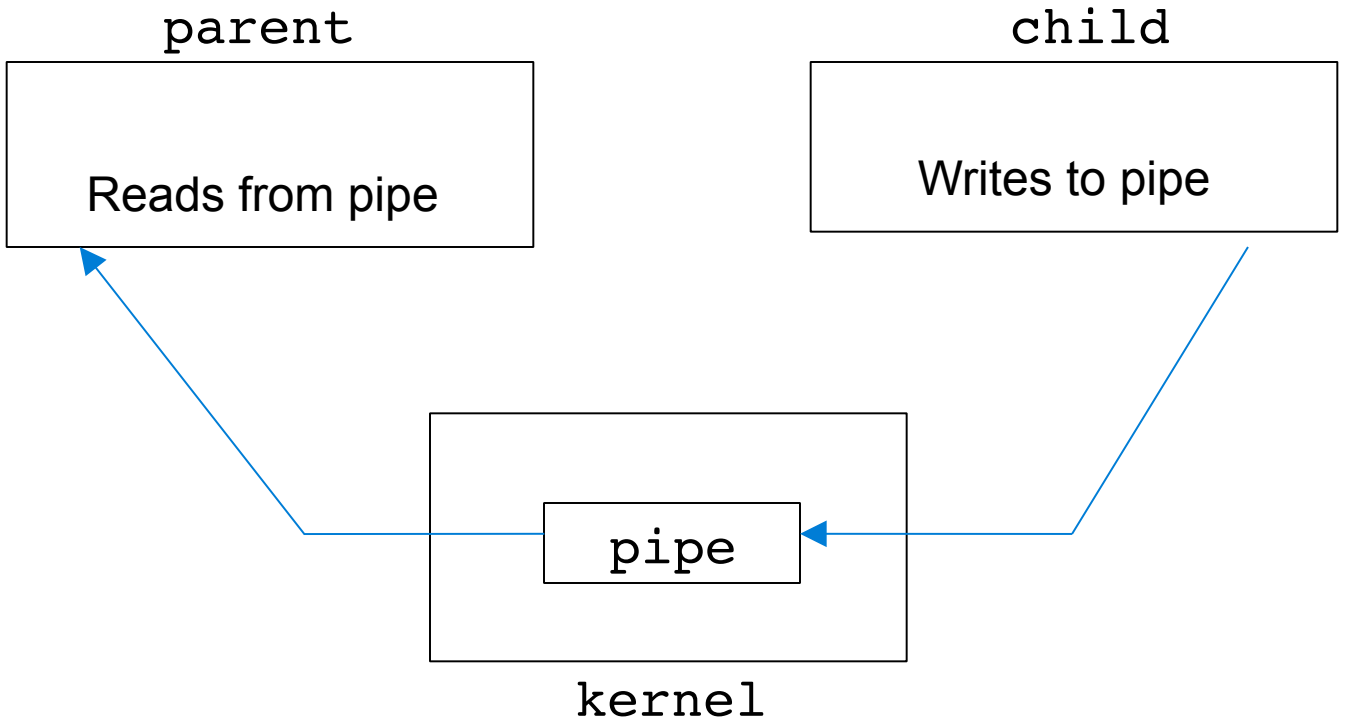
- `execl` and `execv` differ from each other only in how the arguments for the new program are passed
- `execlp` and `execvp` differ from `execl` and `execv` only in that you don't have to specify full path to new program

Processes often need to communicate

- E.g. the different worker processes of a parallel program need to exchange data and/or synchronize

- Options:
  - Last worksheet used exit code – limited solution…
  - Cannot use variables, since after fork each process has separate copy of variables
  - Use files --- coordination is difficult

Solution: Pipes



- Pipes are a one-way (half-duplex) communication channel
- Pipes are buffers managed by the OS
- Processes use low-level file descriptors for pipe operations

Recall: I/O mechanisms in C

- So far we used: File pointers (regular files):
  - You use a pointer to a file structure (`FILE *`) as handle to a file.
  - The file struct contains a file descriptor and a buffer.
  - Use for regular files

```
FILE *fp = fopen("my_file.txt", "w");
fprintf(fp, "Hello!\n");
fclose(fp);
```

- Now we need: File descriptors (low-level):
  - Each open file is identified by a small integer
  - Operations: `open, close, read, write`

File I/O with file descriptors

- `int open(const char *pathname, int flags);`
  - Returns a file descriptor
  - Flags: O_RDONLY, O_WRONLY, or O_RDWR to open the file read-only, write-only, or read/write (and a few more, see man pages)
- `ssize_t read(int fd, void *buf, size_t count);`
  - Returns #bytes read, 0 for EOF, -1 for error
- `ssize_t write(int fd, const void *buf, size_t count);`
  - Returns #bytes actually written, -1 for error
- `int close(int fd);`
  - Returns 0 on success, -1 on error

- Example (error checking omitted …):

```
char buf[6];
int fd = open (“filename.txt”, O_RDONLY);
read (fd, buf, 6);
close(fd);
```

Back to pipes

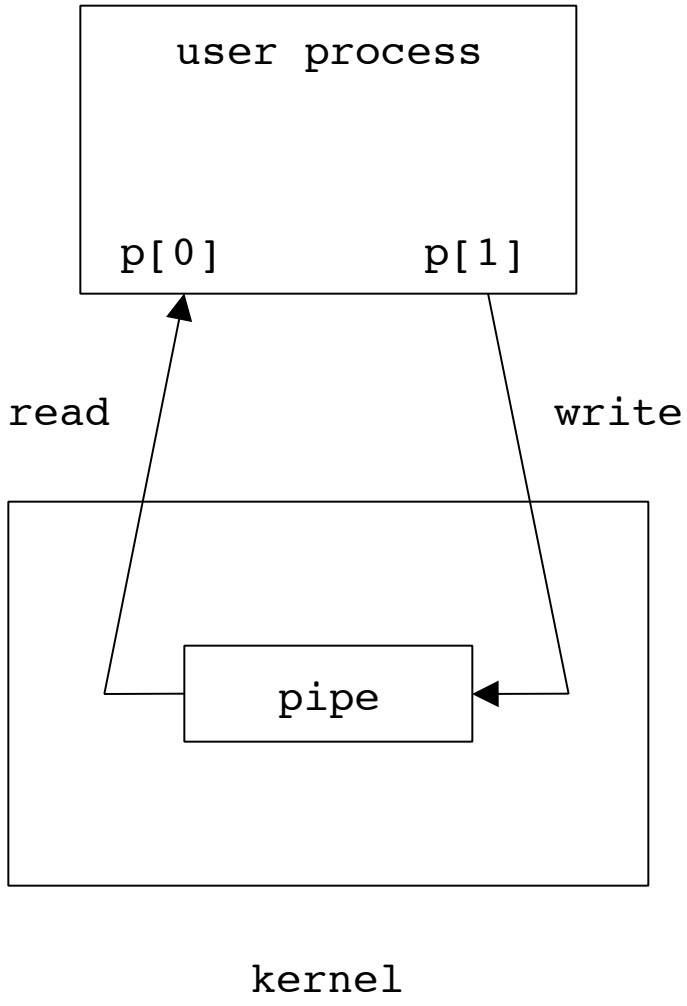
- How do you open/create a pipe?

```
int pipe(int pipefd[2]);
```

- You pass a pointer to two integers (i.e. an array or a malloc of two ints) and pipe fills it with two newly opened FDs.
- Returns 0 on success, -1 on error
- Example:

```
int p[2];
if (pipe(p)) == -1 ) {
    perror (“pipe”);
    exit(1);
}
```

- p[0] is now open for reading
- p[1] is now open for writing



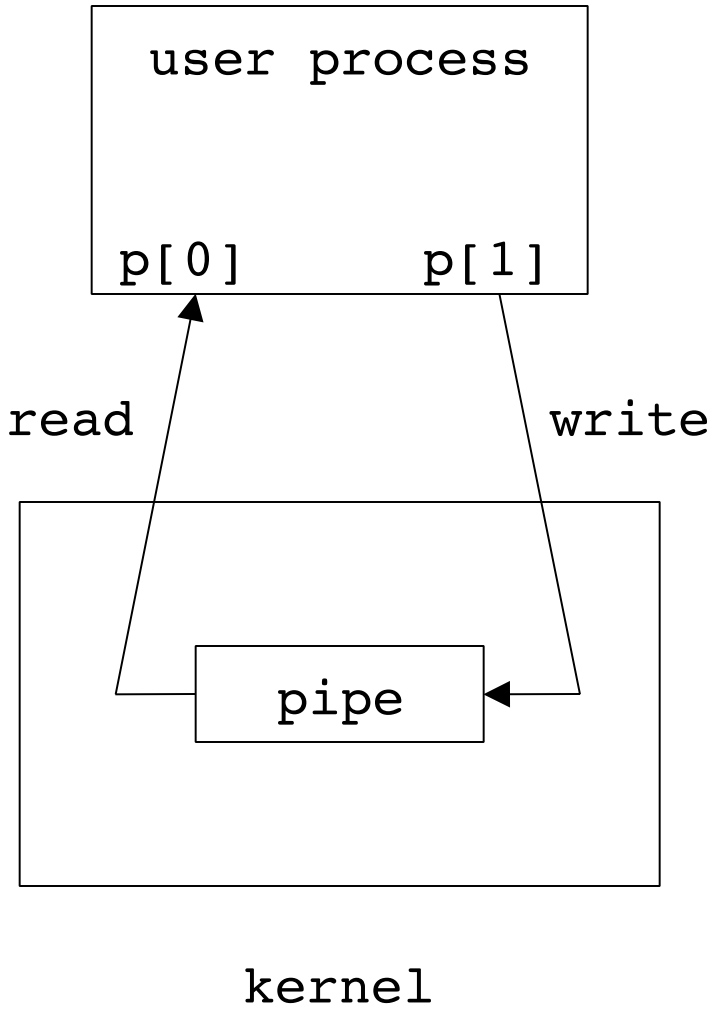
Example: Process talking to itself

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

#define MSG_SIZE 13
char *msg = "hello, world\n";

int main(void) {
  char buf[MSG_SIZE];
  int p[2];

  if(pipe(p) == -1) {
    perror("pipe");
    exit(1);
  }
  write(p[1], msg, MSG_SIZE);
  read(p[0], buf, MSG_SIZE);

  write (STDOUT_FILENO, buf, MSG_SIZE);

  return 0;
}
```

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

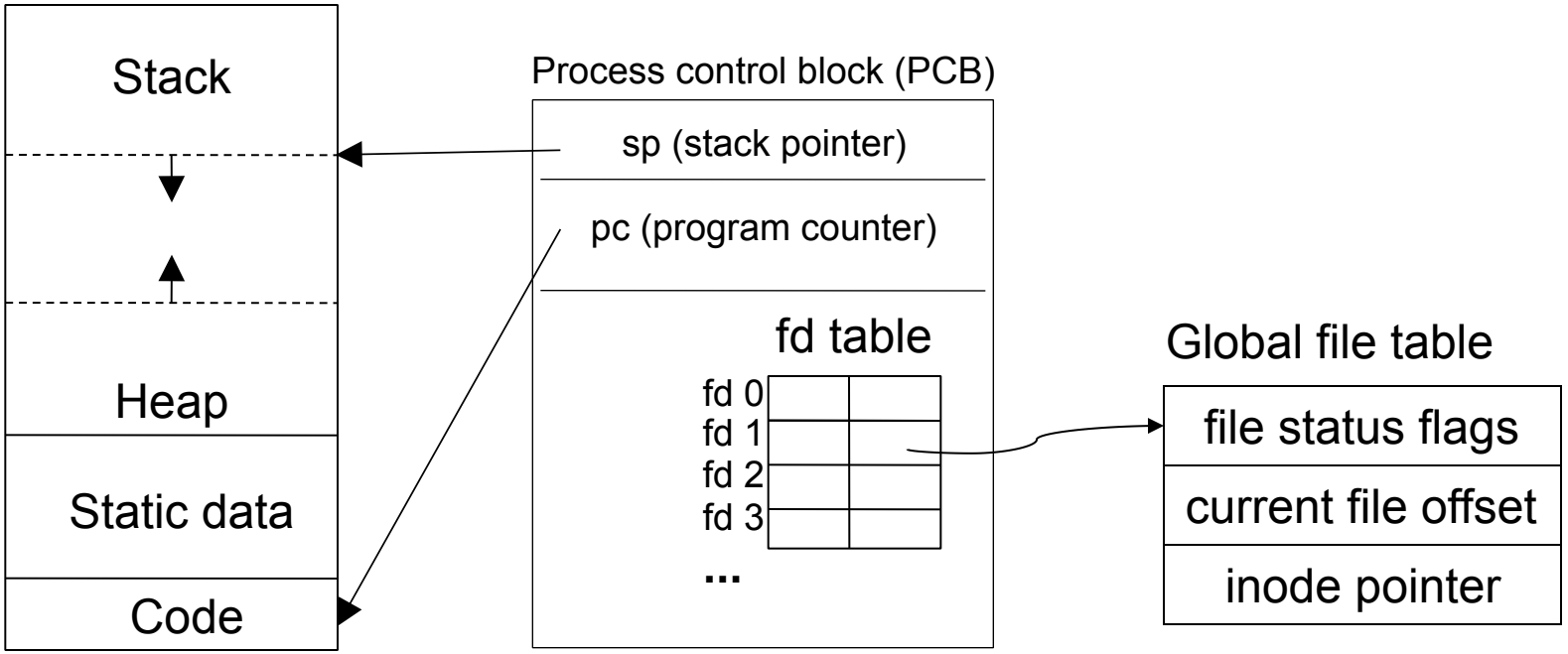
Now lets first write `msg` to the pipe and then read from the pipe

A process talking to itself is not very useful



How do we get two processes talking to each other using pipes?

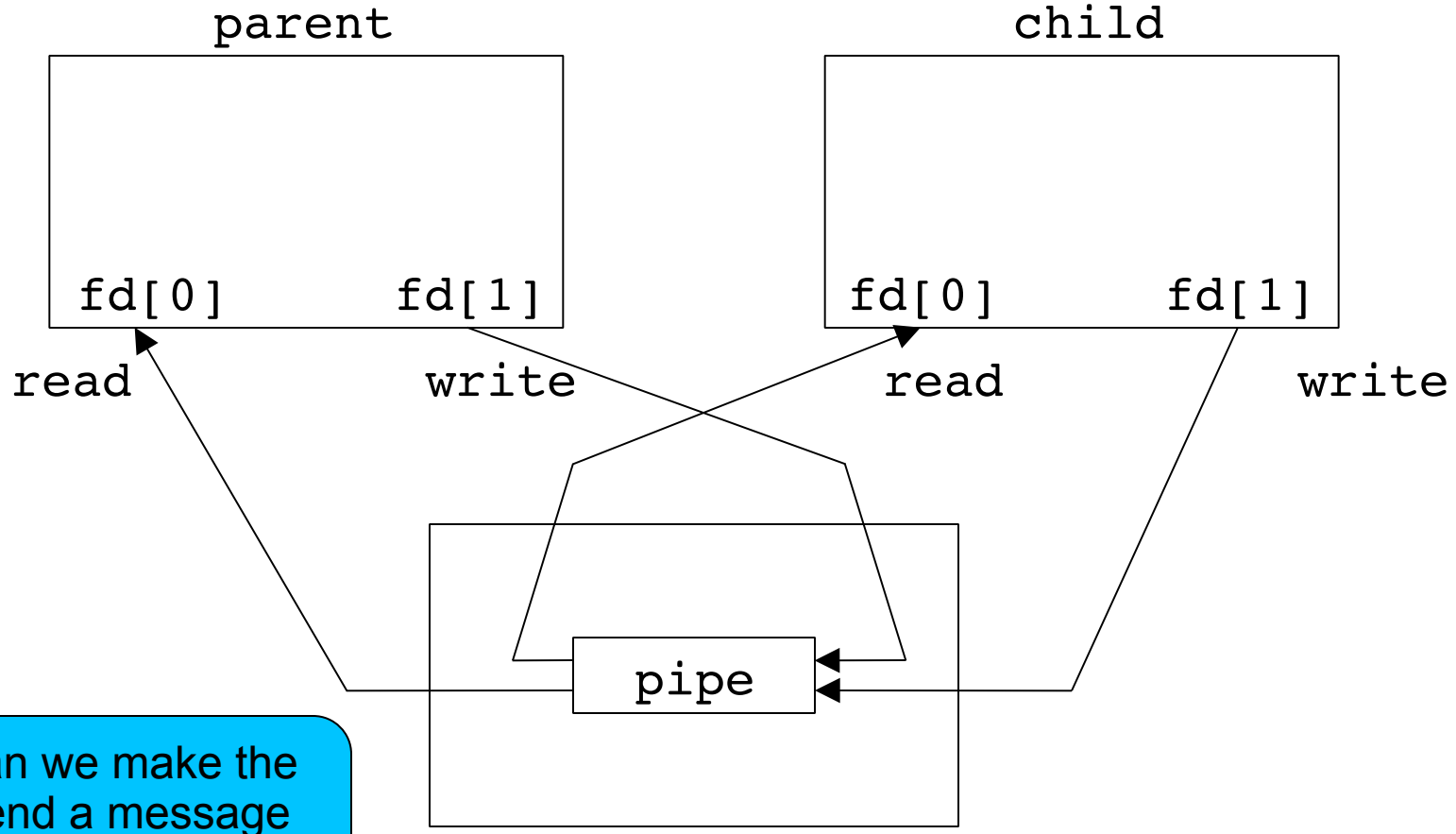
The OS manages file descriptors



- Per-process fd table has all open files of a process
- Global file table has all files open system-wide

- On fork, a child gets a copy of parent's fd table, so it will have same files open
- The fd table is preserved on exec

On fork, child gets copy of parent's file descriptors, including pipes



How can we make the child send a message to the parent?

Example 1: Child talking to parent

```
#define MSG_SIZE 13
char *msg = "hello, world\n";

int main(void) {
  char buf[MSG_SIZE];
  int p[2];
  if(pipe(p) == -1) {
    perror("pipe");
    exit(1);
  }
  if (fork() == 0) //Child writes
    write(p[1], msg, MSG_SIZE);
  else { //Parent reads from pipe and prints
    read(p[0], buf, MSG_SIZE);
    write (STDOUT_FILENO, buf, MSG_SIZE);
  }
  return 0;
}
```

Pipes and EOF

- What if the parent does not know beforehand when and how much data a child will write?

- If no writing end is open, `read` detects EOF and returns 0.
- `read` blocks until data is available in the pipe.
- All open pipes (and other FDs) are closed when a process exits.

Example 2 : Child talking to parent

```
#define BUF_SIZE 4 //Small!
#define MSG_SIZE 13
char *msg = "hello, world\n";

int main(void) {
  char buf[BUF_SIZE];
  int p[2], nbytes;
  if(pipe(p) == -1) {
    perror("pipe");
    exit(1);
  }
  if (fork() == 0)
    write(p[1], msg, MSG_SIZE);
  else {
    while ((nbytes = read(p[0], buf, BUF_SIZE)) > 0)
      write (STDOUT_FILENO, buf, nbytes);
  }
  return 0;
}
```

Parent hangs, even after child exits. Why is EOF not detected?

Example 3 : Child talking to parent

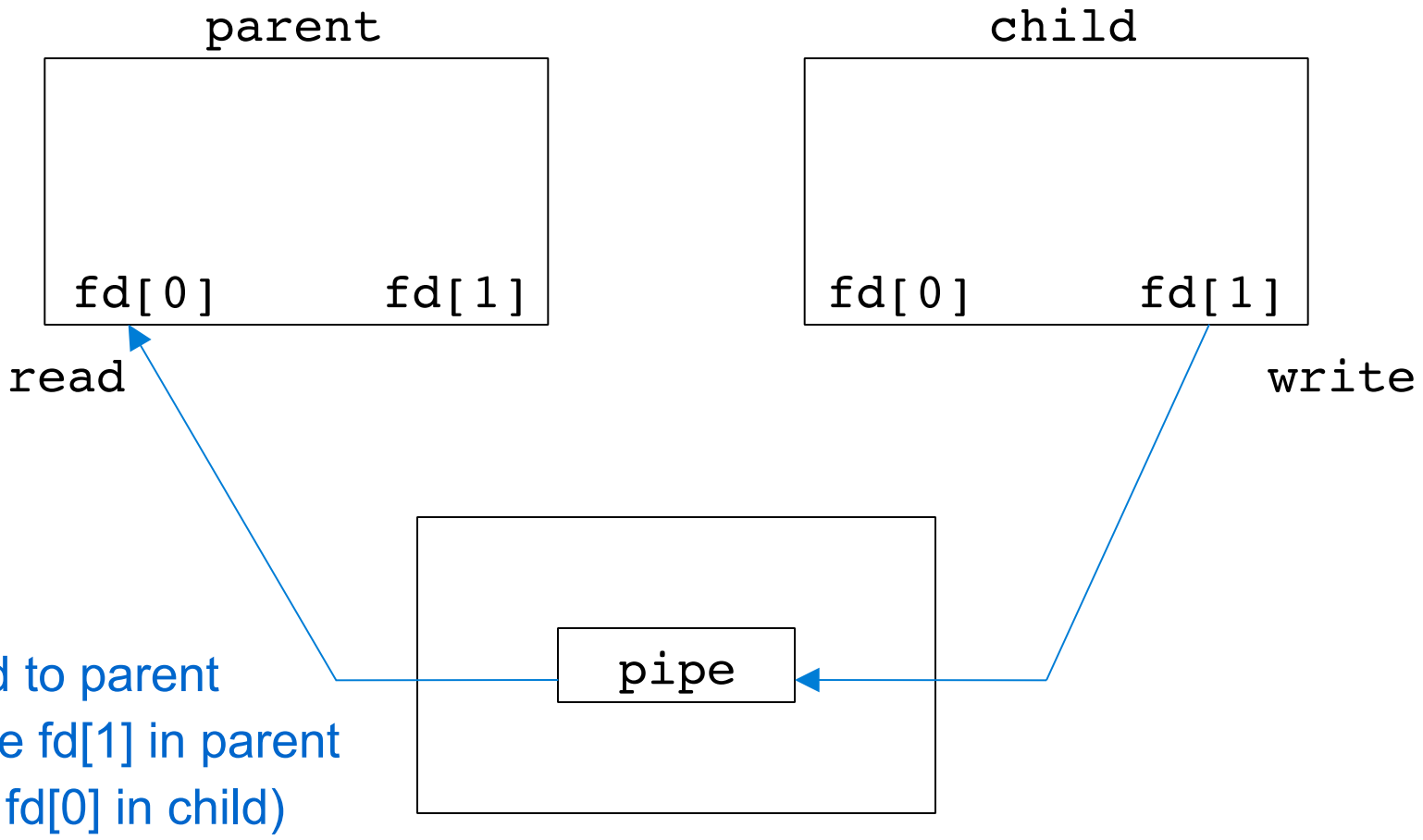
```
#define BUF_SIZE 4 //Small!
#define MSG_SIZE 13
char *msg = "hello, world\n";

int main(void) {
  char buf[BUF_SIZE];
  int p[2], nbytes;
  if(pipe(p) == -1) {
    perror("pipe");
    exit(1);
  }
  if (fork() == 0)
    write(p[1], msg, MSG_SIZE);
  else {
    close (p[1]);
    while ((nbytes = read(p[0], buf, BUF_SIZE)) > 0)
      write (STDOUT_FILENO, buf, nbytes);
  }
  return 0;
}
```

Parent needs to close its writing end.

Closing ends of pipes

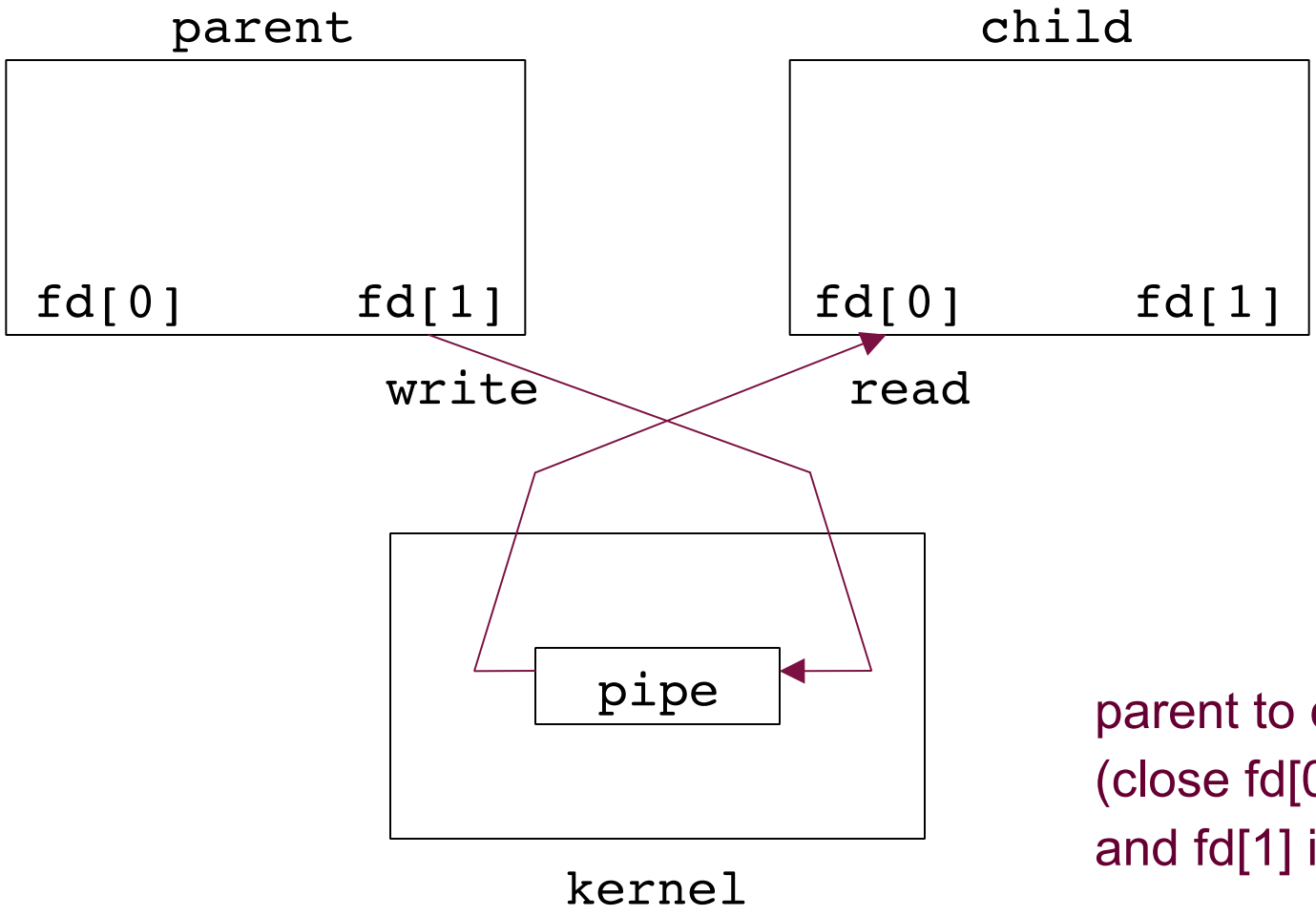
- In general, each process will read or write a pipe (not both)
- Close the end you are not using
  - Before reading: close p[1]
  - Before writing: close p[0]

Direction of data flow?



child to parent
close fd[1] in parent
and fd[0] in child)

Direction of data flow?



parent to child
(close fd[0] in parent
and fd[1] in child)

Summary: Pipes and File Descriptors

- A forked child inherits file descriptors from its parent
- pipe() creates an OS internal system buffer and two file descriptors, one for reading and one for writing.
- After the pipe call, the parent and child should close the file descriptors for the opposite direction.