

# Inter-process Communication

Pipes (Haviland – Ch. 7)

1

## Exchanging data between processes

- After `fork()` is called we end up with two independent processes.
- We cannot use variables to communicate between processes since they each have separate address spaces, and separate memory.
- One easy way to communicate is to use files.
  - process A writes to a file and process B reads from it.
- See `usefiles.c` example. We need to be pretty careful.

2

## Buffering

- **un-buffered** – output appears immediately
  - `stderr` is not buffered
- **line buffered** – output appears when a full line has been written.
  - `stdout` is line buffered when going to the screen
- **block buffered** – output appears when a buffer is filled or a buffer is flushed (on close or explicit flush).
  - normally output to a file is block buffered
  - `stdout` is block buffered when redirected to a file.

3

## File Objects and File Descriptors

- The `stdio` library provides `FILE` objects which handle buffering.
- Why buffering? Efficiency.
- `FILE` objects are built on top of file descriptors. A file descriptor is an index into a per-process table of open file descriptors.
- We will also use file descriptors for other communication such as pipes and sockets.

4

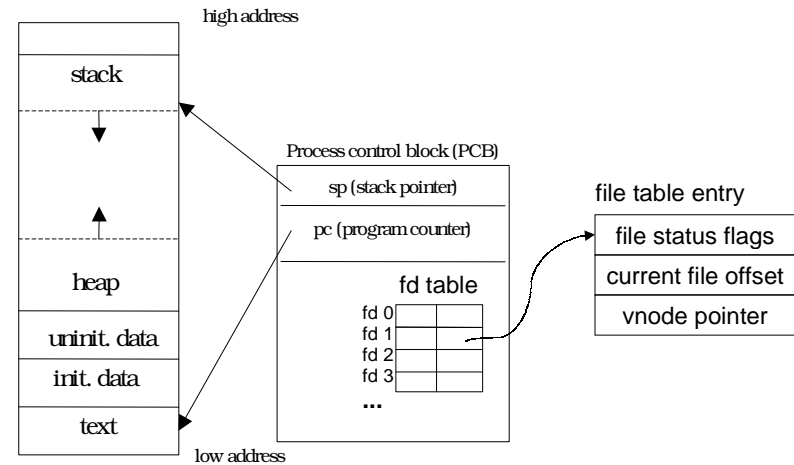
# File Descriptors

- Used by low-level I/O
  - `open()`, `close()`, `read()`, `write()`
- declared as an integer
  - `int fd;`
- A useful system call to convert a FILE object to a fd
 

```
int fileno( FILE *fp);
```
- Of course it is possible to assign a stream interface to a file descriptor

```
FILE *fdopen(int fd, const char *mode);
```

# Process state



# Producer/Consumer Problem

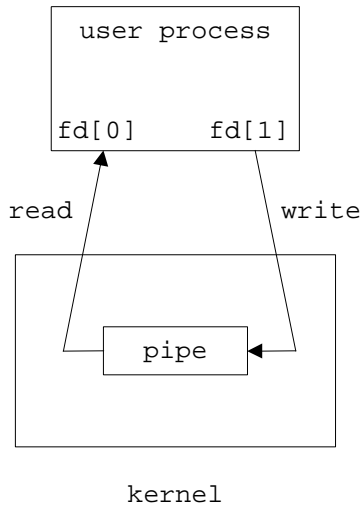
- Simple example: `who | wc -l`
- Both the writing process (`who`) and the reading process (`wc`) of a pipeline execute concurrently.
- A pipe is usually implemented as an internal OS buffer.
- It is a resource that is concurrently accessed by the reader and the writer, so it must be managed carefully.

# Producer/Consumer

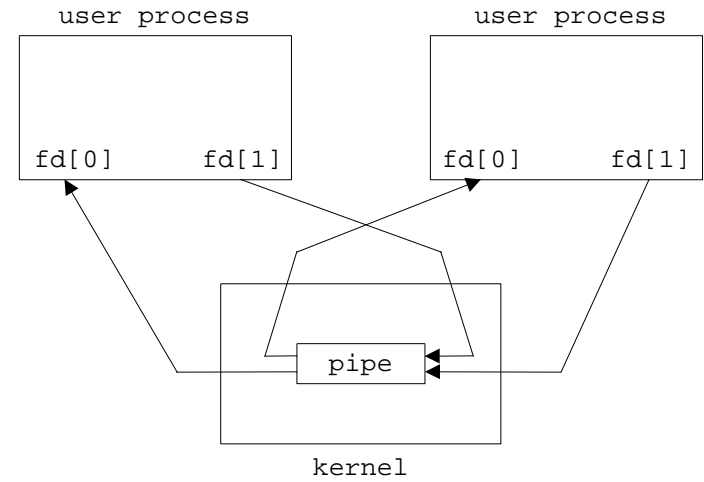
- **Consumer** blocks when buffer is empty
  - **Producer** blocks when buffer is full
  - They should **run independently** as far as buffer capacity and contents permit
  - They should never be updating the buffer at the same instant (otherwise **data integrity** cannot be guaranteed)
- Harder problem if there is more than one consumer and/or more than one producer.

```
int pipe(int fildes[2])
```

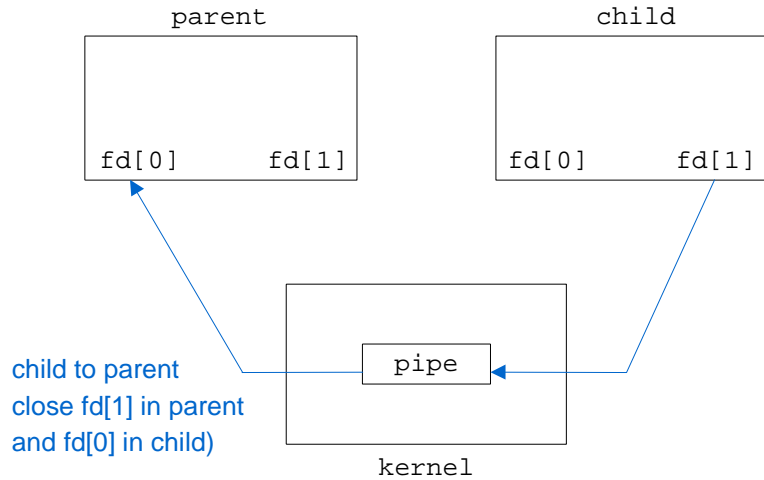
- half-duplex (one-way) communication



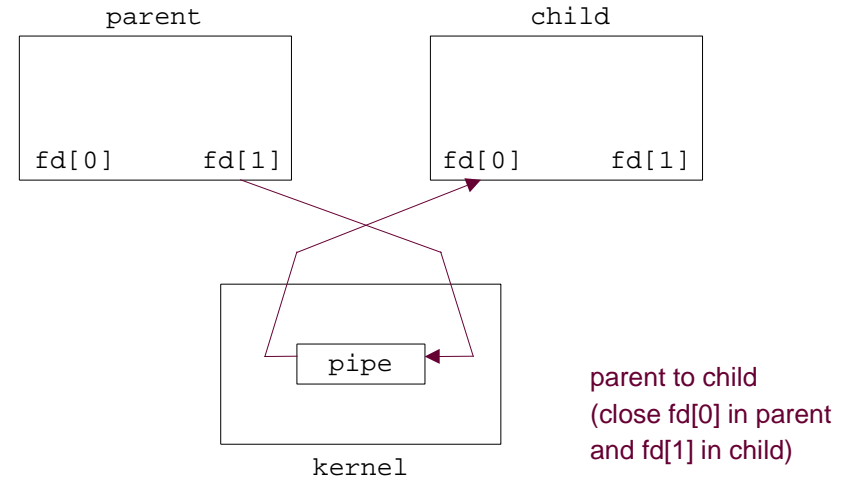
## What happens after fork?



## Direction of data flow?



## Direction of data flow?



# Pipes and File Descriptors

- A forked child inherits file descriptors from its parent
- pipe() creates an 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. Leaving them open does not permit full-duplex communication.

13

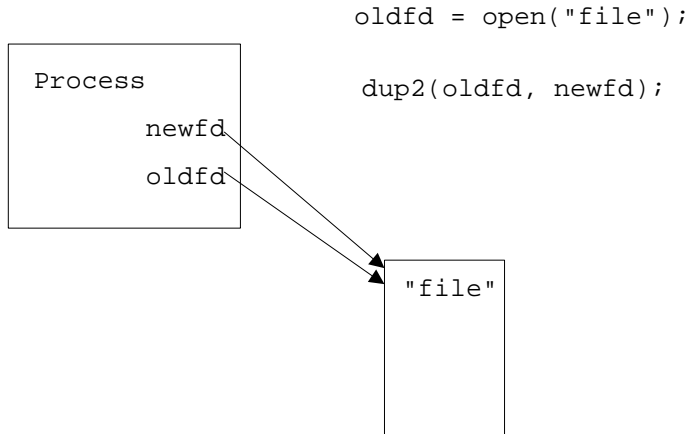
# dup2 ( )

- Often we want the stdout of one process to be connected to the stdin of another process.
- Set one FD to the value of another.

```
returnCode = dup2(oldFD, newFD);  
– newFD and oldFD now refer to the same file  
– if newFD is open, it is first automatically closed  
– Note that dup2 ( ) refer to fds not streams
```

14

# dup2 ( )



15

# pipe ( ) / dup2 ( ) example

```
/* equivalent to "sort < file1 | uniq" */  
int fd[2], pid;  
int filedes = open("file1", O_RDONLY);  
dup2(filedes, fileno(stdin));  
close(filedes);  
pipe(fd);
```

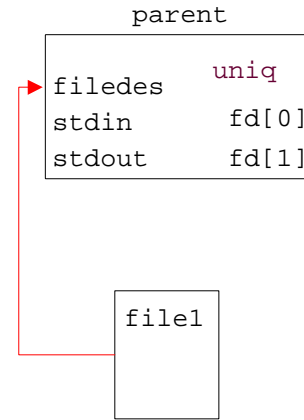
16

# pipe() / dup2() example

```
if((pid = fork()) == 0) { /* child */
    dup2(fd[1], fileno(stdout));
    close(fd[0]); close(fd[1]);
    execl("/usr/bin/sort", "sort", (char *) 0);
} else if(pid > 0) { /* parent */
    dup2(fd[0], fileno(stdin));
    close(fd[1]); close(fd[0]);
    execl("/usr/bin/uniq", "uniq", (char *) 0);
} else {
    perror("fork"); exit(1);
}
```

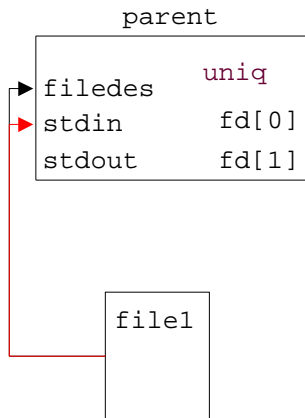
17

```
int filedes = open("file1", O_RDONLY);
```



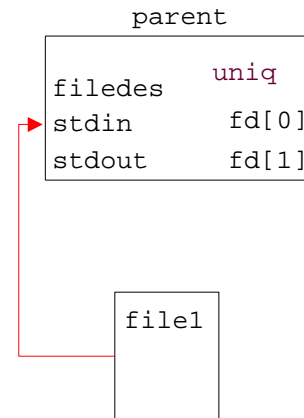
18

```
dup2(filedes, fileno(stdin));
```



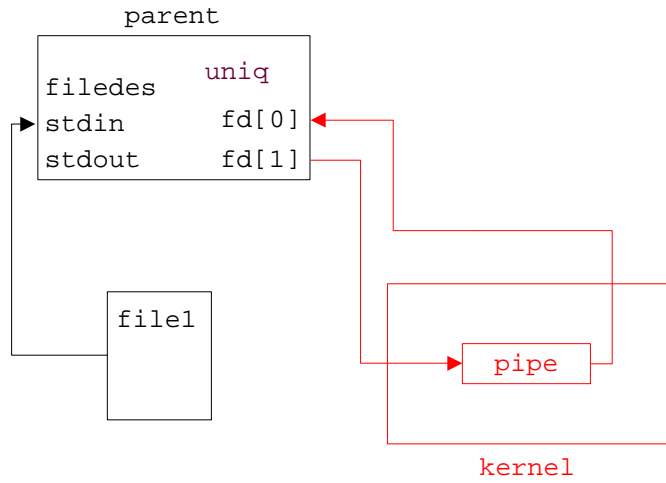
19

```
close(filedes);
```



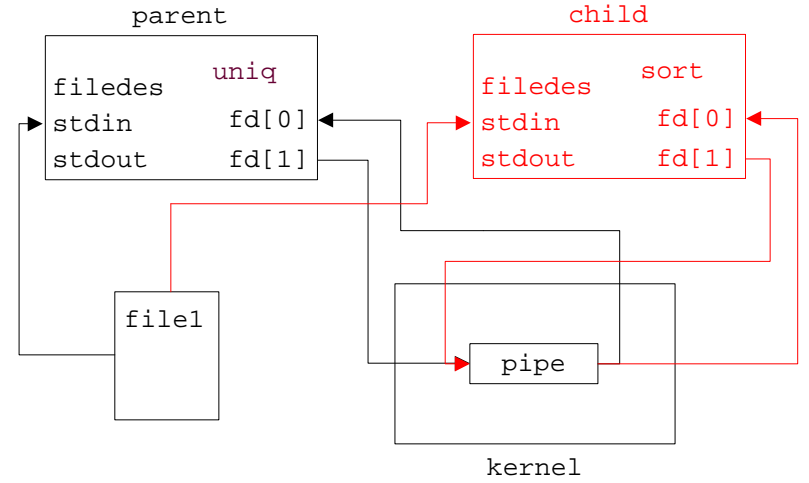
20

`pipe(fd);`



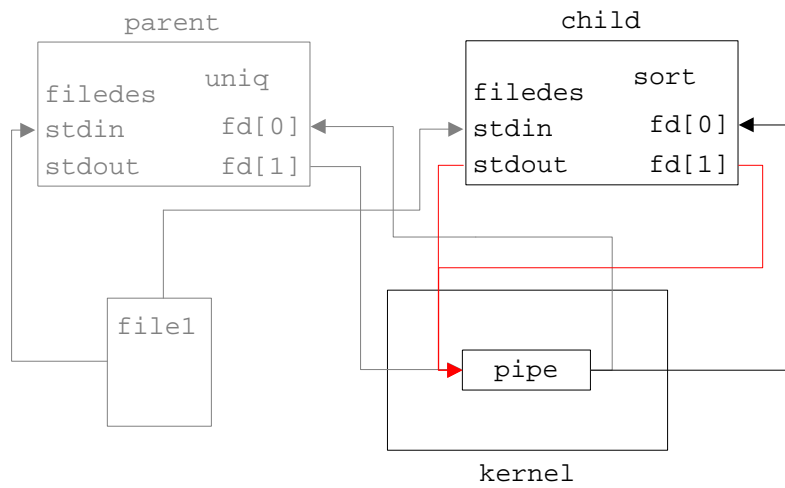
21

`fork();`



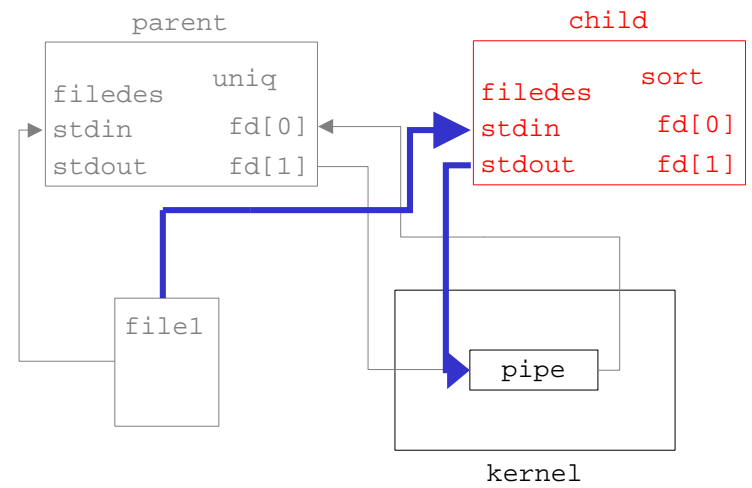
22

`dup2(fd[1], fileno(stdout));`



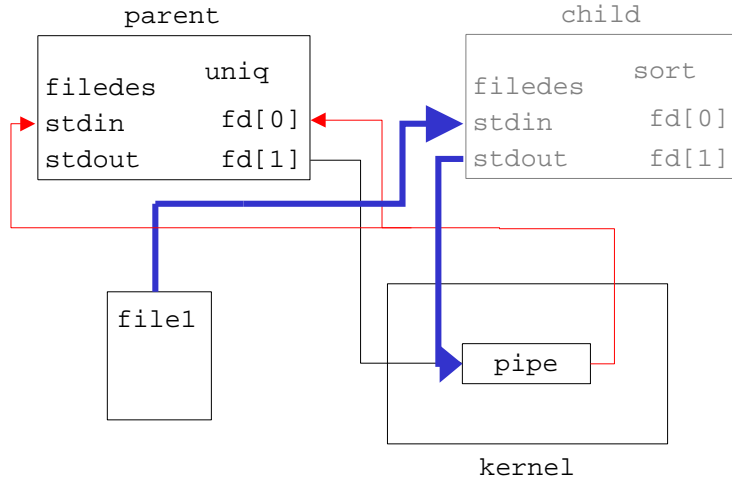
23

`close(fd[0]); close(fd[1]);`



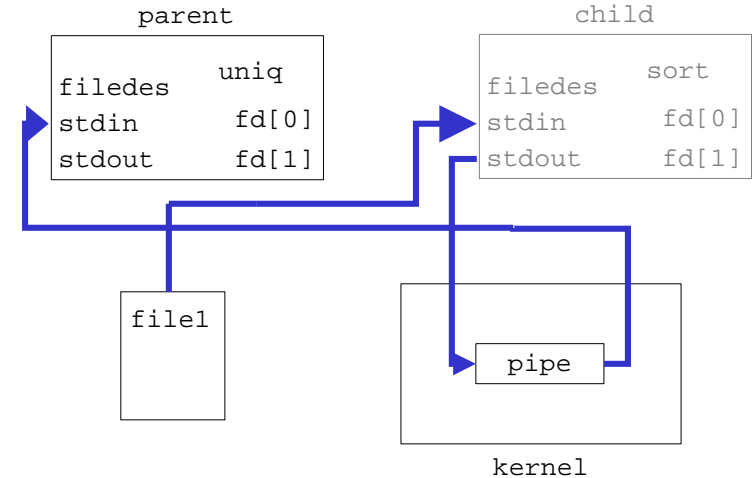
24

```
dup2(fd[0], fileno(stdin));
```



25

```
close(fd[1]); close(fd[0]);
```



26

## Reading and writing to a pipe

- A read on an empty pipe will block until there is something to read.
- A write on a full pipe will block until there is more space. (Pipes have a finite size.)
- Writing to a pipe that has been closed by the other end will result in a SIGPIPE or "Broken Pipe" message.
- Read will return 0 if the write end of the pipe is closed.

27

## popen() and pclose()

- popen() simplifies the sequence of:
  - generating a pipe
  - forking a child process
  - duplicating file descriptors
  - passing command execution via an exec()
- Usage:

```
FILE *popen( const char *command,  
            const char *type );
```
- Example:

```
FILE *pipeFP;  
pipeFP = popen("/usr/bin/ls *.c", "r");
```

28

## popen( )

"r"



"w"

