CSC209 Summer 2015 — Software Tools and Systems Programming

www.cdf.toronto.edu/~csc209h/summer/

Week 9 — July 9, 2015

Peter McCormick
pdm@cs.toronto.edu

Some materials courtesy of Karen Reid

Announcements

- Assignment 3 has been released
  - After today you will have seen everything you need to complete it!
- No tutorial tonight

Last Week Recap

- *Unix* mechanisms and abstractions
- System calls as an API for your programs to talk with the operating system
- Interacting with processes: the **`fork`**, **`wait`** and **`exec*`** system calls

Agenda

- Low-level File I/O
- Pipes
- Signals

Low-level File I/O

Kerrisk 2.5 and 4.1-7

Streams API

```
 FILE *fopen (const char *filename,
              const char *mode);
   int fclose(FILE *fp);
size_t fread (void *ptr,
              size_t size,
              size_t nitems,
              FILE *stream);
size_t fwrite(const void *ptr,
              size_t size,
              size_t nmemb,
              FILE *stream);
   int fseek (FILE *stream,
              long offset,
              int whence);
   int feof  (FILE *stream);
   int fgetc (FILE *stream);
```

… and more …

Streams API

- Specified in the C Standard Library
  - *Portable* across platforms, i.e. Windows, Mac OS X, Linux, BSD, etc.
- This API is vaguely *object-oriented* with a C flavour:
  - These functions are the *methods*
  - The opaque `FILE*` is the *object instance*

Streams API

- This API is built upon lower level, system call file I/O primitives

Unix File I/O API

```
int     open (const char *pathname,
              int flags,
              mode_t mode);
int     close(int fd);
ssize_t read (int fd,
              void *buf,
              size_t count);
ssize_t write(int fd,
              const void *buf,
              size_t count);
off_t   lseek(int fd,
              off_t offset,
              int whence);
```

… and more …

Unix File I/O API

```
int      open (const char *pathname,
               int flags,
               mode_t mode);
int      close(int fd);
ssize_t read (int fd,
               void *buf,
               size_t count);
ssize_t write(int fd,
               const void *buf,
               size_t count);
off_t    lseek(int fd,
               off_t offset,
               int whence);
```

… and more …

Unix File I/O API

- `fd` for *file descriptors*
    - Integer values representing currently open file handles
    - Analogous in use to a `FILE*`

`open(2)` — open/create a file

```
int open(const char *pathname,
         int flags,
         [mode_t mode]);
```

- Opens `pathname` as a file according to `flags`:
  - `flags & O_RDONLY`: read only
  - `flags & O_WRONLY`: write only
  - `flags & O_RDWR`: reading and writing
  - `flags & O_CREAT`: create `pathname` if it does not already exist (`mode` then specifies the default file mode permissions)
  - `flags & O_TRUNC`: if opening for writing and file already exists, truncate its size down to 0

`open(2)` — open/create a file

```
int open(const char *pathname,
         int flags,
         [mode_t mode]);
```

- `O_*` symbols are power-of-2 constants, so you use the *logical OR* (|) operator to combine more than one
- `mode` is only required if `flags & O_CREAT`
- `open` will return `-1` if an error occurred, otherwise returns a non-zero file descriptor

`open(2)` — open/create a file

`fopen()` ***`mode`***	**`open()`** ***`flags`***	**Effect**
`“r”`	`O_RDONLY`	Reading from beginning
`“r+”`	`O_RDWR`	Reading & writing from beginning
`“w”`	`O_WRONLY | O_CREAT | O_TRUNC`	Create/truncate for writing
`“w+”`	`O_RDWR | O_CREAT | O_TRUNC`	Create/truncate for reading & writing
`“a”`	`O_WRONLY | O_APPEND`	Writing (append) from end of file
`“a+"`	`O_RDWR | O_APPEND`	Reading and appending to end of file

Standard in/out/error

Name	`fd` integer	`fd` symbolic	`FILE*`	Mode
Standard Input	**0**	`STDIN_FILENO`	`stdin`	Read only
Standard Output	**1**	`STDOUT_FILENO`	`stdout`	Write only
Standard Error	**2**	`STDERR_FILENO`	`stderr`	Write only

stdiosfds.c

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

Encapsulate a *file descriptor* inside of a **`FILE*`** stream.

You can then use functions like **`fprintf(fp, …)`** for easier text printing.

`read(2)` and `write(2)`

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

- Similar to `fread` and `fwrite`, except with a more straightforward `count` argument

`read(2)` and `write(2)`

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

- Returns:
  - **-1** on error (`ssize_t` is a *signed* variant of `size_t`, so it can actually represent a *negative* value)
  - **0** when EOF reached (when reading)
  - A non-zero value for the number of bytes actually read or written (beware that this *may not* equal **`count`**…)

read.c

write-stdout.c

write.c

fdcat.c

Facts about File Descriptors

- `fd`'s are *per-process*
    - My **`fd 1 stdout`** will be a different destination than your **`fd 1 stdout`**…
- … but open file descriptors are *preserved* across a **`fork()`** system call
- … *and* they are still linked together to the same underlying kernel resource

forkcat.c

Idea: Could we make use of the sharing of `fd`'s across a `fork()` to let a parent and child process communicate?

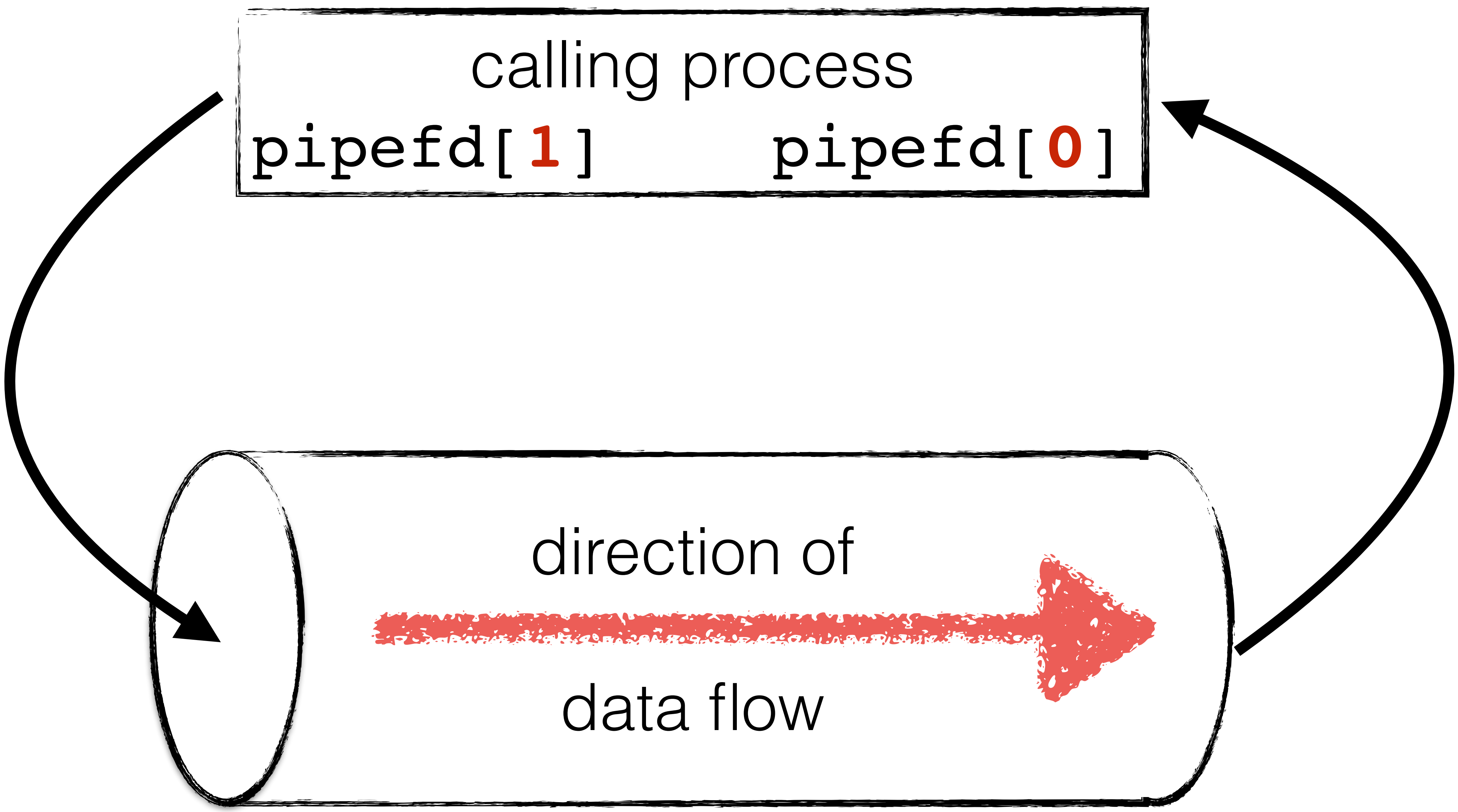
Pipes

Kerrisk 44.1-4

`pipe(2)` — create pipe

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

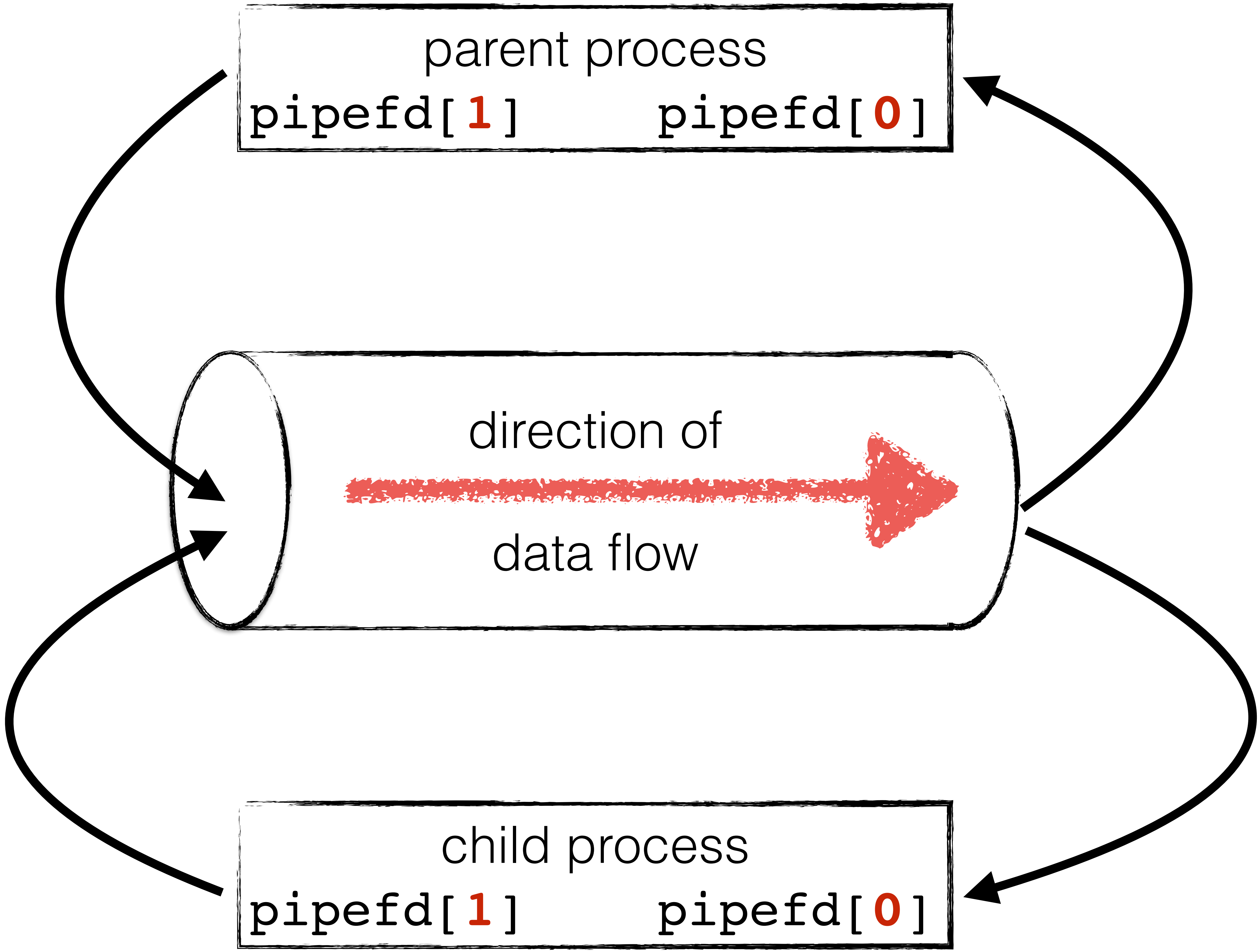
- Creates a *unidirectional* data channel (aka a *pipe*) by returning a pair of connected file descriptors
  - **`pipefd[0]`** is an FD of the *read* end
  - **`pipefd[1]`** is an FD of the *write* end
- Data written to the one end will be read out the other



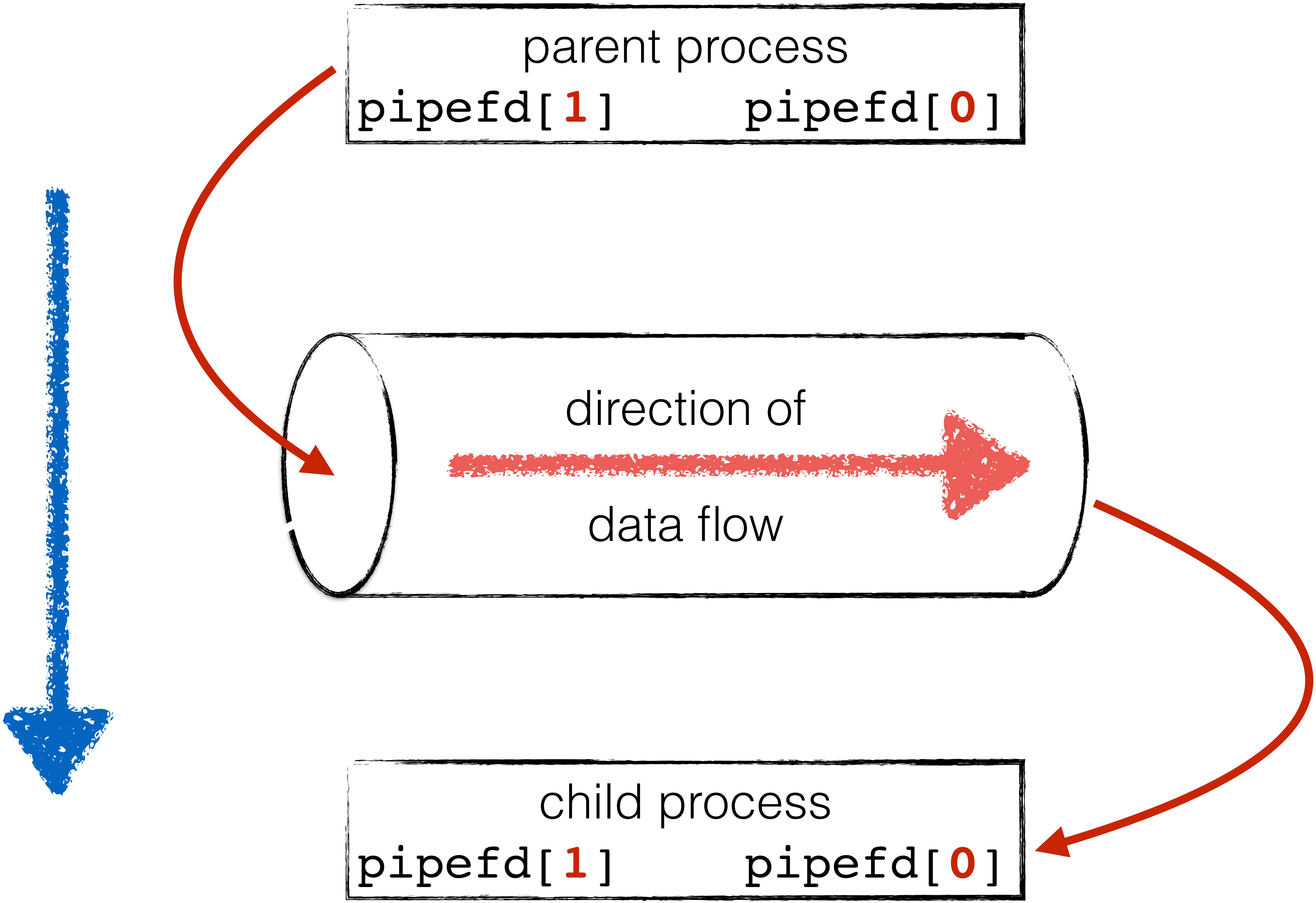
Kerrisk figure 44-2

pipe.c

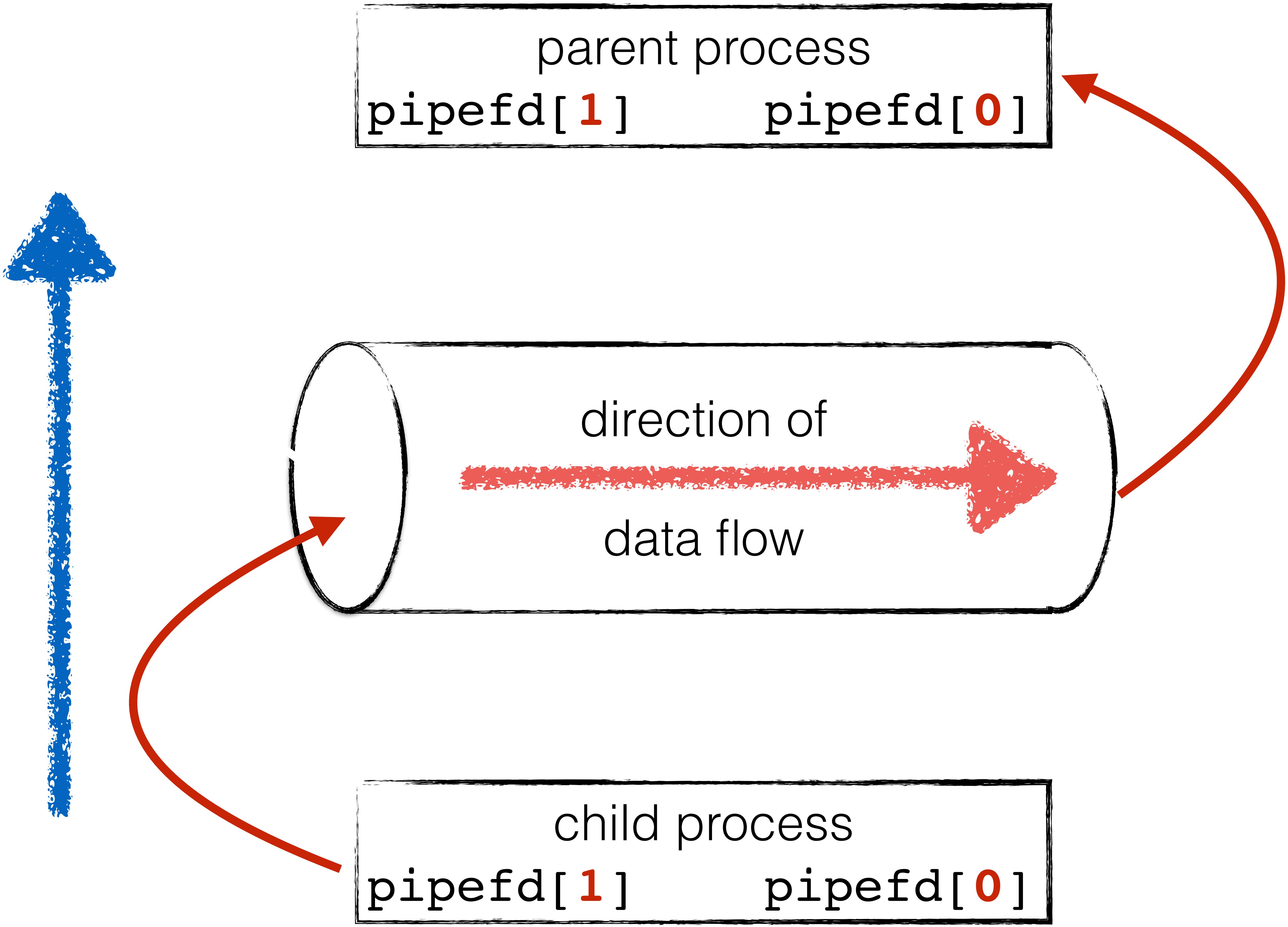
Pipes and `fork()`



Kerrisk figure 44-3a: After fork



Kerrisk figure 44-3b: After closing unused descriptors



Kerrisk figure 44-3b: After closing unused descriptors

Bidirectional data flows requires *two* pipes (one for each direction) pipeforkcat.c

`dup2` — duplicate a FD

```
int dup2(int oldfd, int newfd);
```

From the manpage: *"**`dup2()`** makes **`newfd`** be a copy of **`oldfd`**, closing **`newfd`** first if necessary…"*

dup2 — duplicate a FD

```
// Before: oldfd and newfd are separate FD's

dup2(oldfd, newfd);

// After: previous newfd is now closed
// After: reads/writes to newfd are now
//          reads/writes to oldfd

// The following are the same
write(newfd, buf, buf_size);
write(oldfd, buf, buf_size);
```

dup2 — duplicate a FD

```
// Before: fd and STDOUT_FILENO are separate FD’s

dup2(fd, STDOUT_FILENO);

// After: previous STDOUT_FILENO is now closed
// After: reads/writes to STDOUT_FILENO are now
//          reads/writes to fd

// The following are the same
write(STDOUT_FILENO, buf, buf_size);
write(fd, buf, buf_size);

// “Hello World\n” will be written to fd
fprintf(stdout, “Hello World\n”);
```

dup2.c

Signals

Kerrisk 2.11 and 20
21 (for interest)

Signals

- A lot of software development assumes a *synchronous* model of execution:
  - Function calls (they only return once the work is done)
  - Sequential line-by-line execution of programs
  - Request → Response

Signals

- How do you write systems that can handle unexpected/unpredictable events *asynchronously*?
  - Floating point computation error
  - Death of a child process
  - Interval timer expired (alarm clock)
  - Ctrl-C (`^C`) — request to terminate process
  - Ctrl-Z (`^Z`) — request to suspend process
  - Hardware peripheral requires attention

Signals

- Such events are called *interrupts*
- Classic Unix kernel design supports a form of software interrupt called *signals* that are used to notify processes of important events
- The kernel can signal your process if something bad has happened to it
- Processes can send signals to other processes too as a form of *inter-process communication* (IPC)

Perhaps you've met…

- `SIGINT`: Ctrl-C (`^C`) to terminate a process
- `SIGSTOP`: Ctrl-Z (`^Z`) to suspend process
- `SIGSEGV`: Segmentation fault
- `SIGPIPE`: Writing to a pipe whose read end has been closed
- `SIGCHLD`: sent by a child process to its parent when it terminates (in order for the parent to collect its exit status)

sigsegv.c

sigpipe.c

Sending Signals — `kill(1)`

usage: `kill [-SIGNAL] pid [pid]…`

- If no signal is specified, `kill` sends the `TERM` signal to the process.
- Signal can be specified by the number or name (without the *SIG* prefix)
- Examples:
  - `kill -QUIT 8883`
  - `kill -STOP 78911`
  - `kill -9 76433` (9 == `KILL`)

Also `pgrep(1)` and `pkill(1)`

(a combination of `ps`, `grep` and `kill`)

Software Interrupts

- `/usr/include/sys/signal.h` lists the signal types on CDF.
- "`man 7 signal`" gives some description of various signals
  - `SIGTERM, SIGABRT, SIGKILL`
  - `SIGSEGV, SIGBUS`
  - `SIGSTOP, SIGCONT`
  - `SIGCHLD`
  - `SIGPIPE`
  - `SIGUSR1, SIGUSR2`

Signal Handlers

- Your code can programmatically catch and deal with signals when they arrive by installing a special function called a *signal handler*

- The signal handler function can execute some C statements and exit in 3 different ways:

  1. Return control to the place in the program which was executing when the signal occurred.

  2. Return control to some *other* point in the program.

  3. Terminate the program by calling `exit`.

Default Actions

- Each signal has a default action:
  - Terminate *(shutdown the process)*
  - Stop *(pause the process)*
  - Ignore *(disregard the signal entirely)*
- The default action can be changed for *most* signal types using the **`sigaction()`** function
  - The exceptions are `SIGKILL` and `SIGSTOP` (they will always shutdown or pause your process, respectively)

Signal Table

- For each process the kernel maintains a table of actions associated to each type of signal. Example:

Signal	Default Action	Comment
`SIGINT`	Terminate	Interrupt from keyboard
`SIGSEGV`	Terminate (dump core)	Invalid memory reference
`SIGKILL`	Terminate *(cannot ignore)*	Kill
`SIGCHLD`	Ignore	Child stopped or terminated
`SIGSTOP`	Stop *(cannot ignore)*	Stop process
`SIGCONT`		Continued if stopped

`sigaction` — install a signal handler

```
int sigaction(int signum,
              const struct sigaction *act,
              struct sigaction *oldact);
```

- Installs a new handler (specified by **`act`**) for signal **`signum`**
- If non-**`NULL`**, with the old handler specification is copied to **`oldact`**
- Don't forget to *`#include <signal.h>`* to get necessary definitions!

`struct sigaction`

```
struct sigaction {
  /* SIG_DFL, SIG_IGN, or pointer to function */
  void (*sa_handler)(int);

  /* Signals to block during handler execution */
  sigset_t sa_mask;

  /* Flags and options */
  int sa_flags;
};
```

Additional extensions exist to specify different kinds of handlers (see the description of the `sa_sigaction` field from the `sigaction(2)` manpage.)

signalsoak.c

`kill(2)` — send signal to a process

```
int kill(pid_t pid, int sig);
```

- Sends signal **`sig`** to process **`pid`**
- Misleading name: used for more than just sending **`SIGKILL`** to processes!

`kill(2)` — send signal to a process

```
int kill(pid_t pid, int sig);
```

- Many applications:
  - Kill errant processes
  - Temporarily suspend execution of a process
  - Make a process aware of the passage of time
  - Synchronize the actions of processes.

Timer Signals

- Three interval timers are maintained for each process:
  - **`SIGALRM`** (real-time alarm, like a stopwatch)
  - **`SIGVTALRM`** (virtual-time alarm, measuring CPU time)
  - **`SIGPROF`** (used for profilers)

Timer Signals

- Useful functions to set and get timer info:
  - **`sleep()`** – cause calling process to suspend
  - **`usleep()`** – like **`sleep()`** but at a finer granularity (μs vs seconds)
  - **`alarm()`** – sets **`SIGALRM`**
  - **`pause()`** – suspend until next signal arrives
  - **`setitimer()`**, **`getitimer()`**
- **`sleep()`** and **`usleep()`** are *interruptible* by other signals.

Blocking Signals

- Signals can arrive at *any* time (i.e. in the middle of what your code is doing!)

- To temporarily prevent a signal from being delivered, we *block* it.

    - The signal is held until the process unblocks the signal

- When a process *ignores* a signal, it is thrown away and is never handled

Groups of Signals

- Signal masks are used to store the set of signals that are currently blocked.
- Operations on *sets* of signals:
  - `int sigemptyset(sigset_t *set);`
  - `int sigfillset(sigset_t *set);`
  - `int sigaddset(sigset_t *set, int signo);`
  - `int sigdelset(sigset_t *set, int signo);`
  - `int sigismember(const sigset_t *set, int signo);`

`sigprocmask` - examine/change blocked signals

```
int sigprocmask(int how,
                const sigset_t *set,
                sigset_t *oldset);
```

- **`how`** indicates in what way the signal will be modified
  - **`SIG_BLOCK`**: add to those currently blocked
  - **`SIG_UNBLOCK`**: delete from those currently blocked
  - **`SIG_SETMASK`**: set the collection of signals being blocked
- **`set`** points to the set of signals to be used for modifying the mask
- **`oldset`** (if non-**`NULL`**) will have the previous value of the signal mask copied to it

Suggested Exercises

https://github.com/pdmccormick/csc209-summer-2015/blob/master/lectures/week9/README.md

Next Week

- Back to regular Tuesday office hour schedule
- Lecture: *Networking programming with sockets*