CS 111
Operating Systems Principles
Discussion 1E Week 2

Tengyu Liu

Any questions from last week?

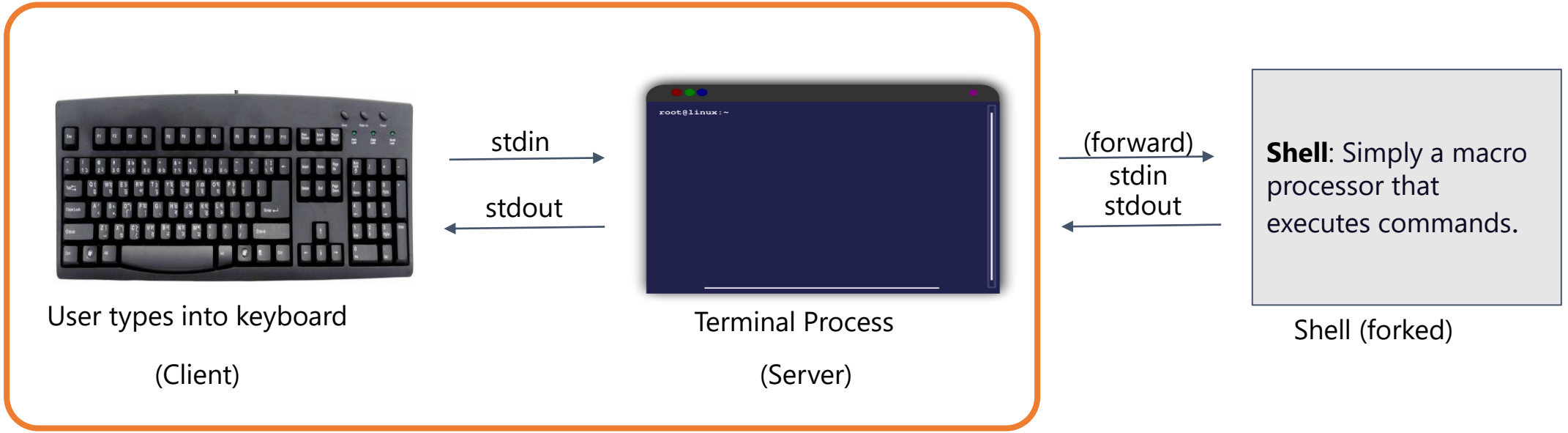
Table of Contents

Project 1

- Build a multi-process telnet-like client and server
  - Character-at-a-time, full duplex terminal I/O
  - Polled I/O and passing input and output between two processes
  - Support a `--shell=`*`program`* argument
  - Shutdown upon receiving EOF (`^D`)
- What is telnet?
  - Telnet is a protocol that allows you to **connect to remote computers** (called hosts) over a TCP/IP network (such as the internet).
  - Using telnet client software on your computer, you can make a connection to a telnet server (that is, the remote host). Once your telnet client establishes a connection to the remote host, your client becomes a virtual terminal, allowing you to communicate with the remote host from your computer
  - Telnet has Full Duplex Communication
  - We are going to build a multi process telnet-like client and server

The big picture

The server program will connect with the client, receive the client's commands and send them to the shell, and will "serve" the client the outputs of those commands.



Character-at-a-time, full duplex terminal I/O

The project spec says:

“put the keyboard (the file open on file descriptor 0) into character-at-a-time, no-echo mode (also known as non-canonical input mode with no echo). It is suggested that you get the current terminal modes, save them for restoration, and then make a copy with only the following changes:

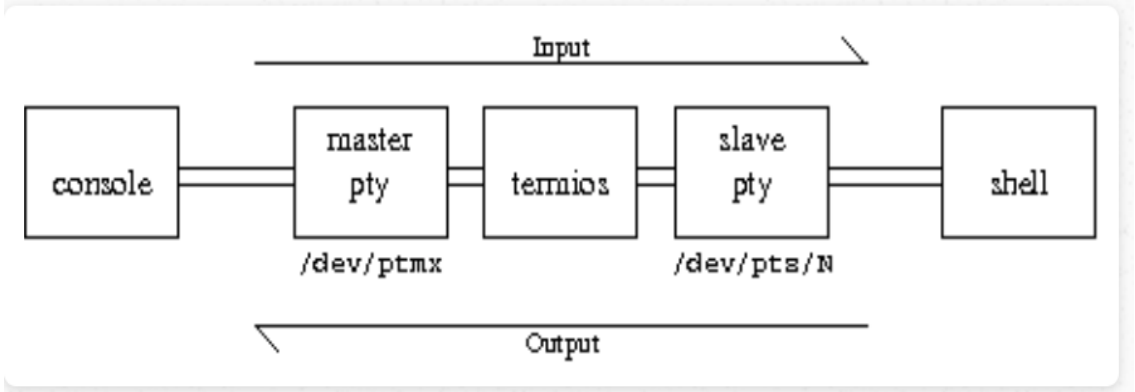
```
c_iflag = ISTRIP;    /* only lower 7 bits          */

c_oflag = 0;                    /* no processing  */

c_lflag = 0;                    /* no processing  */
```

and these changes should be made with the TCSANOW option”

Let’s talk about termios(tcsetattr, tcgetattr), echo, non-canonical, c_iflag/c_oflag/c_lflag.

What is termios?

- termios acts on streams between console and shell



- Functions
  - Echo
  - Line buffering
  - Line editing
  - Newline translation
  - Signal generation
  - …

Canonical vs Non-canonical input

Canonical Input

- The entire line is gathered and edited up until the end of line character — return — is pressed.
- Then, the whole line is made available to waiting programs.
- Example: bash

Non-canonical input

- You press a character, and it is immediately available to the program.
- You aren't held up until you hit return. The system does no editing of the characters; they are made available to the program as soon as they are typed.
- Example: text editor
- Is the input buffer bypassed completely?
  - With non-canonical input, the input buffer is still used; if there is no program with a read() call waiting for input from the terminal, the characters are held in the input buffer. What doesn't happen is any editing of the input buffer.

`termios` in greater detail

The primary programmatic interface to `termios` is:

- `struct termios`, and
- Two functions which retrieve & set the struct termios associated with a given terminal device

```
int tcgetattr(int fd, struct termios *termios_p);
int tcsetattr(int fd, int optional_actions,
              const struct termios *termios_p);
```

So what's inside struct termios?

- POSIX specifies that this structure contains at least the following fields:

```
tcflag_t c_iflag;      /* input modes */
tcflag_t c_oflag;      /* output modes */
tcflag_t c_cflag;      /* control modes */
tcflag_t c_lflag;      /* local modes */
cc_t     c_cc[NCCS];   /* control chars */
```

- Each "flag" field contains a number of flags (implemented as a bitmask) that can be individually enabled or disabled (such as "tattr.c_lflag &= ~(ICANON|ECHO);" disables ICANON and ECHO)
- c_iflag and c_oflag contain flags that affect the processing of input and output, respectively
- c_cflag we will mostly ignore, as it contains settings that relate to the control of modems and serial lines that are mostly irrelevant these days
- c_lflag is perhaps the most interesting of the flag values. It contains flags that control the broad-scale behavior of the tty.

Some local modes inside of c_lflag,c_iflag

- `c_lflag`
  - `ICANON` - “canonical” mode
  - `ECHO` in `c_lflag` controls whether input is immediately re-echoed as output.
  - Example: When password prompts for your password, your terminal is in canonical mode, but `ECHO` is disabled.
- `c_iflag`
  - `ISTRIP` is a `c_iflag` flag constant which strips off eighth bit

Storing current terminal mode and restoring

- In the spec, you are asked to put the keyboard into
  - Character-at-a-time (non-canonical input mode)
  - No-echo mode
- Do not create struct termios from scratch
  - Copy existing one, and make the following changes

```
c_iflag = ISTRIP;       /* only lower 7 bits    */
c_oflag = 0;            /* no processing        */
c_lflag = 0;            /* no processing        */
```

  - Make sure you save a backup for restoring terminal behavior.

Storing current terminal mode and restoring

- `tcgetattr`
  - Header file to include: `#include <termios.h>`
  - Function Signature:
    - `int tcgetattr(int fildes, struct termios *termios_p);`
- `tcsetattr`
  - Header file to include: `#include <termios.h>`
  - Function Signature:
    - `int tcsetattr(int fildes, int optional_actions, const struct termios *termios_p);`

tcgetattr

- `int tcgetattr(int fildes, struct termios *termios_p);`
- The `tcgetattr()` function shall get the parameters associated with the terminal referred to by `fildes` and store them in the `termios` structure referenced by `termios_p`.
- The `fildes` argument is an open file descriptor associated with a terminal.
- The `termios_p` argument is a pointer to a termios structure.
- Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and `errno` set to indicate the error.

tcsetattr

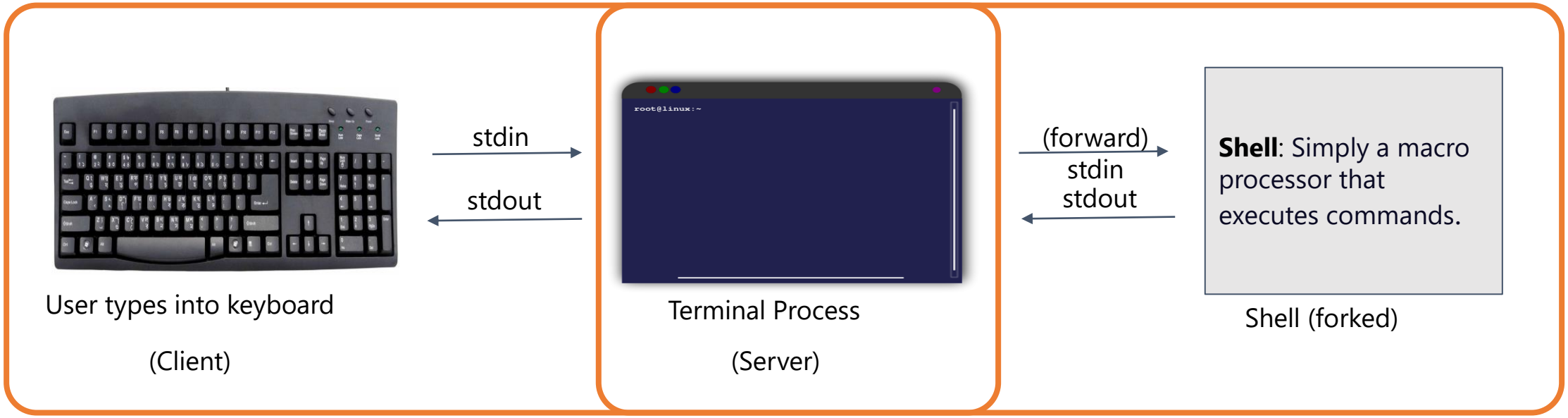
- `int tcsetattr(int fildes, int optional_actions, const struct termios *termios_p);`
- The `tcsetattr()` function sets the parameters associated with the terminal referred to by fildes from termios, according to the following requested action:
  - `TCSANOW` : The change occurs immediately
  - `TCSAFLUSH` : The change occurs after all output written to the file descriptor has been transmitted, and all input so far received but not read is discarded before the change is made.
- The `tcsetattr()` function returns successfully if it was able to perform any of the requested actions, even if some of the requested actions could not be performed.
- If no part of the request can be completed, `tcsetattr()` returns -1 and sets `errno` to `EINVAL`.
- Upon successful completion, 0 shall be returned.

If you messed up…

- Your terminal window is now gibberish.
- Solution
  - In your terminal, type: `stty sane`
  - Even if you can't see what you are typing, `stty sane` restores your terminal settings. You may want to hit Enter a few times first, to make sure you don't have anything else on your input line before you start typing the command.

The big picture

The server program will connect with the client, receive the client's commands and send them to the shell, and will "serve" the client the outputs of those commands.



Passing input and output between two processes

The project spec says:

"**`fork`** to create a new process, and then `exec` the specified program (with no arguments other than its name), whose standard input is a `pipe` from the terminal process, and whose standard output and standard error are (dups of) a pipe to the terminal process. (You will need two pipes, one for each direction of communication, as pipes are unidirectional.)"

Let's talk about fork, exec, and pipe.

fork

- Fork system call is used for creating a child process, which runs concurrently with the process that makes the `fork()` call (parent process).
- After a new child process is created, both processes will execute the next instruction following the `fork()` system call.
- A child process creates a copy of the pc(program counter), CPU registers, open files which the parent process uses.
- It takes no parameters and returns an integer value.

```
int fork();
```

`fork`

- Below are different values returned by `fork()`.
  - Negative Value: creation of a child process was unsuccessful.
  - Zero: Returned to the newly created child process.
  - Positive value: Returned to parent or caller. The value contains process ID of newly created child process

fork

- Below are different values returned by fork().
  - Negative Value: creation of a child process was unsuccessful.
  - Zero: Returned to the newly created child process.
  - Positive value: Returned to parent or caller. The value contains process ID of newly created child process

```
int main() {
    // ...
    int pid = fork();
    if (pid < 0) {
        fprintf("fork failed");
    } else if (pid == 0) {
        fprintf("this is child");
    } else {
        fprintf("this is parent");
    }
}
```

fork

Parent process

```
int main() {
    // ...
    int pid = fork();
    if (pid < 0) {
        fprintf("fork failed");
    } else if (pid == 0) {
        fprintf("this is child");
    } else {
        fprintf("this is parent");
    }
}
```

fork

Parent process

```
int main() {
    // ...
    int pid = fork();
    if (pid < 0) {
        fprintf("fork failed");
    } else if (pid == 0) {
        fprintf("this is child");
    } else {
        fprintf("this is parent");
    }
}
```

fork

Parent process

```
int main() {
    // ...
    int pid = fork();
    if (pid < 0) {
        fprintf("fork failed");
    } else if (pid == 0) {
        fprintf("this is child");
    } else {
        fprintf("this is parent");
    }
}
```

Child process

```
int main() {
    // ...
    int pid = fork();
    if (pid < 0) {
        fprintf("fork failed");
    } else if (pid == 0) {
        fprintf("this is child");
    } else {
        fprintf("this is parent");
    }
}
```

fork

Parent process

```
int main() {
    // ...
    int pid = fork();          // pid: ?
    if (pid < 0) {
        fprintf("fork failed");
    } else if (pid == 0) {
        fprintf("this is child");
    } else {
        fprintf("this is parent");
    }
}
```

Child process

```
int main() {
    // ...
    int pid = fork();          // pid: ?
    if (pid < 0) {
        fprintf("fork failed");
    } else if (pid == 0) {
        fprintf("this is child");
    } else {
        fprintf("this is parent");
    }
}
```

fork

Parent process

```
int main() {
    // ...
    int pid = fork();          // pid: 1
    if (pid < 0) {
        fprintf("fork failed");
    } else if (pid == 0) {
        fprintf("this is child");
    } else {
        fprintf("this is parent");
    }
}
```

Child process

```
int main() {
    // ...
    int pid = fork();          // pid: 0
    if (pid < 0) {
        fprintf("fork failed");
    } else if (pid == 0) {
        fprintf("this is child");
    } else {
        fprintf("this is parent");
    }
}
```

fork

Parent process

```
int main() {
    // ...
    int pid = fork();          // pid: 1
    if (pid < 0) {
        fprintf("fork failed");
    } else if (pid == 0) {
        fprintf("this is child");
    } else {
        fprintf("this is parent");
    }
}
```

Child process

```
int main() {
    // ...
    int pid = fork();          // pid: 0
    if (pid < 0) {
        fprintf("fork failed");
    } else if (pid == 0) {
        fprintf("this is child");
    } else {
        fprintf("this is parent");
    }
}
```

fork

Parent process

```
int main() {
    // ...
    int pid = fork();          // pid: 1
    if (pid < 0) {
        fprintf("fork failed");
    } else if (pid == 0) {
        fprintf("this is child");
    } else {
        fprintf("this is parent");
    }
}
```

Child process

```
int main() {
    // ...
    int pid = fork();          // pid: 0
    if (pid < 0) {
        fprintf("fork failed");
    } else if (pid == 0) {
        fprintf("this is child");
    } else {
        fprintf("this is parent");
    }
}
```

Passing input and output between two processes

The project spec says:

"`fork` to create a new process, and then **`exec`** the specified program (with no arguments other than its name), whose standard input is a `pipe` from the terminal process, and whose standard output and standard error are (dups of) a pipe to the terminal process. (You will need two pipes, one for each direction of communication, as pipes are unidirectional.)"

Let's talk about fork, exec, and pipe.

exec

- Loading a new program in a calling process
- `int execlp(const char *path, const char *arg, …)`

exec

- Loading a new program in a calling process
- `int execlp(const char *path, const char *arg, …)`
  - `On error, return -1`

exec

- Loading a new program in a calling process
- `int execlp(const char *path, const char *arg, …)`
  - `Executable`

exec

- Loading a new program in a calling process
- `int execlp(const char *path, const char *arg, …)`
  - Pointers to arguments
  - Includes the path to executable
  - List is terminated by a NULL pointer

exec

- Loading a new program in a calling process
- `int execlp(const char *path, const char *arg, …)`
- For Project 1A:
  - `execlp(program, program, (char*)NULL);`

Passing input and output between two processes

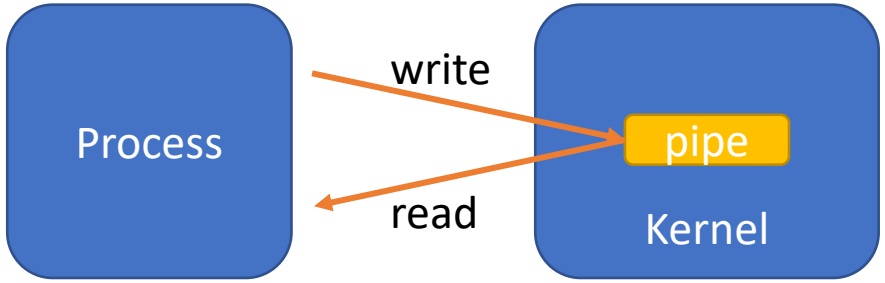
The project spec says:

"`fork` to create a new process, and then `exec` the specified program (with no arguments other than its name), whose standard input is a **`pipe`** from the terminal process, and whose standard output and standard error are (dups of) a **pipe** to the terminal process. (You will need two pipes, one for each direction of communication, as pipes are unidirectional.)"

Let's talk about fork, exec, and pipe.

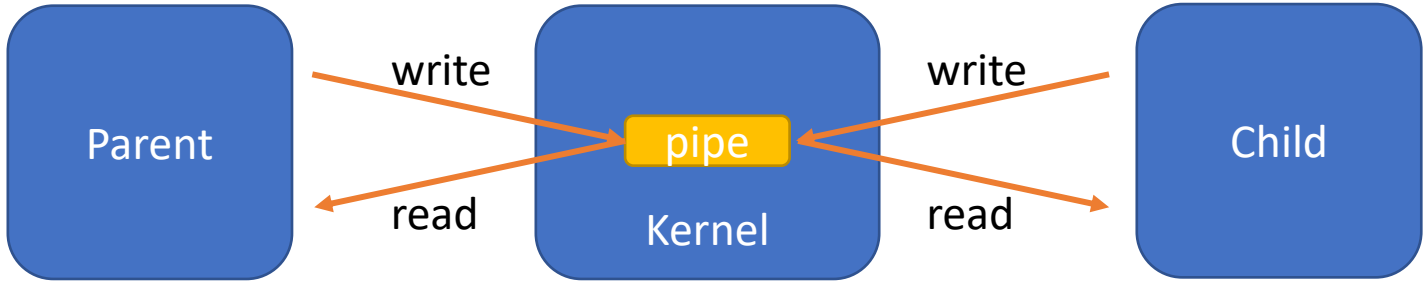
`pipe`

- A `pipe` is a method of connecting the output of one process to the input of another
- When a process creates a pipe, the kernel sets up two file descriptors to the pipe, one for input (write) and one for output (read)
- At this point, this is fairly useless.



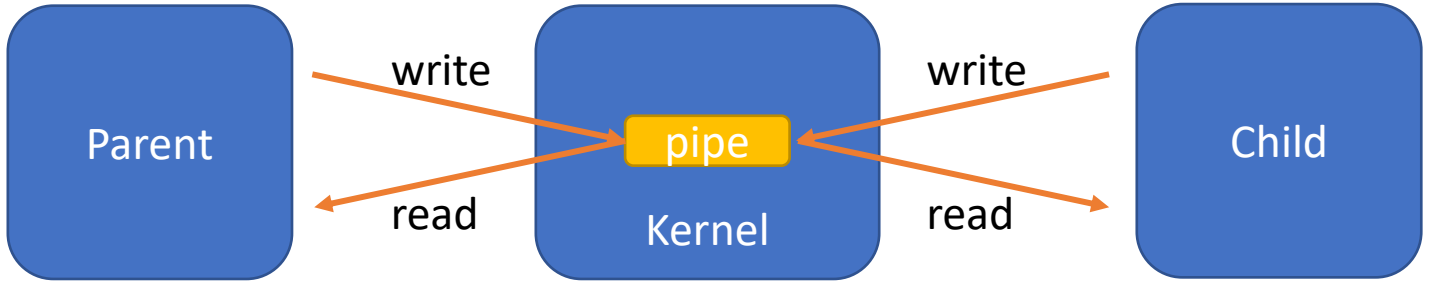
`pipe`

- Then, the creating process typically `fork`s a child process. Since the child process inherits any file descriptor from the parent, we now have the basis for inter-process communication.



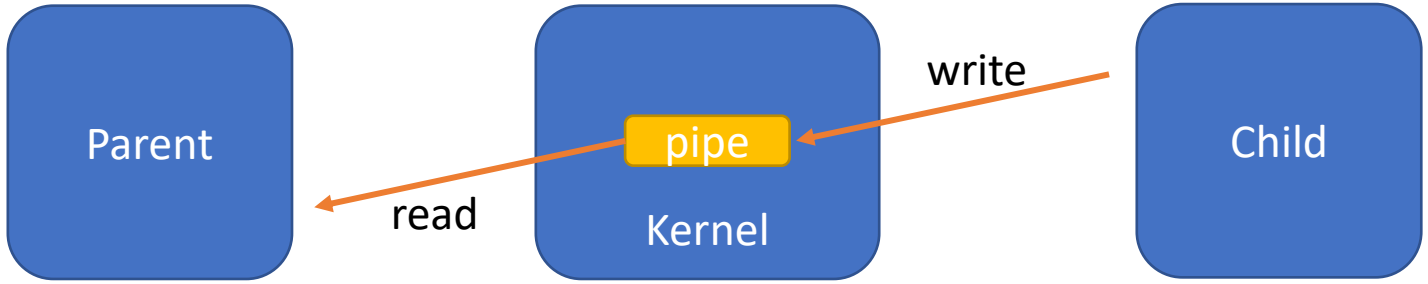
pipe

- Then, the creating process typically forks a child process. Since the child process inherits any file descriptor from the parent, we now have the basis for inter-process communication.
- Then, a decision has to be made. In which direction do we desire data to travel?



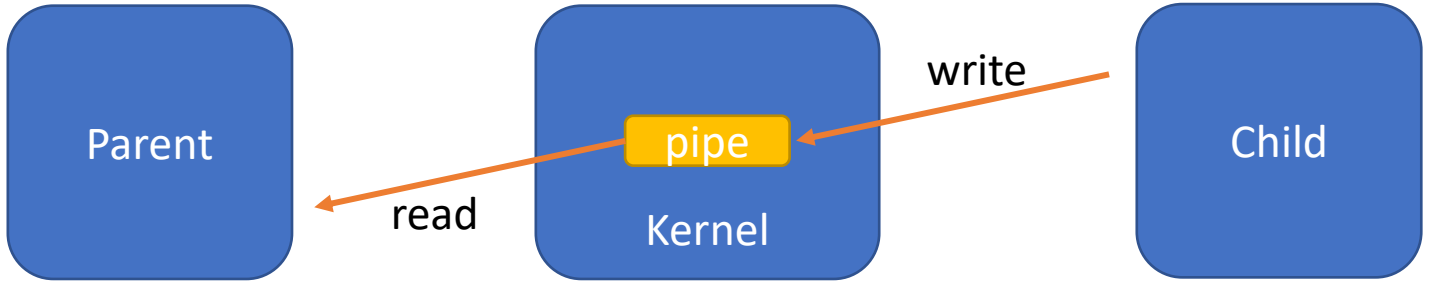
pipe

- Then, the creating process typically forks a child process. Since the child process inherits any file descriptor from the parent, we now have the basis for inter-process communication.
- Then, a decision has to be made. In which direction do we desire data to travel?
- Unused file descriptors are closed.



pipe

- We now have a pipe!
- `int pipe( int fd[2] );`
- When success
  - Returns 0
  - `fd` contains the opened file descriptors
- It is user's responsibility to then spawn child processes.



Dilema: Where to read from?

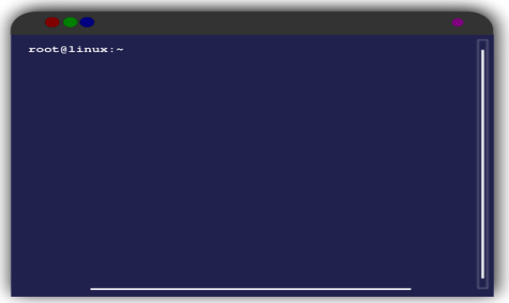
- We have two sources of input
- `read()` is a blocking call

User types into keyboard

(Client)

stdin

stdout



Terminal Process

(Server)

(forward)
stdin
stdout

Shell: Simply a macro processor that executes commands.

Shell (forked)

Dilema: Where to read from?

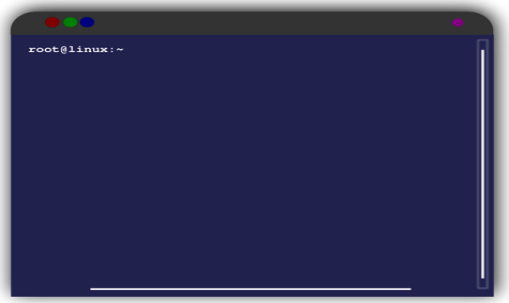
- We have two sources of input
- Read is a blocking call
- If we read from keyboard
  - Block until keyboard input
  - Cannot receive input from shell
- If we read from shell
  - Block and cannot receive input from keyboard

User types into keyboard

(Client)

stdin

stdout



Terminal Process

(Server)

(forward)
stdin
stdout

Shell: Simply a macro processor that executes commands.

Shell (forked)

poll

- It waits for one of a set of file descriptors to become ready to perform I/O (input/output multiplexing)
  - `int poll(struct pollfd *fds, nfds_t nfds, int timeout)`
  - The set of file descriptors to be monitored is specified in the `fds` argument, which is an array of structs of the following form:

```
struct pollfd {
    int   fd;         /* file descriptor */
    short events;     /* requested events */
    short revents;    /* returned events */
};
```

poll

- It waits for one of a set of file descriptors to become ready to perform I/O (input/output multiplexing)
  - `int poll(struct pollfd *fds, nfds_t nfds, int timeout)`
  - The set of file descriptors to be monitored is specified in the `fds` argument, which is an array of structures of the following form:

```
struct pollfd {
    int   fd;         /* file descriptor */
    short events;     /* requested events */
    short revents;    /* returned events */
};
```

- Input parameter:
  - Specifies what type of events are expected for this file descriptor.
- Expecter value:
  - `POLLIN`
  - `POLLOUT`
  - `...`

poll

- It waits for one of a set of file descriptors to become ready to perform I/O (input/output multiplexing)
  - `int poll(struct pollfd *fds, nfds_t nfds, int timeout)`
  - The set of file descriptors to be monitored is specified in the `fds` argument, which is an array of structures of the following form:

```
struct pollfd {
    int   fd;         /* file descriptor */
    short events;     /* requested events */
    short revents;    /* returned events */
};
```

- Output parameter:
  - Signals what type of event has actually happened
- Expecter value:
  - `POLLIN`
  - `POLLOUT`
  - `POLLERR`
  - `POLLHUP`
  - `POLLNVAL`
  - `...`

`poll`: an example

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/poll.h>

#define TIMEOUT 5

int main()
{
    struct pollfd fds[2];
    int ret;

    fds[0].fd = STDIN_FILENO;
    fds[0].events = POLLIN;

    fds[1].fd = STDOUT_FILENO;
    fds[1].events = POLLOUT;

    ret = poll(fds, 2, TIMEOUT*1000);

    if (ret == -1) {
        perror("poll");
        return 1;
    }

    if (!ret) {
        printf("%d seconds elapsed.\n", TIMEOUT);
        return 0;
    }

    if (fds[0].revents & POLLIN) printf("stdin is readable\n");
    if (fds[1].revents & POLLOUT) printf("stdout is writable\n");

    return 0;
}
```

- What would happen if we run
  - ./a.out
  - ./a.out < someText.txt

`poll`: an example

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/poll.h>

#define TIMEOUT 5

int main()
{
    struct pollfd fds[2];
    int ret;

    fds[0].fd = STDIN_FILENO;
    fds[0].events = POLLIN;

    fds[1].fd = STDOUT_FILENO;
    fds[1].events = POLLOUT;

    ret = poll(fds, 2, TIMEOUT*1000);

    if (ret == -1) {
        perror("poll");
        return 1;
    }

    if (!ret) {
        printf("%d seconds elapsed.\n", TIMEOUT);
        return 0;
    }

    if (fds[0].revents & POLLIN) printf("stdin is readable\n");
    if (fds[1].revents & POLLOUT) printf("stdout is writable\n");

    return 0;
}
```

- What would happen if we run
  - ./a.out
  - ./a.out < someText.txt

`poll`: an example

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/poll.h>

#define TIMEOUT 5

int main()
{
    struct pollfd fds[2];
    int ret;

    fds[0].fd = STDIN_FILENO;
    fds[0].events = POLLIN;

    fds[1].fd = STDOUT_FILENO;
    fds[1].events = POLLOUT;

    ret = poll(fds, 2, TIMEOUT*1000);

    if (ret == -1) {
        perror("poll");
        return 1;
    }

    if (!ret) {
        printf("%d seconds elapsed.\n", TIMEOUT);
        return 0;
    }

    if (fds[0].revents & POLLIN) printf("stdin is readable\n");
    if (fds[1].revents & POLLOUT) printf("stdout is writable\n");

    return 0;
}
```

- What would happen if we run
  - ./a.out
    - stdout is writable
  - ./a.out < someText.txt
    - stdin is readable
    - stdout is writable

`poll`: an example

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/poll.h>

#define TIMEOUT 5

int main()
{
    struct pollfd fds[2];
    int ret;

    fds[0].fd = STDIN_FILENO;
    fds[0].events = POLLIN;

    fds[1].fd = STDOUT_FILENO;
    fds[1].events = POLLOUT;

    ret = poll(fds, 2, TIMEOUT*1000);

    if (ret == -1) {
        perror("poll");
        return 1;
    }

    if (!ret) {
        printf("%d seconds elapsed.\n", TIMEOUT);
        return 0;
    }

    if (fds[0].revents & POLLIN) printf("stdin is readable\n");
    if (fds[1].revents & POLLOUT) printf("stdout is writable\n");

    return 0;
}
```

- What would happen if we run
  - ./a.out
    - stdout is writable
  - ./a.out < someText.txt
    - stdin is readable
    - stdout is writable
  - How to access content of someText.txt?

`poll`: an example

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/poll.h>

#define TIMEOUT 5

int main()
{
    struct pollfd fds[2];
    int ret;

    fds[0].fd = STDIN_FILENO;
    fds[0].events = POLLIN;

    fds[1].fd = STDOUT_FILENO;
    fds[1].events = POLLOUT;

    ret = poll(fds, 2, TIMEOUT*1000);

    if (ret == -1) {
        perror("poll");
        return 1;
    }

    if (!ret) {
        printf("%d seconds elapsed.\n", TIMEOUT);
        return 0;
    }

    if (fds[0].revents & POLLIN) printf("stdin is readable\n");
    if (fds[1].revents & POLLOUT) printf("stdout is writable\n");

    return 0;
}
```

- What would happen if we run
  - ./a.out
    - stdout is writable
  - ./a.out < someText.txt
    - stdin is readable
    - stdout is writable
  - How to access content of someText.txt?
    - read(fds[0].fd, …)

Special Characters: `\n` and `\r`

- If you read \r or \n
  - Echo to stdout as \r\n
  - Forward to shell as \n

Special Characters: `^C(ctrl-c)`

- Usually this would send an interrupt signal to our shell program
- Because of our termios setting, this doesn’t happen
- Here what you should do
  - Echo to stdout as ^C (special character corresponding to ctrl-c is not printable)
  - Use kill(2) to send an interrupt signal to the shell program
  - As a result, the shell program may or may not die
  - If it dies, proceed to shutdown everything
  - Otherwise, continue as usual

One more thing: shutting down

- Signs to shutdown:
  - Receive an EOF (^D, or 0x04) from the terminal
  - Pipe from shell is closed (POLLHUP or read size == 0)
- What to do?
  - Close the pipe to the shell, but continue processing input from the shell.
  - Use `waitpid(2)` to collect exit status of the shell program

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

One more thing: shutting down

- **pid_t** waitpid(pid_t pid, int *status, int options)
  - pid of the terminated child
  - On error, return -1

One more thing: shutting down

- `pid_t waitpid(`**`pid_t pid`**`, int *status, int options)`
  - `pid of the child process we are waiting on`

One more thing: shutting down

- `pid_t waitpid(pid_t pid, int *status, int options)`
  - `Output parameter that will store the exit information`

One more thing: shutting down

- `pid_t waitpid(pid_t pid, int *status, int options)`
  - OR of the following flags
    - WNOHANG: return immediately if no child exited
    - WUNTRACED: …, also return if a child has stopped. Exit status is provided
    - WCONTINUED: …, also return if a stopped child has been resumed by SIGCONT

Questions?

- Remember to check for errors after all system calls.