CS 111 Week 2

Project 1A: Terminal IO and IPC

Discussion 1B

Tianxiang Li

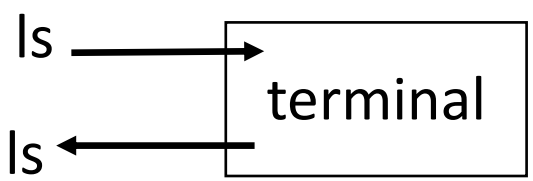
Project 1A

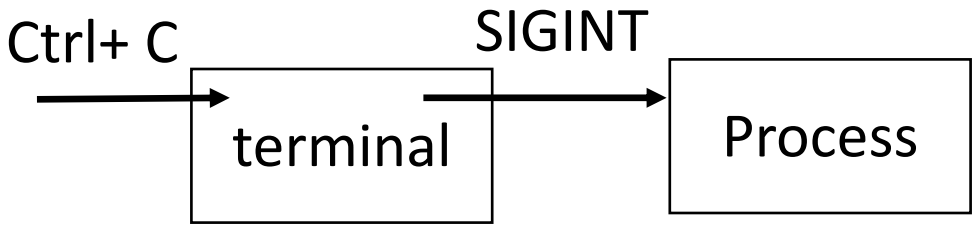
- Terminal
  - Character-at-a-time, full duplex terminal I/O
- Shell
  - Passing input and output between two processes
  - Shutdown processing

Part A: Terminal

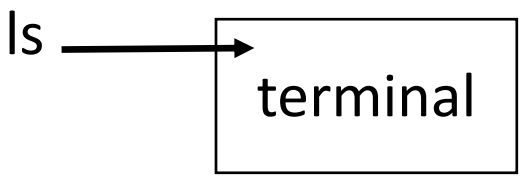
1. Set the stdin into no echo mode, non-canonical input
2. Read from stdin, write to stdout
3. If read '\r' or '\n', write '\r' and '\n' to stdout.
4. When receive EOF (ctrl + D), restore the terminal to normal mode and exit

default terminal mode:
echo and canonical





no echo and non-canonical mode:
implement echo and canonical mode with our own code



Non-canonical input, no echo mode: APIs

```
struct termios
{
        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];   /* special characters */
}
```

stores state in termios structure

int **tcgetattr**(int fd, struct termios *termios_p); //**get the state** of the terminal specified by fd

set params from termios structure

int **tcsetattr**(int fd, int optional_actions, const struct termios *termios_p);
//**set the state** of the terminal specified by fd

//optional_actions: how to set it (e.g. waits for all buffered input/output to finish?)

Returns 0 if success and -1 if non-success

Non-canonical input, no echo mode: example

```
void terminal_setup(void) //error handling code omitted for brevity
{
        struct termios tmp;
        tcgetattr(0, &tmp);
        tmp.c_iflag = ISTRIP, tmp.c_oflag = 0, tmp.c_lfag  = 0;
        tcsetattr(0, TCSANOW, &tmp);
}
```

change occurs immediately on start-up (before any characters are entered)

```
//set character-at-a-time no-echo mode
        //tcgetattr → get terminal mode, save normal mode
        //struct termios → specify required mode
        //tcsetattr → set mode
//restore saved terminal mode and exit
        //tcsetattr → restore mode
        //exit
```

Part A: Terminal

1. Set the stdin into non-canonical input, no echo mode

2. Read from stdin, write to stdout

3. If read ‘\r’ or ‘\n’, write ‘\r’ and ‘\n’ to stdout.

4. If read EOF (ctrl + D), restore the terminal to normal mode and exit

Part A: Terminal skeleton code

```
int main(void)
{
        terminal_setup();
        while (1)
        {
                read(0, buffer, sizeof(buffer));
                for (char in buffer)
                {
                        if (char == 0x4) //ctrl + D
                                write(1, “^D”), restore terminal, exit.
                        else if (char == ‘\r’ || char == ‘\n’)
                                write(1, “\r\n”);
                        else
                                write(1, char);
                }
        }
}
```

Part B: Shell

Extend your program to support a new command line option: --shell=program (*--shell=/bin/bash*)

1. Fork a new process to execute program
2. Forward the stdin of the original process to the stdin of the new process.
   '\r' or '\n' should be forwarded as '\n'
3. Forward the stdout/stderr of the new process to the stdout of the original process.
   '\n' should be forwarded as '\r'

1. When receive ctrl + C, send a SIGINT signal to the new process

2. When
   A. receive EOF from the stdin of the original process
   B. receive EOF from the stdout of the original process
   C. SIGPIPE signal

   Get all the output from the new process, report the exit status of the new process and exit.

Fork APIs

pid_t fork(void)

creates a new process (child process) by duplicating the calling process (parent process).

Child process has (almost) the same state as the parent process, including register/memory state and file descriptor.

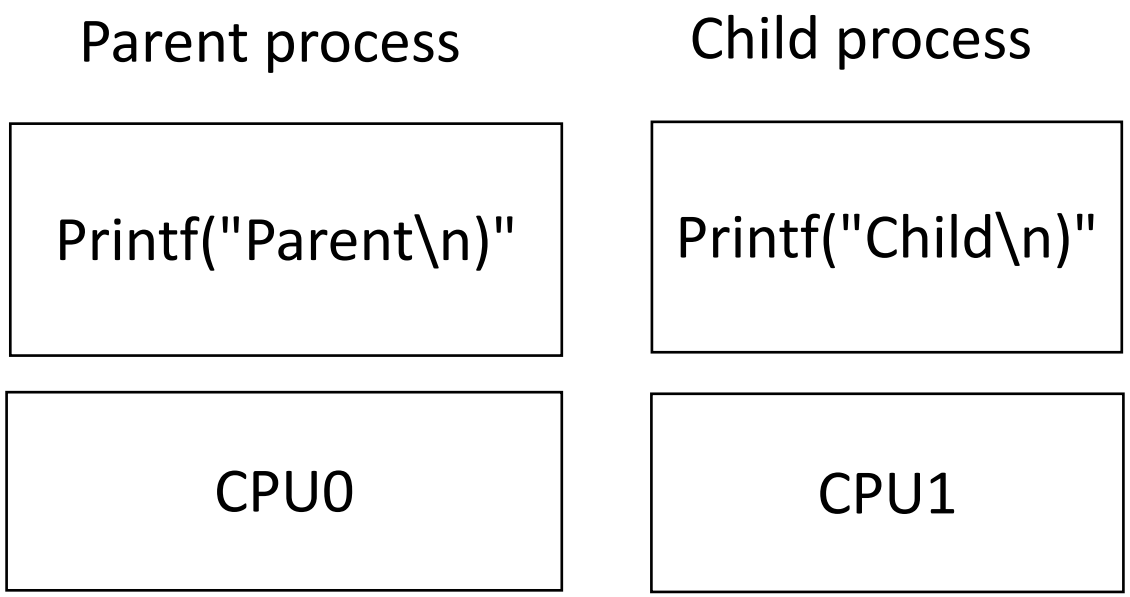
On success, return 0 to the child process
return pid of the child process to the parent process

Both parent and child process start executing from the next instruction

```
int main(void)
{
        fork();
        printf("Hello World\n");
}
```

Fork example

```
int main(void)
{
        int ret = fork();
        if (ret == 0)
        {
                printf(“Child\n”);
        }
        else if (ret > 0)
        {
                printf(“Parent\n”);
        }
}
//If ret <0 means error occured, e.g. system out of memory
```



Concurrent execution!
non-deterministic results!

What is the output of this program?

waitpid APIs

pid_t waitpid(pid_t pid, int *wstatus, int options);

Block and wait for the termination of the child process.

Exit status of the child process will be stored at wstatus.

Low order 7-bits of wstatus: signal number → *WTERMSIG(wstatus)*

The next higher order byte of wstatus: exit code → *WEXITSTATUS(wstatus)*

```
SHELL EXIT SIGNAL=0 STATUS=0
```

- After you have closed the write pipe to the shell and processed the final output returned from the shell:
  - use *waitpid(2)* to await the process' completion and capture its return status

Fork and waitpid example

```
int main(void)
{
        int ret = fork();
        if (ret == 0)
        {
                printf("Child process\n")
                exit(5);
        }
        else if (ret > 0)
        {
                int status  = 0;
                waitpid(ret, &status, 0);
                printf("Child process exits with code: %d\n", WEXITSTATUS(status));
        }
}
```

Fork and waitpid example

```
int main(void) {
        int ret = fork();
        if (ret == 0) {
                printf("child!\n");
                exit(5);
        }
        else if (ret > 0) {
                int status  = 0;
                waitpid(ret, &status, 0);
                printf("Child process exits
                with: %d\n", WEXITSTATUS(status));
        }
}
```

What is the output of this program?

Parent process	Child process
waitpid: blocked	
	printf("child!\n");
	exit(5);
waitpid: return	
printf(...);	

Parent process	Child process
	printf("child!\n");
	exit(5);
waitpid: wakeup	
printf(...);	

waitpid: non-deterministic
→ deterministic

APIs to create a new process

int execlp(const char ***file**, const char ***arg**, ... /* (char *) NULL */);

Replaces the current process image with a new process image.

file: path to the new process image (e.g. your executable file name)
arg: command line arguments of the new process, NULL terminated.

arg[0] is the same as file name.

Example:

execlp(shell_pgm, shell_pgm, NULL);

Fork, exec and waitpid example

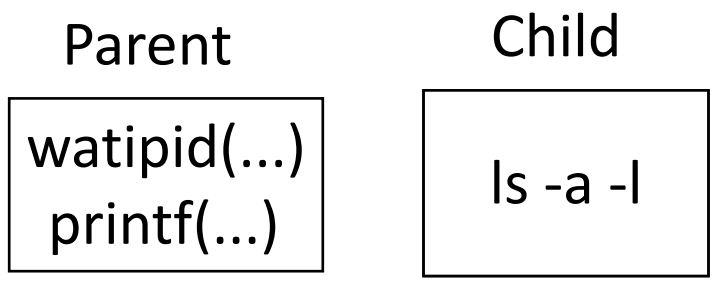
```
int main(void)
{
        int ret = fork();
        if (ret == 0)
        {
                execlp("ls", "ls", "-a", "-l", NULL);
                exit(5);
        }
        else if (ret > 0)
        {
                int status  = 0;
                waitpid(ret, &status, 0);
                printf("Child process exits with code: %d\n", (status & 0xff));
        }
}
```

Parent	Child
watipid(...) printf(...)	execlp(..) exit(5)

What's the value of WEXITSTATUS(status)?

Fork, exec and waitpid example

```
int main(void)
{
        int ret = fork();
        if (ret == 0)
        {
                execlp("ls", "ls", "-a", "-l", NULL);
                exit(5);
        }
        else if (ret > 0)
        {
                int status  = 0;
                waitpid(ret, &status, 0);
                printf("Child process exits with code: %d\n", (status & 0xff));
        }
}
```



What's the value of WEXITSTATUS(status)?

0 because execlp starts a new program

Part B: Shell

Extend your program to support a new command line option: --shell=program

1. Fork a new process to execute program
2. Forward the stdin of the original process to the stdin of the new process. '\r' or '\n' should be forwarded as '\n'
3. Forward the stdout/stderr of the new process to the stdout of the original process. '\n' should be forwarded as '\r'

4. When receive ctrl + C, send a SIGINT signal to the new process

5. When
   A. receive EOF from the stdin of the original process
   B. receive EOF/Error from the stdout/stderr of the new process
   C. receive SIGPIPE signal

   Get all the output from the new process, report the exit status of the new process and exit.

Pipe syscalls

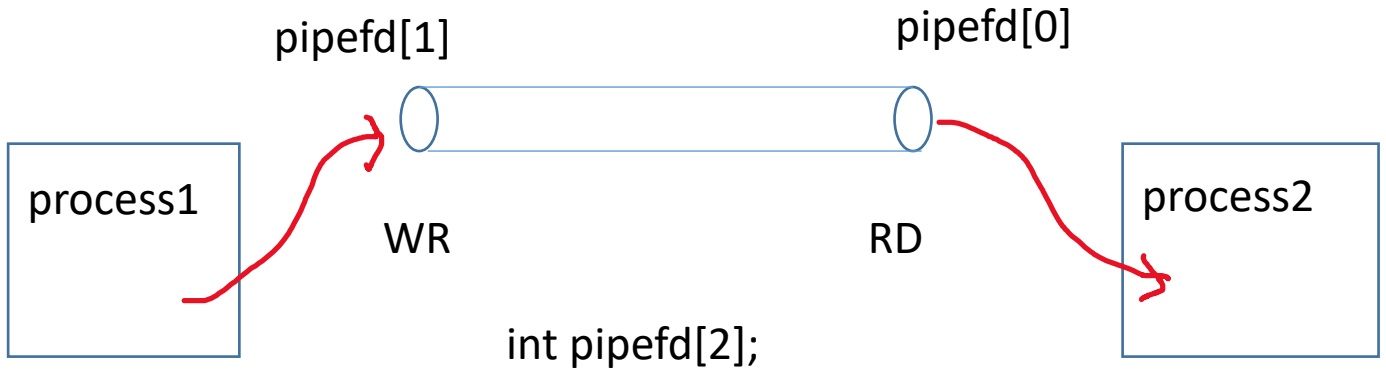
int pipefd[2];

int pipe(pipefd);

Creates a pipe, a *unidirectional* data channel (*inter-process communication*).

pipefd *returns* two file descriptor.

pipefd[0] is the read end of the pipe and pipefd[1] is the write end of the pipe



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

0	
1	
2	
…	
pipefd[0]	RD
pipefd[1]	WR

Pipe syscall examples

```
int main(void)
{
    int fds[2];
    pipe(fds); //pass the pointer of fds array to pipe syscall
                // once returns: fds[0] for read end of pipe, fds[1] for write end of pipe
    write(fds[1], “hahaha”, 6);
    read(fds[0], buffer , 6); //buffer will be “hahaha”
}
```

pipe and fork

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

child
---
pipefd[0]
pipefd[1]

parent
---
pipefd[0]
pipefd[1]

execute → ~~stdout~~ dup

~~stdin~~ dup

Write
pipefd[1]

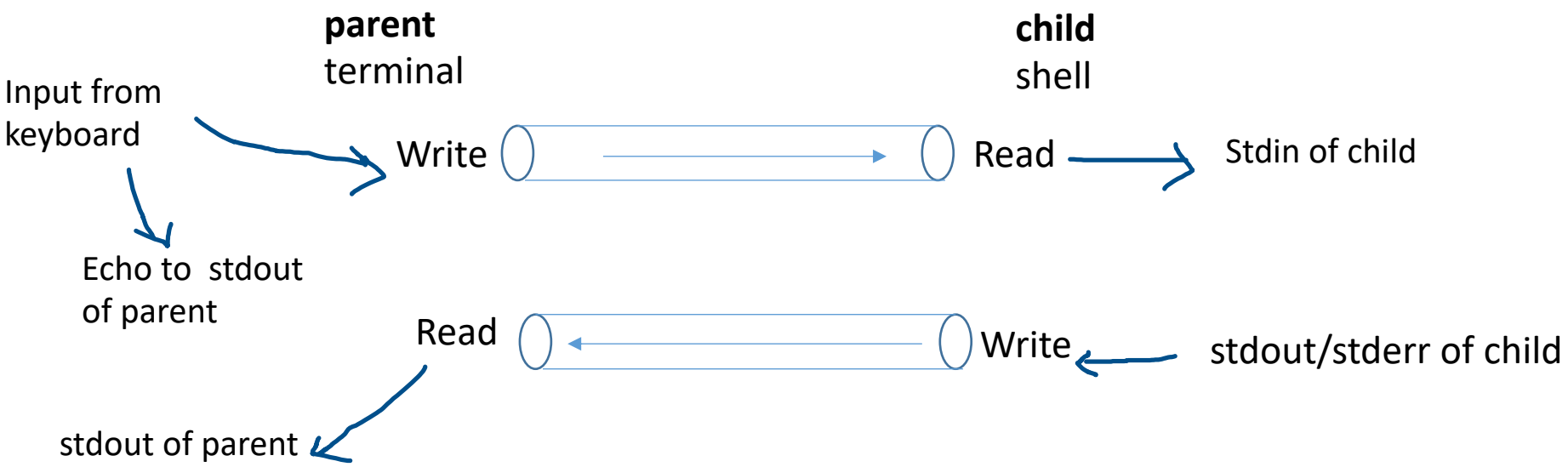
Read
pipefd[0]

Pipe syscall examples

```
int main(void)
{
    int fds[2];
    pipe(fds);
    ret = fork();
    if (ret == 0)
    {
        read(fds[0], buffer, 6); //buffer is “hahaha”
    }
    else if (ret > 0)
    {
        write(fds[1], “hahaha”, 6);
    }
}
```

Forward stdin/stdout between two process

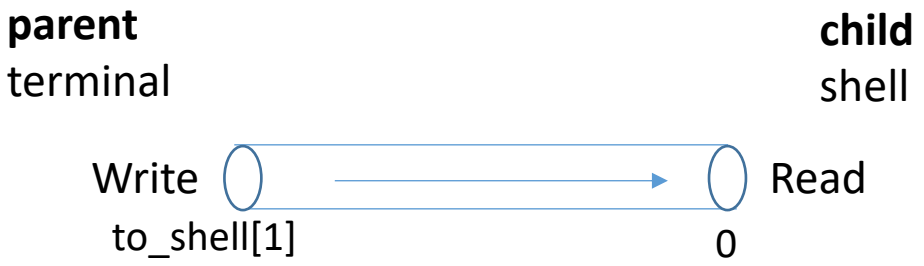
Create two pipes, one end is the stdin, stdout/stderr of the child process, the parent process writes/reads from the other end of the pipe.



Forward stdin/stdout between two process

Create two pipes, one end is the stdin, stdout/stderr of the child process, the parent process writes/reads from the other end of the pipe.

```
int to_shell[2];
pipe(to_shell);
if (fork() == 0) {   //child process
    close(to_shell[1]);
    close(0);
    dup(to_shell[0]);  //0 → read end of the pipe
    close(to_shell[0]);
    execlp(...);
}
else { //parent process
        close(to_shell[0]);
        while (read(0, buffer, sizeof(buffer) > 0)
                write(to_shell[1], buffer, sizeof(buffer));
}
```



child process

0 → stdin

to_shell[0] → read end of pipe

to_shell[1] → write end of pipe

parent process

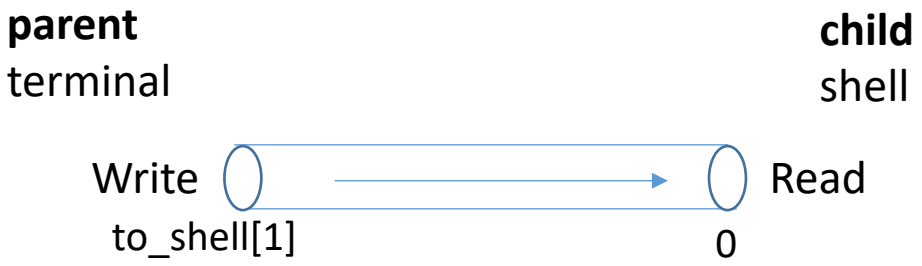
to_shell[0] → read end of pipe

to_shell[1] → write end of pipe

Forward stdin/stdout between two process

Create two pipes, one end is the stdin, stdout/stderr of the child process, the parent process writes/reads from the other end of the pipe.

```
int to_shell[2];
pipe(to_shell);
if (fork() == 0) {   //child process
    close(to_shell[1]);
    close(0);
    dup(to_shell[0]);  //0 → read end of the pipe
    close(to_shell[0]);
    execlp(...);
}
else { //parent process
        close(to_shell[0]);
        while (read(0, buffer, sizeof(buffer) > 0)
                write(to_shell[1], buffer, sizeof(buffer));
}
```



child process

0 → stdin

to_shell[0] → read end of pipe

to_shell[1] write end of pipe

parent process

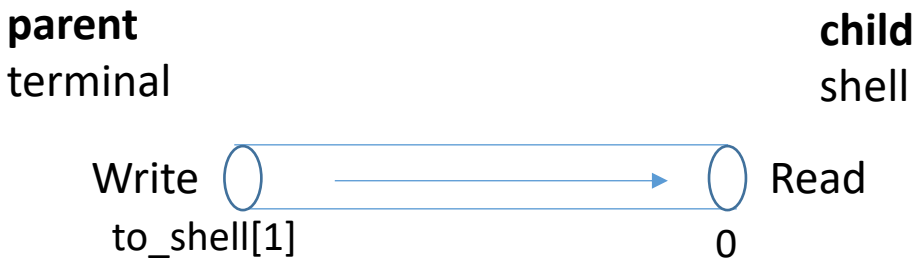
to_shell[0] → read end of pipe

to_shell[1] → write end of pipe

Forward stdin/stdout between two process

Create two pipes, one end is the stdin, stdout/stderr of the child process, the parent process writes/reads from the other end of the pipe.

```
int to_shell[2];
pipe(to_shell);
if (fork() == 0) {   //child process
    close(to_shell[1]);
    close(0);
    dup(to_shell[0]);  //0 → read end of the pipe
    close(to_shell[0]);
    execlp(...);
}
else { //parent process
        close(to_shell[0]);
        while (read(0, buffer, sizeof(buffer) > 0)
                write(to_shell[1], buffer, sizeof(buffer));
}
```



child process

0	stdin
to_shell[0] →	read end of pipe
to_shell[1]	write end of pipe

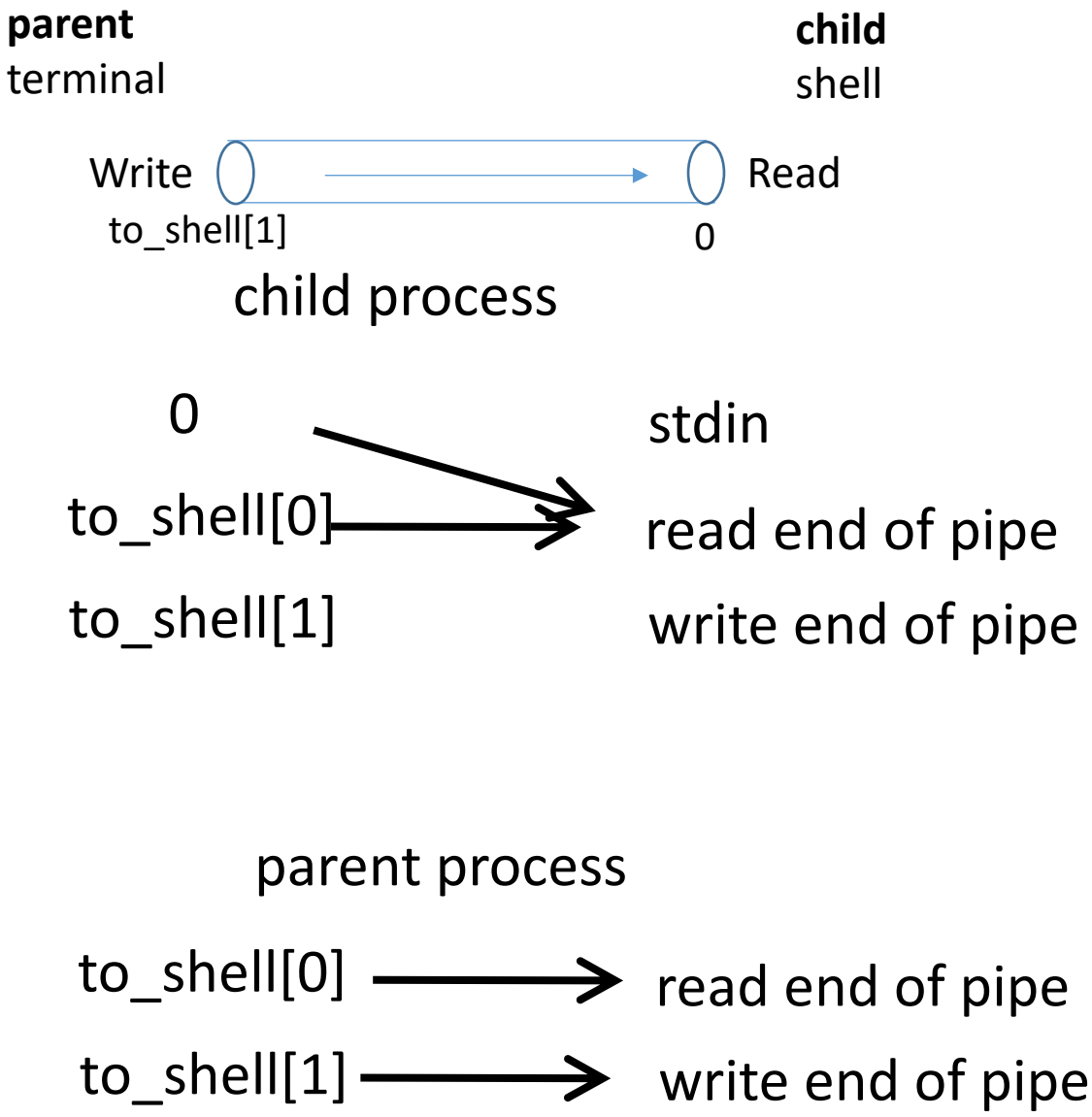
parent process

to_shell[0] →	read end of pipe
to_shell[1] →	write end of pipe

Forward stdin/stdout between two process

Create two pipes, one end is the stdin, stdout/stderr of the child process, the parent process writes/reads from the other end of the pipe.

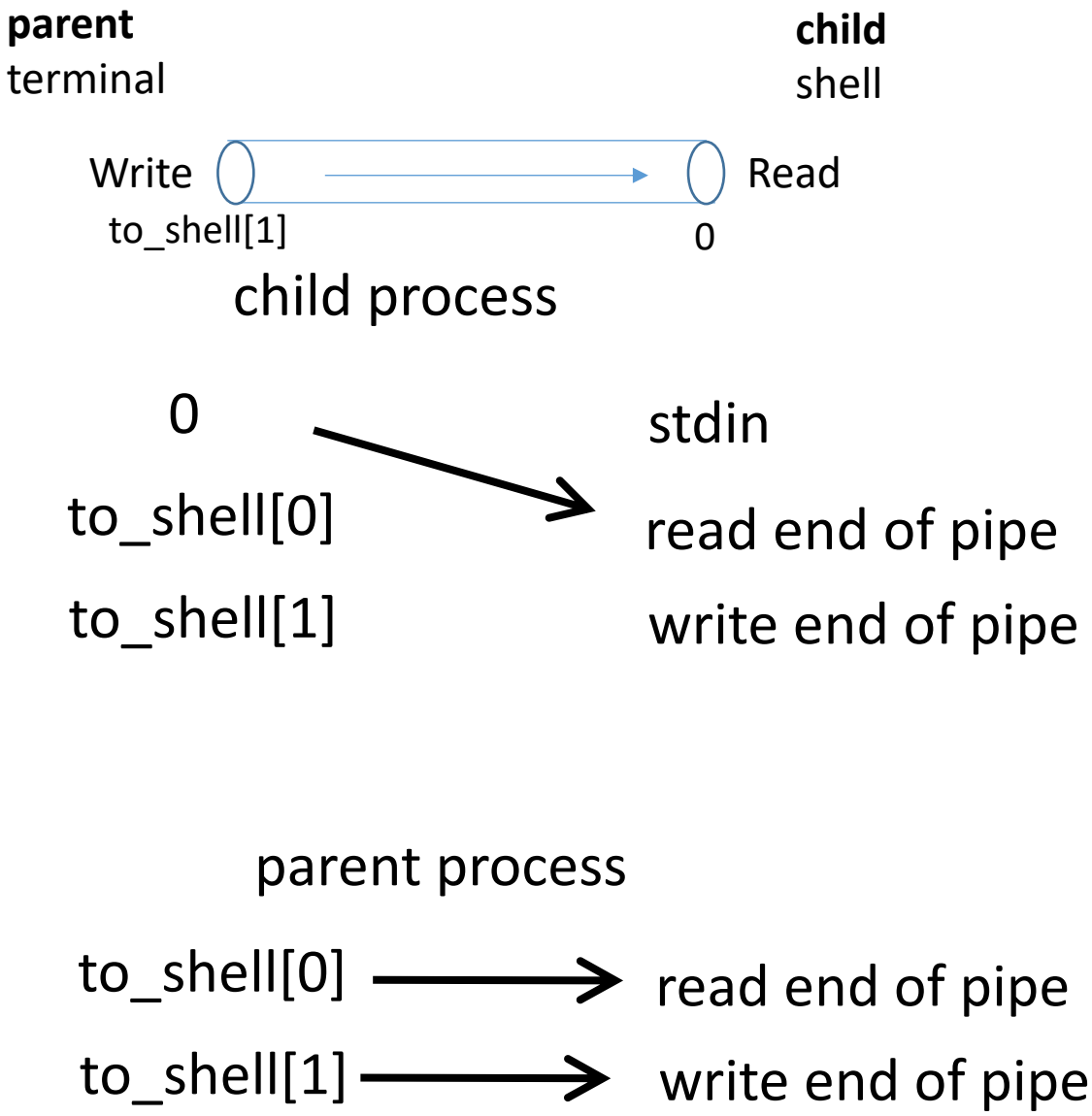
```
int to_shell[2];
pipe(to_shell);
if (fork() == 0) {   //child process
    close(to_shell[1]);
    close(0);
    dup(to_shell[0]);  //0 → read end of the pipe
    close(to_shell[0]);
    execlp(...);
}
else { //parent process
        close(to_shell[0]);
        while (read(0, buffer, sizeof(buffer) > 0)
                write(to_shell[1], buffer, sizeof(buffer));
}
```



Forward stdin/stdout between two process

Create two pipes, one end is the stdin, stdout/stderr of the child process, the parent process writes/reads from the other end of the pipe.

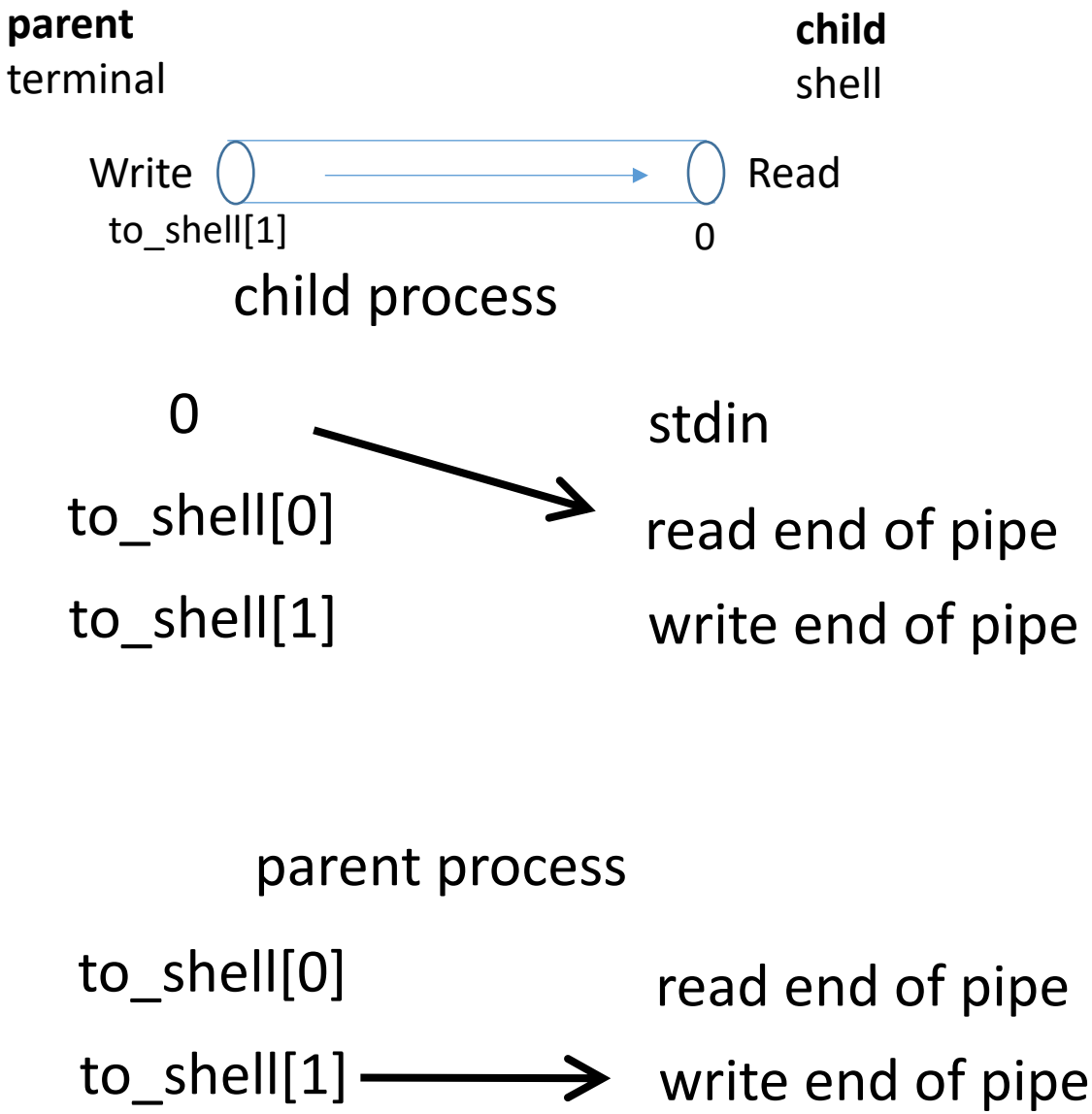
```
int to_shell[2];
pipe(to_shell);
if (fork() == 0) {   //child process
    close(to_shell[1]);
    close(0);
    dup(to_shell[0]);  //0 → read end of the pipe
    close(to_shell[0]);
    execlp(...);
}
else { //parent process
        close(to_shell[0]);
        while (read(0, buffer, sizeof(buffer) > 0)
                write(to_shell[1], buffer, sizeof(buffer));
}
```



Forward stdin/stdout between two process

Create two pipes, one end is the stdin, stdout/stderr of the child process, the parent process writes/reads from the other end of the pipe.

```
int to_shell[2];
pipe(to_shell);
if (fork() == 0) {   //child process
    close(to_shell[1]);
    close(0);
    dup(to_shell[0]);  //0 → read end of the pipe
    close(to_shell[0]);
    execlp(...);
}
else { //parent process
        close(to_shell[0]);
        while (read(0, buffer, sizeof(buffer) > 0)
                write(to_shell[1], buffer, sizeof(buffer));
}
```



```
handle_cmd_line(argc, argv); // prog = "new program to execute"
Create pipes
if (fork() == 0) { //child
        close unused end of pipes
        redirect stdin, stdout, stderr, to/from pipes.
        execlp(prog, prog, NULL);
}
else {  //parent process
        close unused end of pipes
        while (1) {
                read(stdin, buffer, sizeof(buffer));  //input from keyboard
                handle "\r", "\n", forward to pipe, stdout
                read(from_shell[0], buffer, sizeof(buffer)); //input from shell
                handle "\n", forward to  stdout
        }
}
```

Problem: Read may be blocked forever!

Solution: Make sure read will not block before reading it.

poll APIs

int poll(struct pollfd ***fds**, nfds_t **nfds**, int timeout);

poll blocks and returns when

(1) one of a set of file descriptors to become ready to perform I/O or

(2) the specified timeout (number of miliseconds) has passed.

fds: an array of struct pollfd
nfds: number of elements in fds array

```
struct pollfd {
        int   fd;            /* file descriptor to monitor */
        short events;        /* events to monitor, e.g. ready to read, errors occurred */
        short revents;       /* return value, encode which events have occurred */
    };
```

```
pollfds[0].fd = 0;
pollfds[0].events = POLLIN + POLLHUP + POLLERR;
pollfds[1].fd = from_shell[0];
pollfds[1].events = POLLIN + POLLHUP + POLLERR;

while (1) {
        poll(pollfds, 2, -1); // -1 means no time out.
        if (pollfds[0].revents & POLLIN) {
                count = read(0, buf, sizeof(buf));
                forward to stdout, to_shell [1]
        }
        if (pollfds[1].revents & POLLIN) {
                count = read(from_shell[0], buf, sizeof(buf));
                forward to stdout.
        }
        if (pollfds[0].revents & (POLLHUP | POLLERR))  {
        }
        ....
}
```

POLLIN: data to read,
POLLHUP: FD is closed by the other side,
POLLERR: error occurred

```
struct pollfd {
        int   fd;          /* file descriptor to monitor */
        short events;      /* events to monitor, e.g. ready to read, errors occurred */
        short revents;     /* return value, encode which events have occurred */
};
```

Part B: Shell

Extend your program to support a new command line option: --shell=program

1. Fork a new process to execute program
2. Forward the stdin of the original process to the stdin of the new process. '\r' or '\n' should be forwarded as '\n'
3. Forward the stdout/stderr of the new process to the stdout of the original process. '\n' should be forwarded as '\r'

4. When receive ctrl + C, send a SIGINT signal to the new process

5. When
   A. receive EOF from the stdin of the original process
   B. receive EOF/Error from the stdout/stderr of the new process
   C. receive SIGPIPE signal

   Get all the output from the new process, report the exit status of the new process and exit.

kill syscalls APIs

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

sends the signal specified by sig to pid.

```
read(0, buffer, sizeof(buffer));
for (char in buffer)
{
    if (char == 0x03) //Recieve ctrl + C
    {
        write(1, “^C”, 2);
        kill(child_pid, SIGINT);
    }
}
```

Part B: Shell (Error Handling)

When

A. receive EOF from the stdin of the original process
B. receive EOF/Error from the stdout/stderr of the new process
C. receive SIGPIPE signal

Get all the **output** from the new process, report the **exit status** of the new process and **exit**.

For A: Detect ^D character in buffer

For B: Handled by poll: revents & (POLLHUP | POLLERR)

For C: Need to register **SIGPIPE** handler in parent process

indication that the shell has shut down