

PA2: Your Own Linux Shell

TANZIR AHMED
CSCE 313 SPRING 2021

Programming Assignment # 2

Main Challenges (after fork + exec)

- I/O Redirection
- Piping

To solve these challenges, we need the topics:

- File I/O basics
- Inter Process Communication using **pipes**

Main Idea in Pseudocode

A shell's main logic is as follows:

```
while (!eofinput){  
    1. get the next command from user  
    2. execute the command fully  
}
```

However, the we know that **exec()** function (i.e., system call to execute a command) does not return

- Overwrite the main process

Therefore, we need to **fork()** a child process and then **exec()**

- Note that, the parent (i.e., the shell) must **wait()** for the child
- For instance, “sleep 5” should block the shell for 5 seconds

Skeleton Code

```
int main (){
    while (true){
        cout << "My Shell$ "; //print a prompt
        string inputline;
        getline (cin, inputline); //get a line from standard input
        if (inputline == string("exit")){
            cout << "Bye!! End of shell" << endl;
            break;
        }
        int pid = fork ();
        if (pid == 0){ //child process
            // preparing the input command for execution
            char* args [] = {(char *) inputline.c_str(), NULL};
            execvp (args [0], args);
        }else{
            waitpid (pid, 0, 0); //parent waits for child process
        }
    }
}
```

Issues With Skeleton Code

The skeleton is almost a fully functional shell

- Can find commands in the PATH and current directory and execute them properly

However, the following features are missing:

- Cannot handle arguments to the program: `"ls -l"`
- I/O redirection (e.g., `"ls -l > a.txt"`, `"grep pattern < a.txt"`)
- Piping (e.g., `"ls -l | grep pattern1|grep pattern2|less -5"`)
- Directory handling commands (e.g., `"cd /path/to/dir"`)
- Handling background processes (e.g., `"sleep 5 &"`)
- Handling commands with arguments (e.g., `"ls -l -a"`)
- ...

File I/O – File Descriptors

```
int main ()
{
    int fd; // file descriptor
    char* buf [] = "file content";
    fd = open ("foobar.txt", O_CREAT|O_WRONLY);
    write (fd, buf, strlen (buf)); close (fd);
    fd = open ("foobar.txt", O_RDONLY, 0);
    read(fd, &c, 1);
    printf("c=%c\n", c);
    close (fd);
    return 0;
}
```

Every process has a file descriptor table, where there are 3 default entries to begin with:

- Standard input, output, and error

Descriptor table
[one table per process]

stdin	fd 0	
stdout	fd 1	
stderr	fd 2	
	fd 3	fd

I/O Redirection

Question: How does a shell implement I/O redirection?

```
unix> ls > foo.txt
```

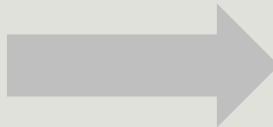
Answer: By calling the **dup2(source, destination)** function

- Copies (per-process) descriptor table entry **source** to entry **destination**

Descriptor table

before dup2 (3, 1)

fd 0	
fd 1	a
fd 2	
fd 3	b



Descriptor table

after dup2 (3, 1)

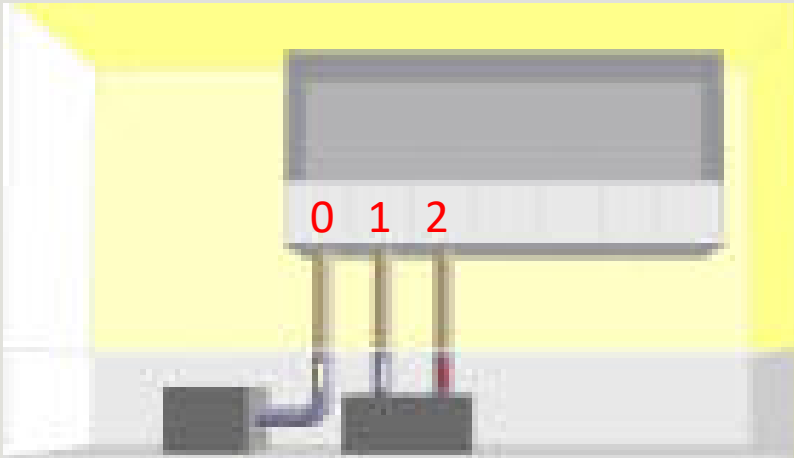
fd 0	
fd 1	b
fd 2	
fd 3	b

Implementing “ls -l -a -w> foo.txt”

```
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
int main ()
{
    int fd = open ("foo.txt", O_CREAT|O_WRONLY|O_TRUNC,
        S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    dup2 (fd, 1); // overwriting stdout with the new file
    execlp ("ls", "ls", "-l", "-a", "-w", NULL); // execute
    return 0;
}
```

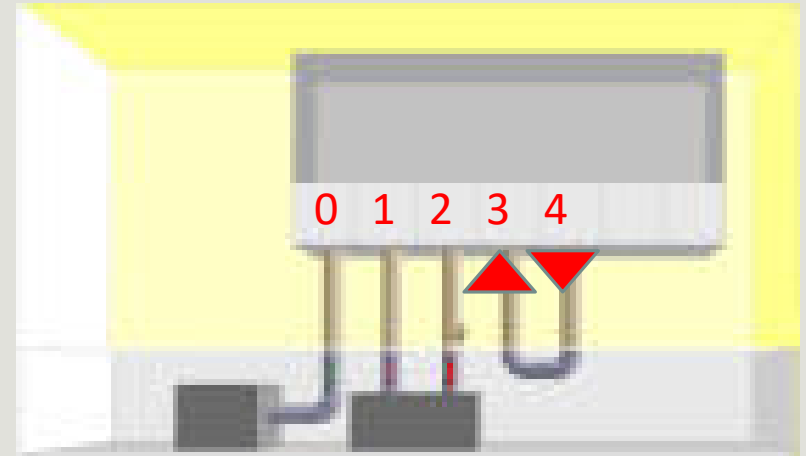

IPC Pipe

BEFORE pipe



Process has some usual files open

AFTER pipe



Kernel creates a pipe and sets file descriptors

IPC Pipe - Method

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

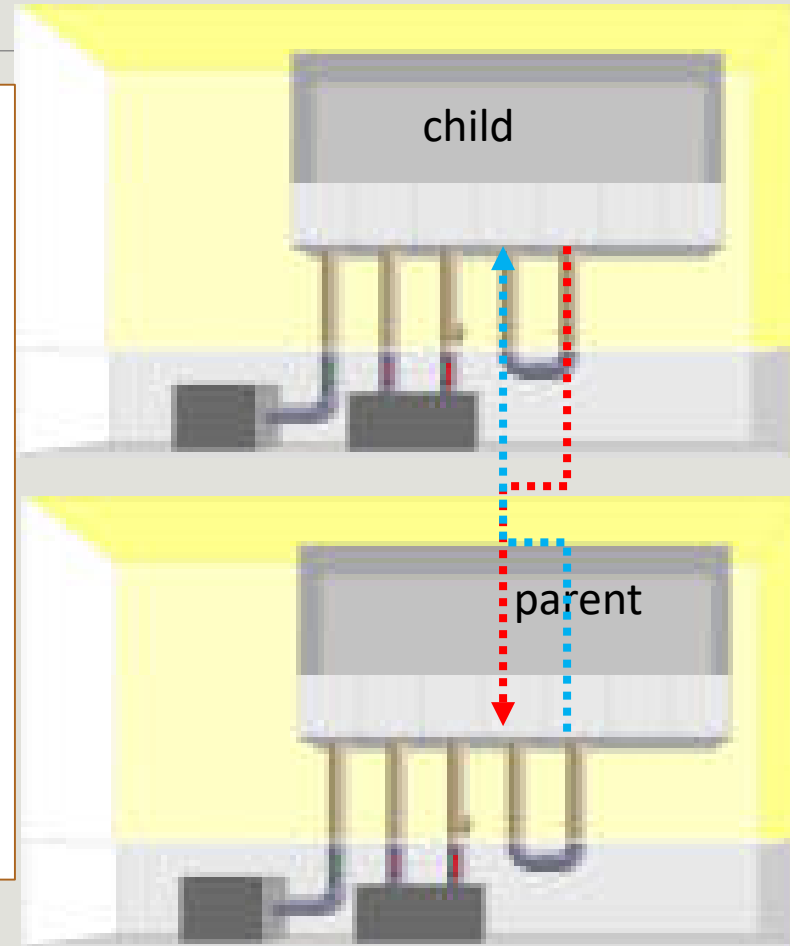
void main ()
{
    char buf [10];
    int fds [2];
    pipe (fds);
    printf ("sending msg: Hi\n");
    write (fds[1], "Hi", 3);
    read (fds[0], buf, 3);
    printf ("Received msg: %s\n", buf);
}
```

Connects the two
fds as pipe

```
compute-linux1 tanzir/code> ./a.out
sending msg: Hi
Received msg: Hi
```

Pipe Between Two Processes

```
int main ()
{
    int fds [2];
    pipe (fds); // connect the pipe
    if (!fork()){ // on the child side
        char * msg = "a test message";
        printf ("CHILD: Sent %s\n", msg);
        write (fds[1],msg,strlen(msg)+1);
    }else{
        char buf [100];
        read (fds [0], buf, 100);
        printf ("PARENT:Recv %s\n", buf);
    }
    return 0;
}
```



Shell Piping Example:

`"ls -l | grep soda"`

Meaning of the command:

- Find all files that has the string “soda” in the filename and show detailed properties of those files

How many processes do we have to run (in addition to our shell process)?

- Process # 1: To run “ls -l”
- Process # 2: To run “grep soda”

What else do we need so that the process #1 sends its output to process #2

- Idea: If we can connect **stdout of p1** to **stdin of p2**, we are done!!
- Step 1: Redirect stdout of p1 to a file descriptor fd1
- Step 2: Redirect stdin of p2 to a another file descriptor fd2
- Step 3: Now, pipe fd1 and fd2 together so that fd1 is the “write side” and fd2 is the “read side”

Shell Piping:

"ls -l | grep soda"

```
void main ()
{
    int fds [2];
    pipe (fds); // connect the pipe
    if (!fork()) { // on the child side
        dup2 (fds[1], 1); // redirect stdout to pipeout
        execlp ("ls", "ls", "-l", NULL);
    } else {
        dup2 (fds[0], 0); // redirect stdin to pipe in
        execlp ("grep", "grep", "soda", NULL);
    }
}
```

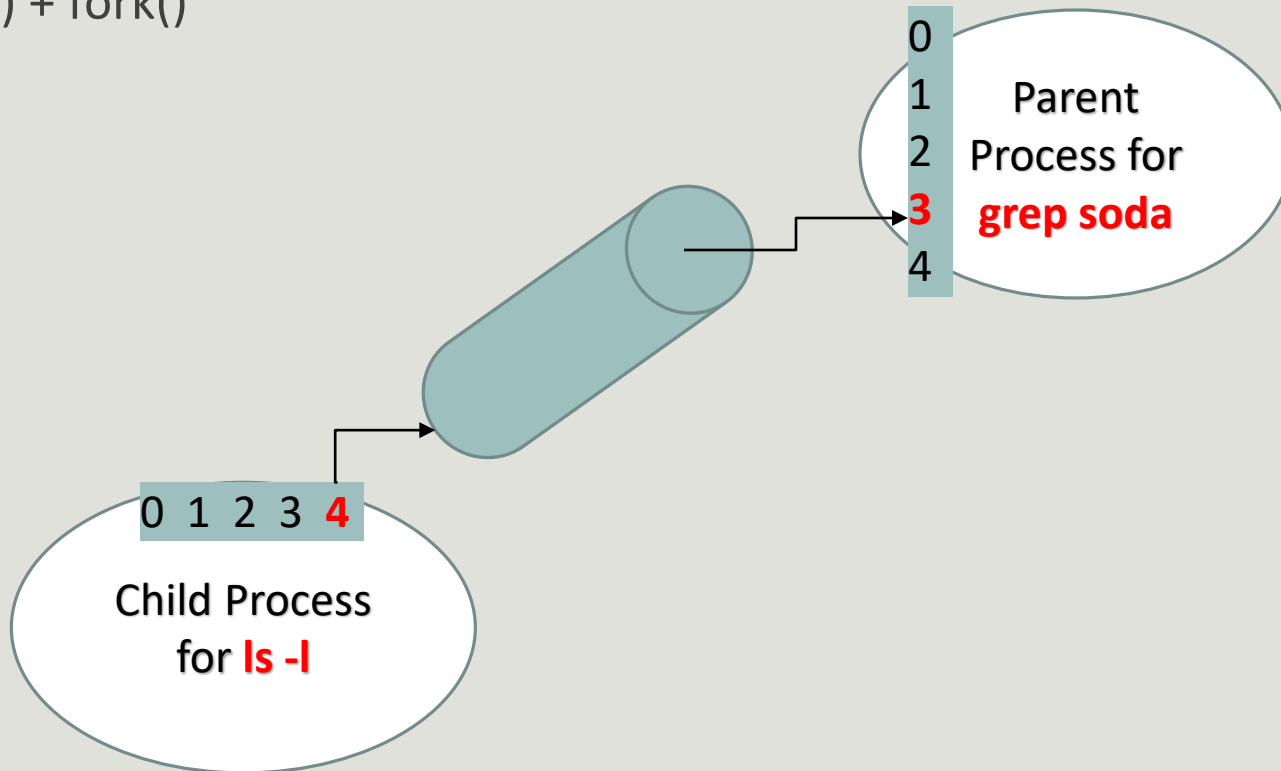
Step 3

Step 1

Step 2

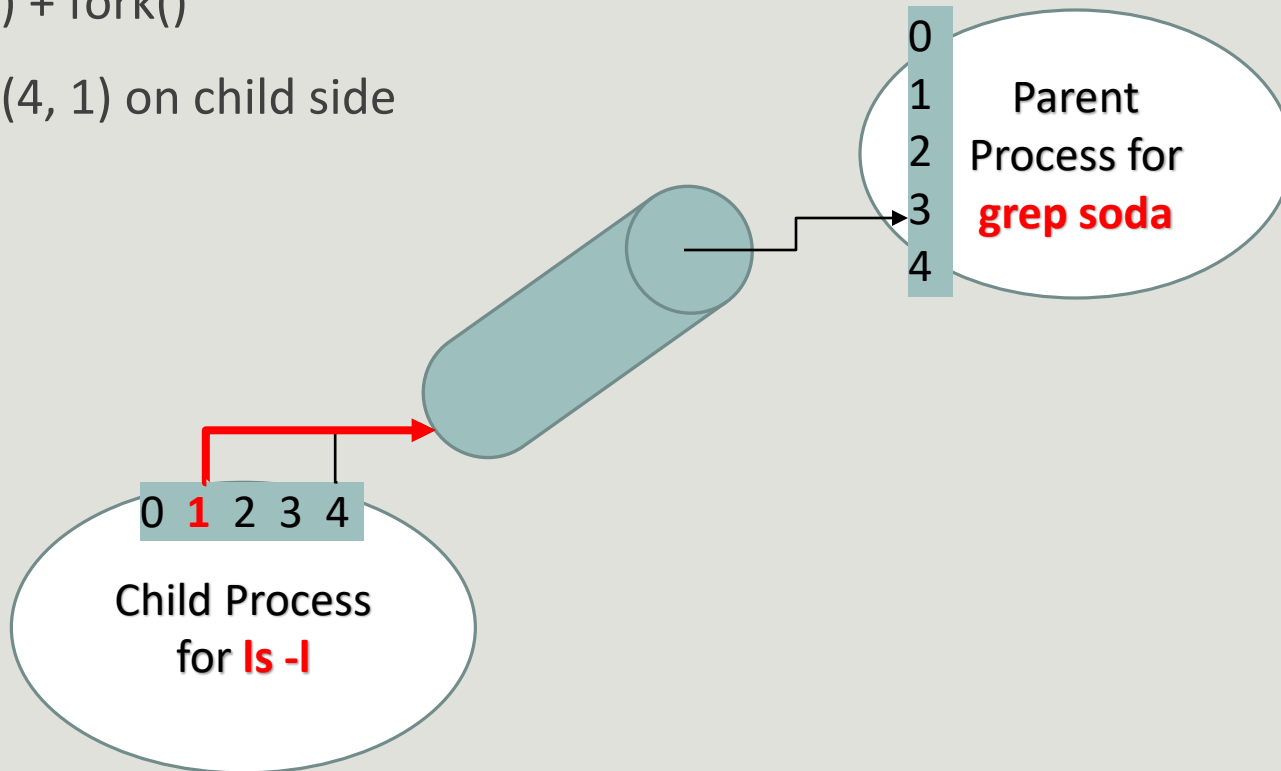
Visual Representation

1. Pipe() + fork()



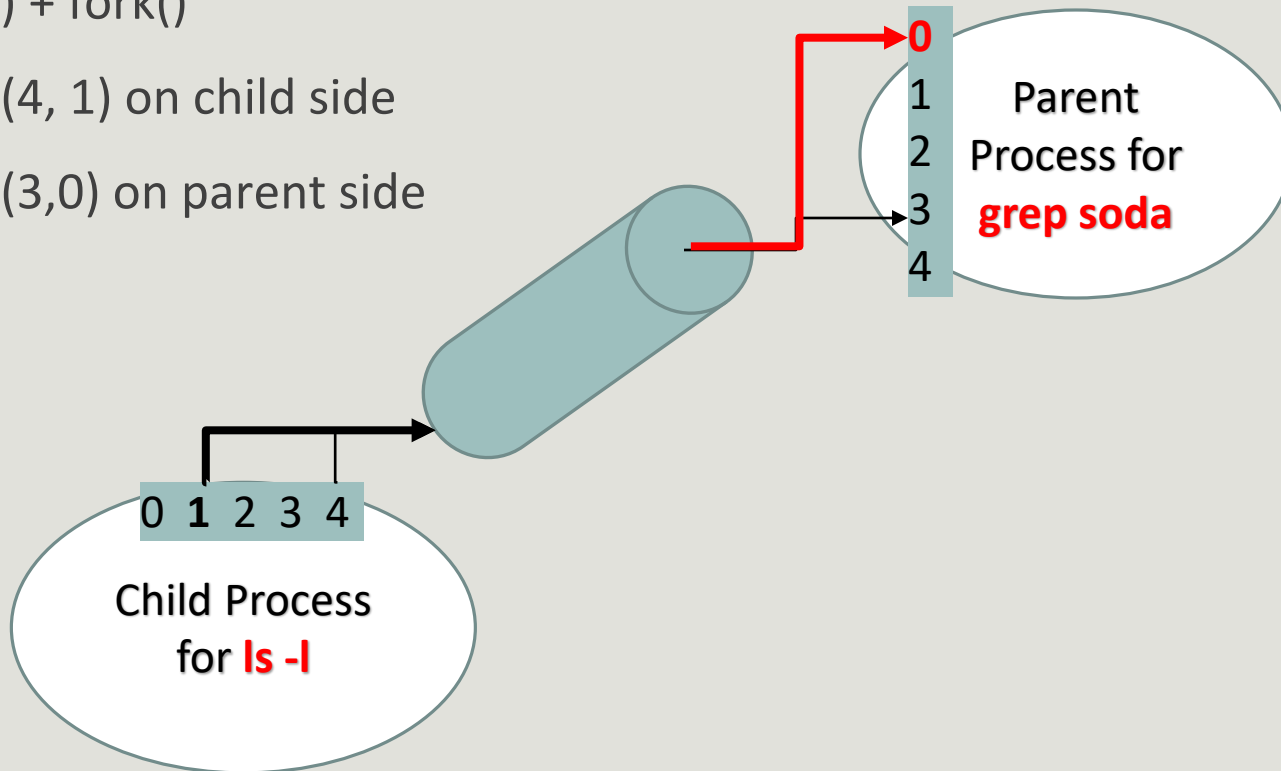
Visual Representation

1. Pipe() + fork()
2. dup2(4, 1) on child side



Visual Representation

1. Pipe() + fork()
2. dup2(4, 1) on child side
3. dup2(3,0) on parent side



Running “ls -l | grep soda”

We fork()-ed only once

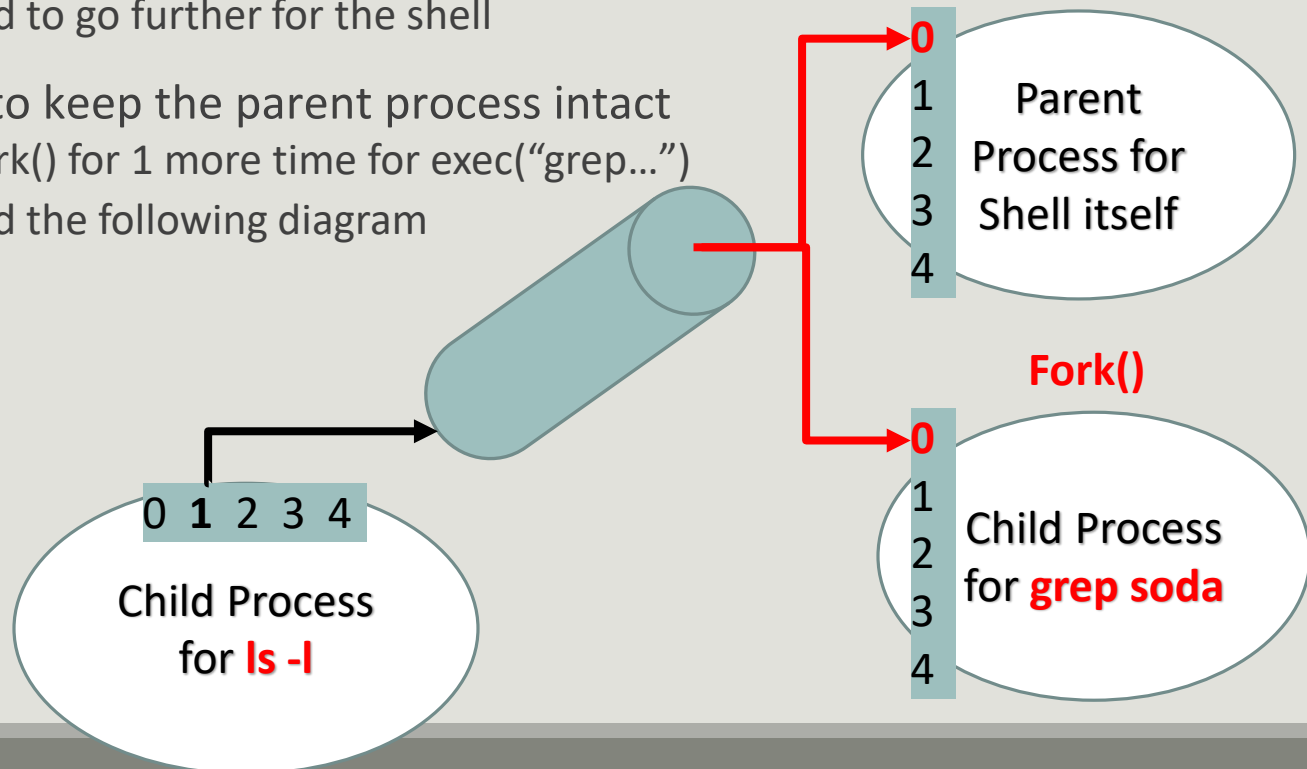
- Child ran “ls -l” and parent ran “grep soda”

But exec(“grep”..) wipes out parent, preventing any further action

- We need to go further for the shell

We need to keep the parent process intact

- Must fork() for 1 more time for exec(“grep...”)
- We need the following diagram



How about more than 2 levels of pipe?

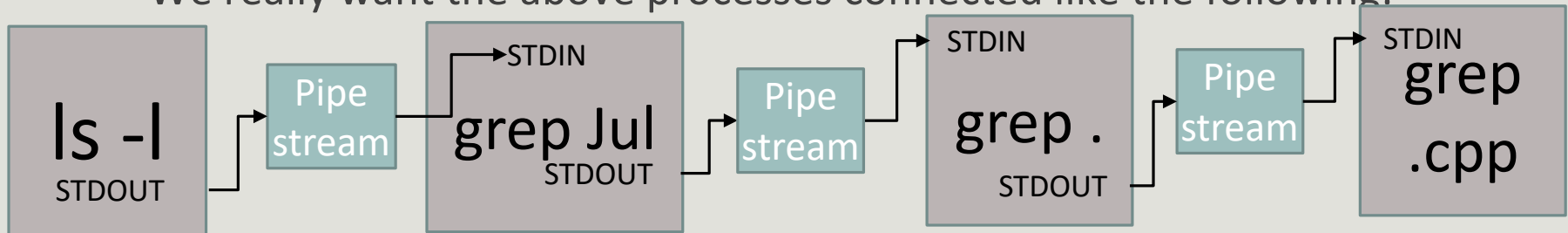
The previous code does not scale to more than 2 levels of “|” symbol

- For instance, how about:

```
ls -la | grep Jul | grep . | grep .cpp
```

We need a loop-structure that goes beyond the previously described hard-coded structure

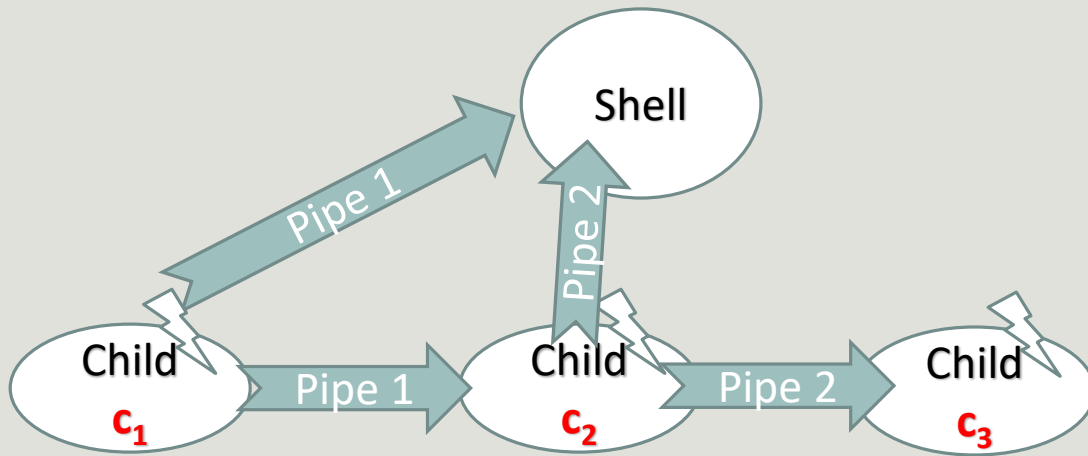
We really want the above processes connected like the following:



Also, we need a skeletal structure for PA4

Plan of Action

Say the input line is: " $c_1 \mid c_2 \mid \dots \mid c_n$ "



```

while (true){// continue until the user enters a blank like
    string line = read a line from the user;
    // now split the line by | symbol that gives us separate commands
    // You NEED to write this function
    // for instance: ls -la | grep Jul | grep . | grep .cpp
    vector<string> c = split (line, "|");
    //after the above, levels.size() == 4, because there are
    // 3 pipe symbols making 4 pipe levels

    for (int i=0; i< c.size(); i++){
        // set up the pipe
        int fd [2];
        pipe (fd);
        int pid = fork();
        if (!pid){ // in the child process
            // 1. redirect the output to the next level
            // 2. execute the command at this level
        }else{ // in the parent process
            waitpid (pid); /*1. wait for the child process running
            the current level command */
            //2. redirect input from the child process
        }
    }
}
}

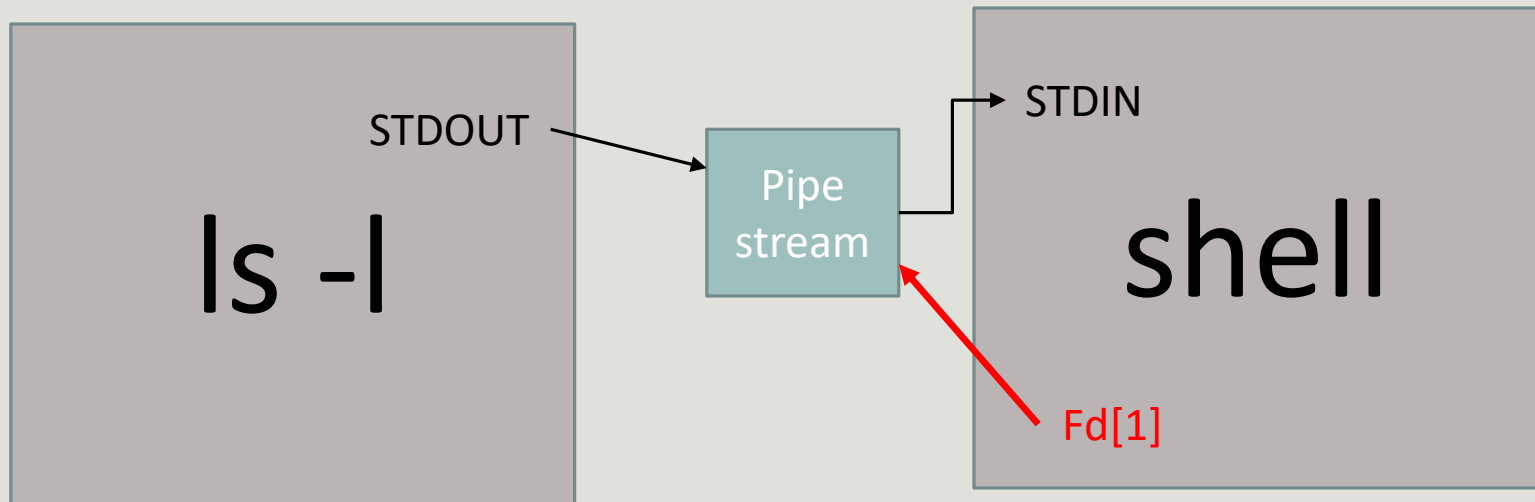
```

Each Iteration of the For Loop

```
int cid = fork();
if (!cid){// child process
    /* redirect output to the next level
       unless this is the last level */
    if (i < levels.size() - 1){
        dup2 (fd[1], 1); // redirect STDOUT to fd[1], so that it
                           can write to the other side
                           be closed
    }
    //execute function that can split the command by spaces to
    // find out all the arguments, see the definition
    execute (levels [i]); // this is where you execute the
                           command, you NEED to write this function
} else{
    if (i == levels.size() - 1) // wait only for the last child
        waitpid (cid);
    dup2 (fd [0], 0); // now redirect the input for the next
loop iteration
    close (fd [1]); // fd [1] MUST be closed, otherwise the next
level will wait
}
```

Why close fd[1]?

Because with pipe() and redirection, we obtain a picture like the following:



If we do not close fd[1], the next level process grep thinks that there is more data coming and thus keeps waiting

Therefore, closing the fd[1] side finally provides the EOF (end of file) symbol to the input to grep, which then finishes

Zombie Processes

Idea

- When process terminates, still consumes system resources
 - Various tables maintained by OS (to store exit status)
- Called a “zombie”
 - Living corpse, half alive and half dead

Reaping

- Performed by parent on terminated child by calling **wait()** function
- Parent is given exit status (i.e., **wait (&status)**)
- Kernel discards process

Zombie Example

```
void main ()
{
    if (fork() == 0){        // child process
        // exiting the child process right away
        printf ("Child PID: %d\n", getpid ());
    }
    else{// parent process
        printf ("Parent PID: %d\n", getpid ());
        // do nothing
        while (1);
    }
}
```

```
[tanzir@compute forkdemos]$ ./a.out
```

```
Parent PID: 4902
```

```
Child PID: 4903
```

```
^Z
```

```
Suspended
```

```
[tanzir@compute forkdemos]$ ps -l
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	21956	3476	3475	0	80	0	-	31643	sigsus	pts/2	00:00:00	tcsh
0	T	21956	4902	3476	28	80	0	-	1040	signal	pts/2	00:00:02	a.out
1	Z	21956	4903	4902	0	80	0	-	0	exit	pts/2	00:00:00	a.out <defunct>
0	R	21956	4904	3476	0	80	0	-	37233	-	pts/2	00:00:00	ps

```
[tanzir@compute forkdemos]$
```

- ps shows child process as “defunct”
- Terminated child continue to take resources in Zombie state
- This state is over when
 - Parent is killed, or
 - Parent performs wait ()
- Killing parent allows child to be reaped
- There is no Zombie processes in WSL (aka. Ubuntu Subsystem under Windows)
 - So, WSL is not supported for this PA

Handling Background Processes

First things is, we cannot issue `wait()`

- That will get your shell stuck on the bg process
- Example: “sleep 5” should take 5 seconds to run and give shell back
- On the other hand, “sleep 5 &” should take just 0 seconds, because it is running in the bg

However, not using `wait()` would lead to Zombie processes

How to get around this problem?

Solution:

1. Collect the pid of each background process in a list
2. Periodically (e.g., before each command) call `waitpid ()` function in a non-blocking manner on each pid in the above
 - Blocking `waitpid()` will get the shell process stuck
3. Remove the pids that are done from the list