PROGRAMMING ASSIGNMENT # 2 – SUPPLEMENTARY DISCUSSION

Tanzir Ahmed

CSCE 313 Fall 2018

Programming Assignment # 2

- Main Challenges (Still after fork + exec)
  - *I/O Redirection*
  - *Piping*
- To solve these challenges, we need the topics:
  - *File I/O basics*
  - *Inter Process Communication using **pipes***

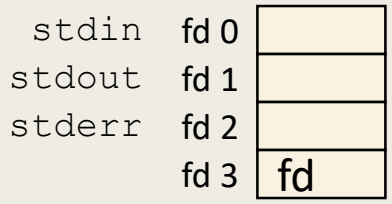
File I/O – File Descriptors

```
int main ()
{
      int fd;  // file descriptor
      char* buf [] = "file content";
      fd = open ("foobar.txt", O_CRAEATE|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]



/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(4,1)`

fd 0	
fd 1	a
fd 2	
fd 3	
fd 4	b

Descriptor table
after `dup2(4,1)`

fd 0	
fd 1	b
fd 2	
fd 3	
fd 4	b

Implementing “ls –la>foo.txt”

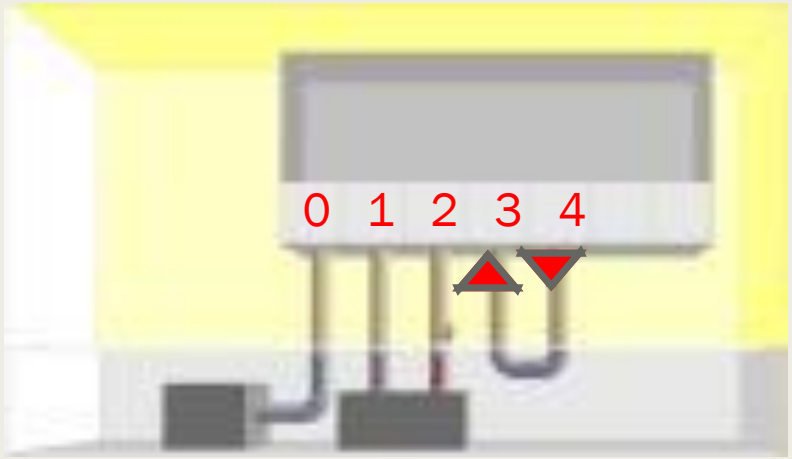
```
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
int main ()
{
        int fd = open ("foo.txt", O_CREAT|O_WRONLY,
        S_IRUSR | S_IWUSR);
        dup2 (fd, 1); // overwriting stdout with the new file
        execlp ("ls", "ls", "-l", "-a", NULL); // now execute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

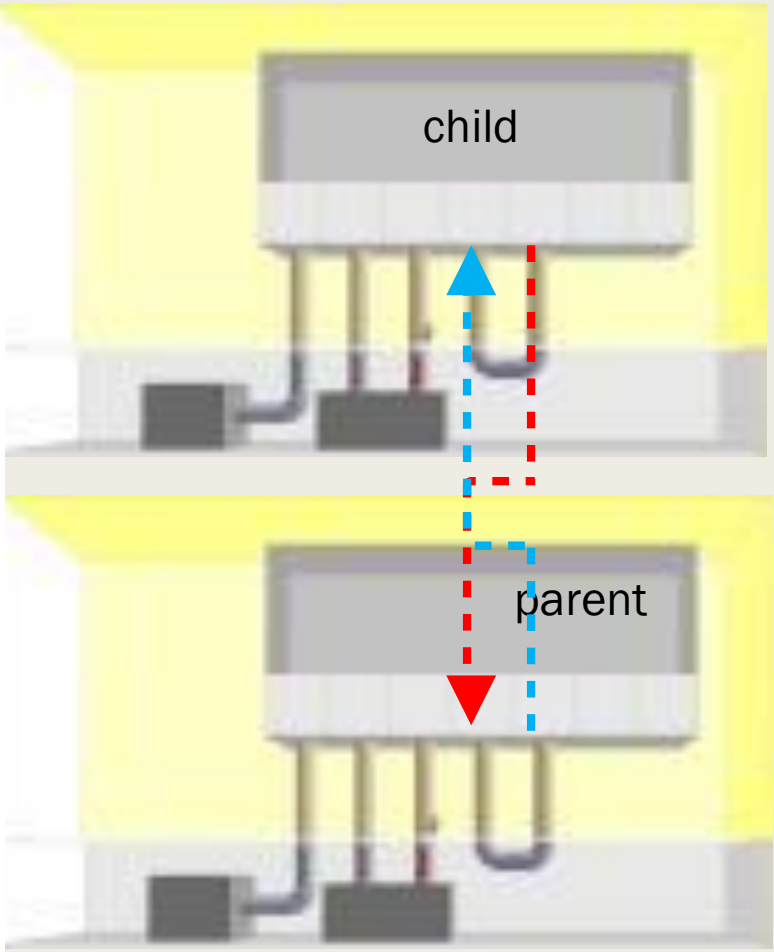
```c
#include <stdio.h>
#include <unistd.h>

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: Sending %s\n",
msg);
         write (fds [1], msg,
strlen(msg)+1);
    }else{
         char buf [100];
         read (fds [0], buf, 100);
         printf ("PARENT: Received %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", "soda", "-l", NULL);
      }
}
```

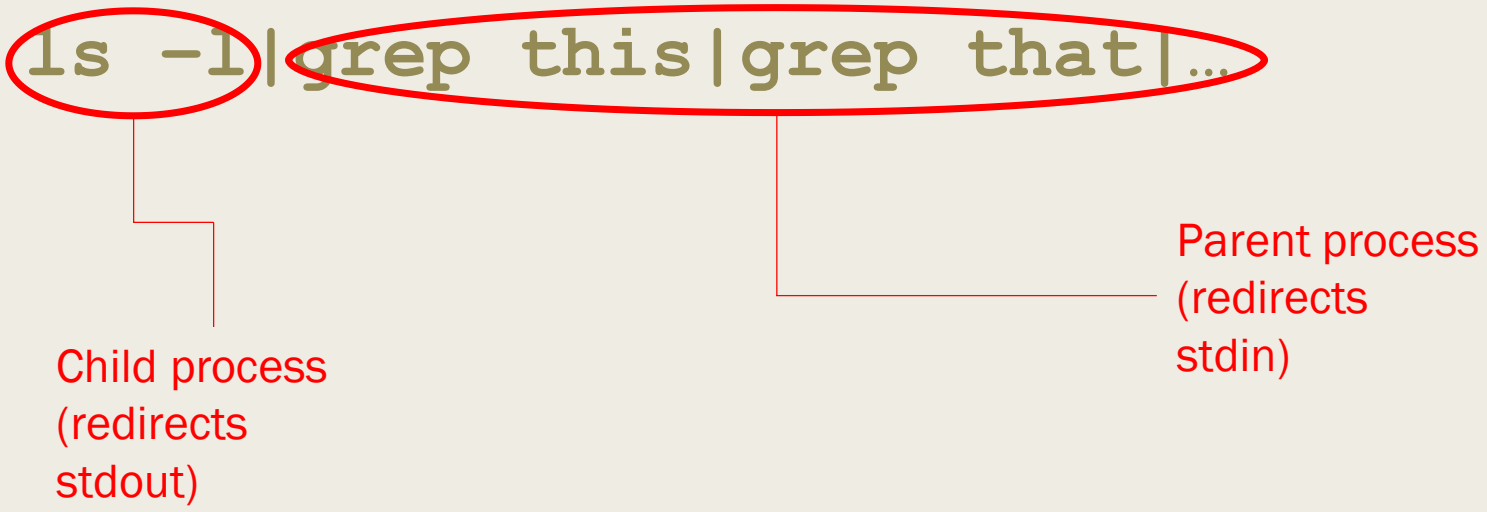
Step 3

Step 1

Step 2

Difficulty with the Previous

- You have to hard code the stages
  - *2 pipes (3 pipe separated portions) will change the code completely: cannot generalize this code*
  - *Will require us to have multiple pairs of file descriptors*
- Can we think of a more general way of doing the same?



A General Pipe Portion

```
int fd [2];
pipe (fd);
if (fork() == 0){
      dup2 (fd[1], 1); // overwriting stdout to the pipe's WRITE end
      close (fd[0]); // close unused pipe end
      execute (portion[i]);
}else{
      wait (0);
      close (fd[1]); // close unused pipe end
      dup2 (fd [0],0); // overwriting stdin to pipe's READ end for the later portions
 }
```

the later portions?

- Shouldn't the #of fd's be 2*(#of portions)
- Also, should the last portion do the same thing (i.e., redir its stdout)?
- You also need to close the unused pipe ends
  - *Otherwise, your program will deadlock*

Over All – Repeat the Following

- Read the line full of command
- Split the line into portions by “|”
- For each portion except the last:
  - *Pipe()*
  - *Fork() a child*
  - *Redirect the child’s STDOUT to Write End of pipe*
  - *Execute the current portion under this child*
  - *Under the parent, just redirect the STDIN to the Read End of the pipe*