Forks, Execs, Pipes, Dup2 …

Additional Materials and Exercises

2310

Sem 1, 2020

Week 7.5

~ half the course down, half to go

- 2nd half is more difficult, but you are better than you were when you started

- 2 assignments complete
- 3rd assignment out
  (See last week for modular coding & makefiles - very important for A3)
- Final Exam ~2 months away

Additional learning materials

- These slides will provide a refresher on the following concepts to help you with Assignment 3 and the final exam:
  - Processes
  - Forking & Execing
  - Piping & Dup2
- Process diagrams will be provided alongside code snippets to understand execution flow
- Exercises at the end are designed to reinforce your understanding and **prepare you for Assignment 3**

Forks: Process diagrams

```
void fork4() {
   printf("L0\n");
   if (fork() != 0) {
      printf("L1\n");
      if (fork() != 0) {
         printf("L2\n");
         fork();
      }
    }
   printf("Bye\n");
}
```

What will be the output of this code snippet?

- *Try to do this without a compiler!*

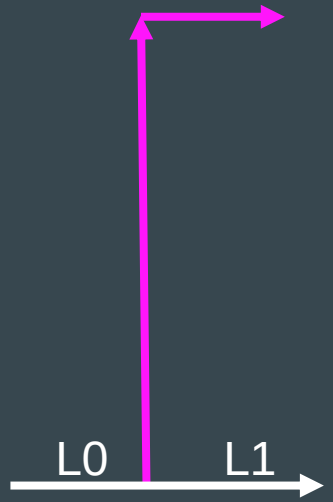
What will the process diagram look like?

Forks: Process diagrams

```
void fork4() {
   printf("L0\n");
   if (fork() != 0) {
      printf("L1\n");
      if (fork() != 0) {
         printf("L2\n");
         fork();
      }
    }
   printf("Bye\n");
}
```

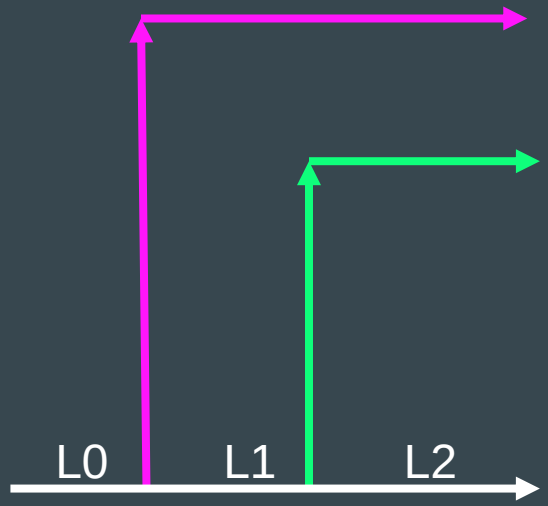
Forks: Process diagrams

```
void fork4() {
   printf("L0\n");
   if (fork() != 0) {
      printf("L1\n");
      if (fork() != 0) {
         printf("L2\n");
         fork();
      }
   }
   printf("Bye\n");
}
```



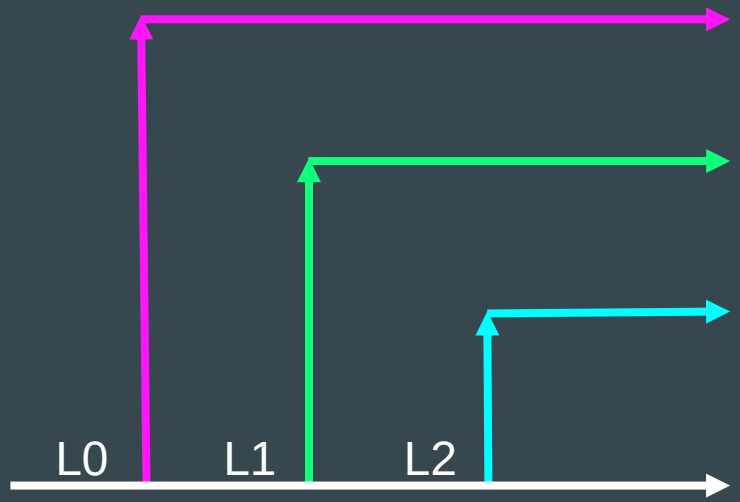
Forks: Process diagrams

```
void fork4() {
   printf("L0\n");
   if (fork() != 0) {
      printf("L1\n");
      if (fork() != 0) {
         printf("L2\n");
         fork();
      }
   }
   printf("Bye\n");
}
```



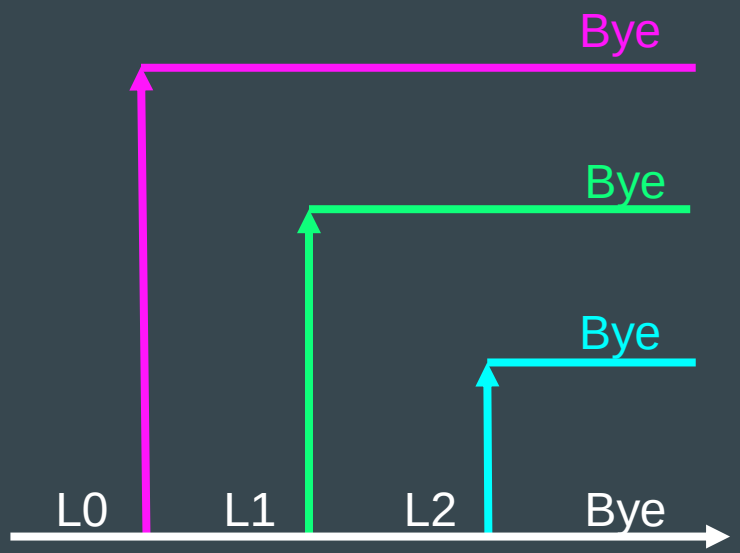
Forks: Process diagrams

```
void fork4() {
   printf("L0\n");
   if (fork() != 0) {
      printf("L1\n");
      if (fork() != 0) {
         printf("L2\n");
         fork();
      }
    }
   printf("Bye\n");
}
```



Forks: Process diagrams

```
void fork4() {
   printf("L0\n");
   if (fork() != 0) {
      printf("L1\n");
      if (fork() != 0) {
         printf("L2\n");
         fork();
      }
    }
   printf("Bye\n");
}
```

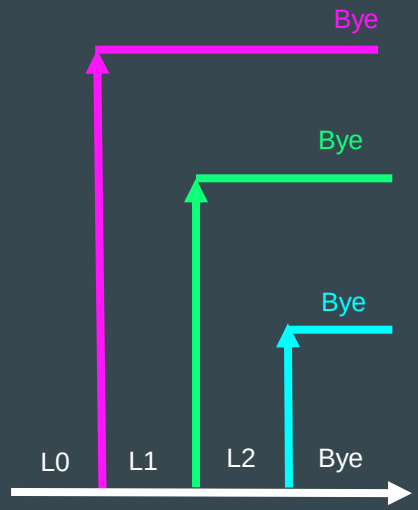


Forks: Process diagrams

```
void fork4() {
   printf("L0\n");
   if (fork() != 0) {
      printf("L1\n");
      if (fork() != 0) {
         printf("L2\n");
         fork();
      }
    }
   printf("Bye\n");
}
```

Things to note:

- 0 is returned to the child process, their process ID (PID, non-zero) is returned to the parent.
- The L*n* prints occur only once, in the parent, as they are specified to occur only when fork != 0.
    - Contrarily, ‘Bye’ is printed for all.



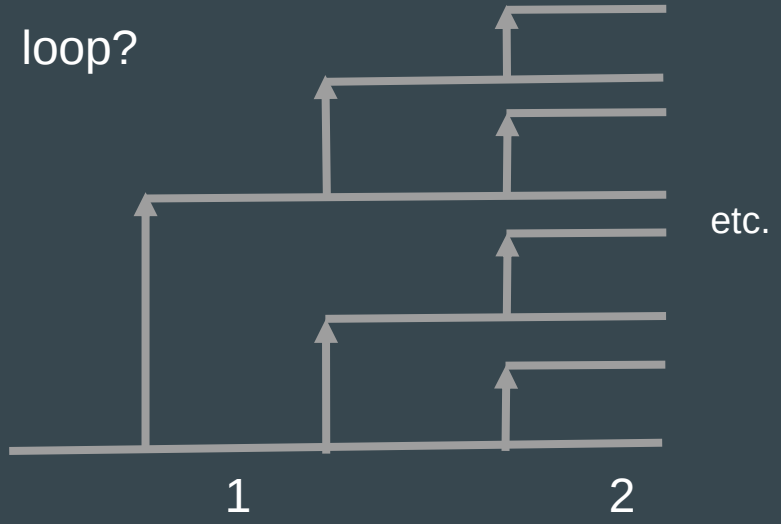
Fork refresher

Consider the following code which process A executes. Assume all fork calls succeed.

```
int main() {
        int k = 0;
        // don't try this on moss

        while(k < 20) {
                fork();
        }

}
```

How many processes are **created** after 20 iterations of the loop?

How many **children** does process A have after 20 iterations of the loop?



Fork refresher answers

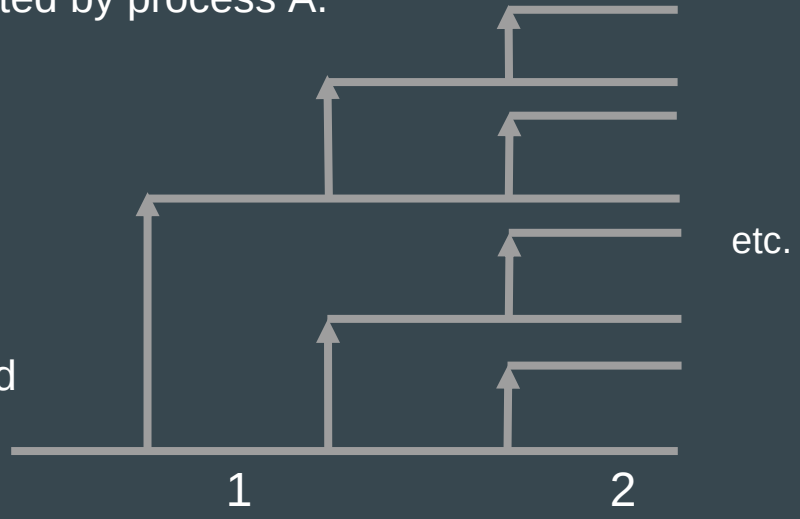
This code is a fork bomb, since k is never incremented it will fork forever.

How many processes are **created** after 20 iterations of the loop?

Each iteration the number of processes doubles. So there are 2^20 processes in total after 20 iterations, but only 2^20 - 1 are created by process A.

How many **children** does process A have after 20 iterations of the loop?

Note that this does not include grandchildren or further descendants of process A. After 20 iterations of the loop, Process A calls fork 20 times in total, and so it will have 20 children in total.



Oops, I ran the code from the last slide. Help!

- Don't Panic! But a “Fork Bomb” will most likely stop you from being able to log in. This is because there's a cap on how many processes each user is allowed to have.
- If you’re still logged in then kill it yourself if you can
  - pgrep [name] | xargs kill
- Lodge a ticket with the EAIT Helpdesk helpdesk@eait.uq.edu.au
  - Include your student number and information on what's happened.
  - Wait for a response from them about the resolution.
  - Course staff can do nothing to accelerate this process.

What process diagram does the following code produce?

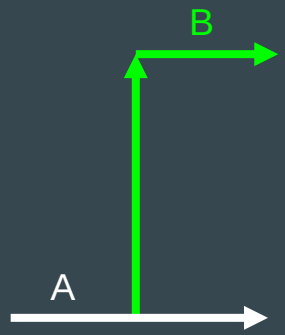
```
printf("A\n");

if (fork() == 0) {
        printf("B\n");
}

if (fork() == 0) {
        printf("C\n");
}

while (wait(NULL) >= 0);

printf("Bye\n");
return 0;
```

What process diagram does the following code produce?

```
printf("A\n");

if (fork() == 0) {
        printf("B\n");
}

if (fork() == 0) {
        printf("C\n");
}

while (wait(NULL) >= 0);

printf("Bye\n");
return 0;
```

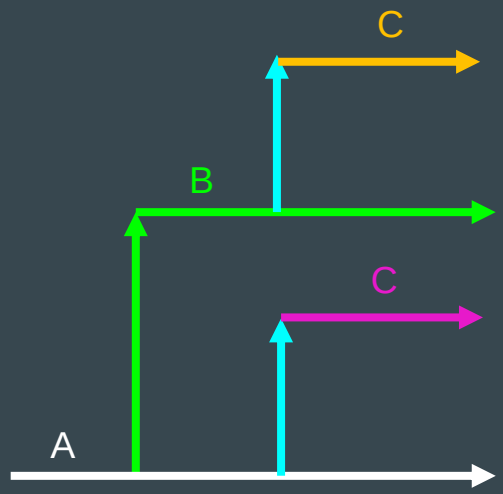
What process diagram does the following code produce?

```
printf("A\n");

if (fork() == 0) {
        printf("B\n");
}

if (fork() == 0) {
        printf("C\n");
}

while (wait(NULL) >= 0);

printf("Bye\n");
return 0;
```



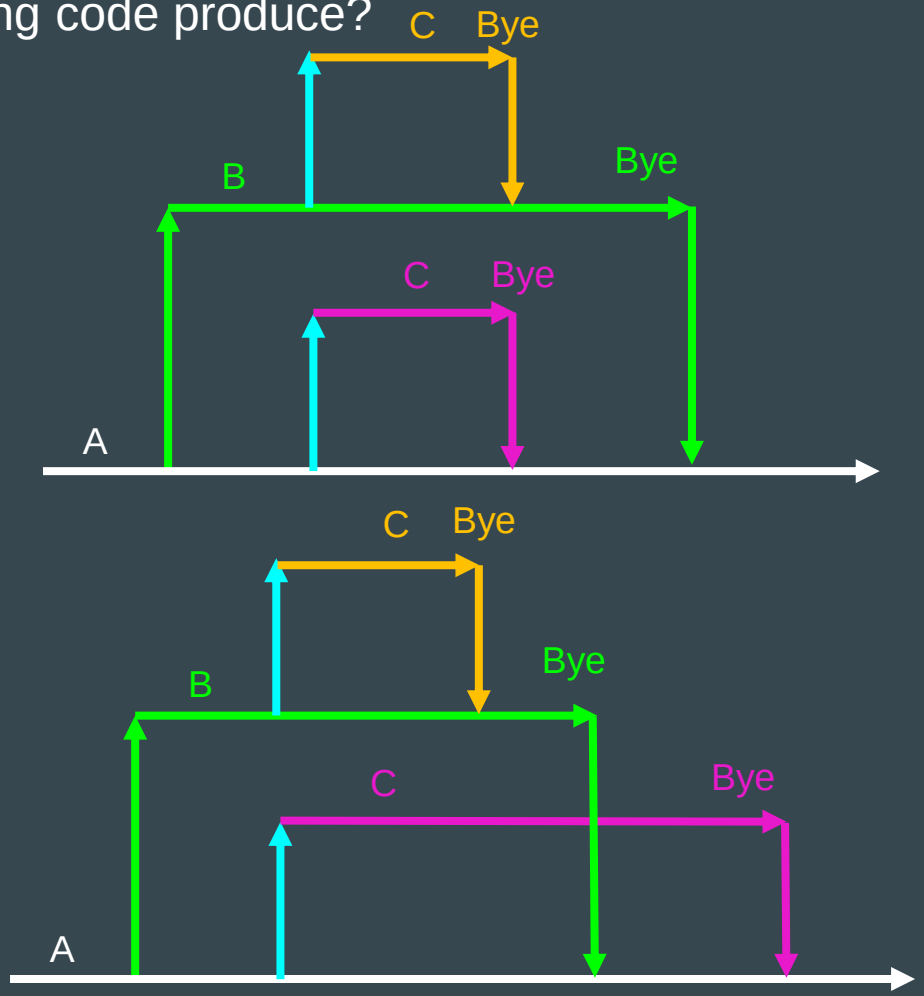
What process diagram does the following code produce?

```
printf("A\n");

if (fork() == 0) {
        printf("B\n");
}

if (fork() == 0) {
        printf("C\n");
}

while (wait(NULL) >= 0);

printf("Bye\n");
return 0;
```



What process diagram does the following code produce?

```
printf("A\n");

if (fork() == 0) {
        printf("B\n");
}

if (fork() == 0) {
        printf("C\n");
}

while (wait(NULL) >= 0);

printf("Bye\n");
return 0;
```

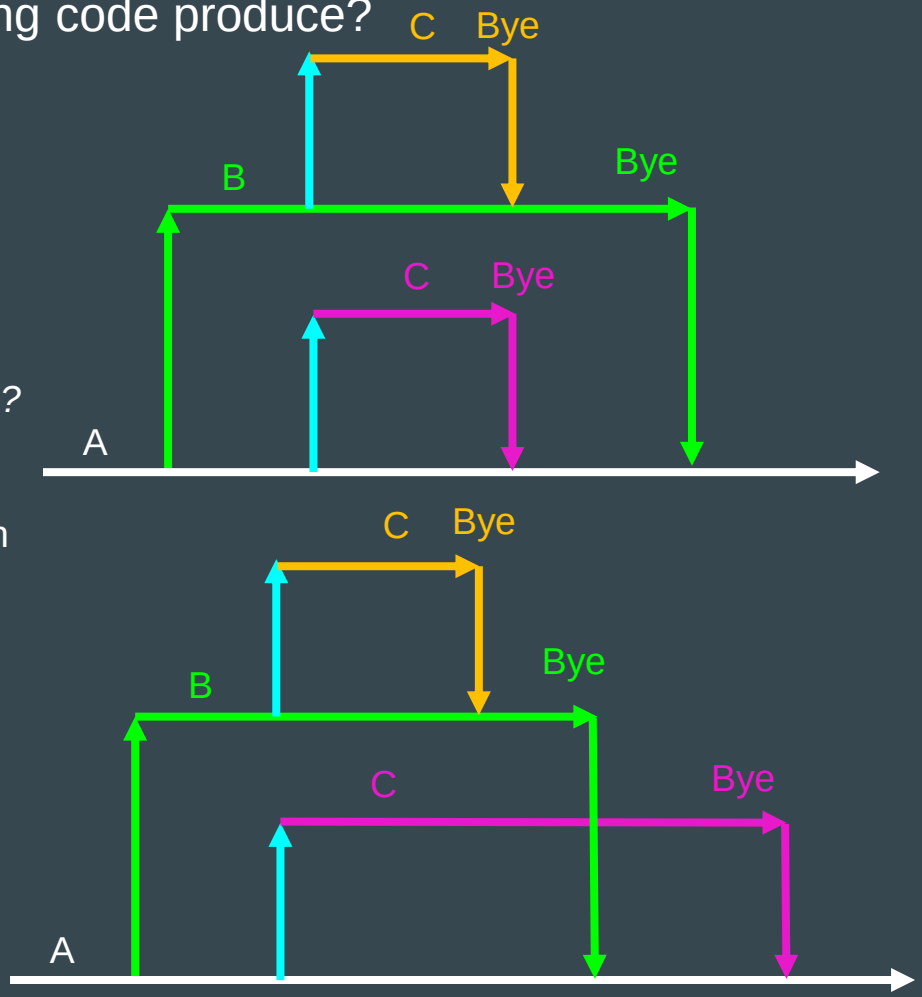


What process diagram does the following code produce?

```
printf("A\n");

if (fork() == 0) {
                printf("B\n");
}

if (fork() == 0) {
                printf("C\n");
}

while (wait(NULL) >= 0);

printf("Bye\n");
return 0;
```

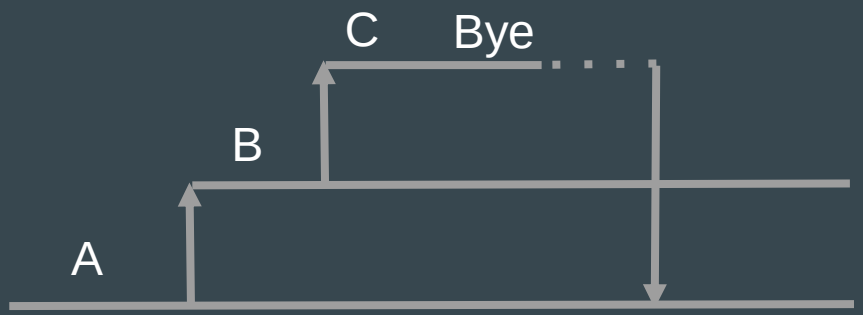
Why is it non-deterministic?

Process C and process B can join in either order with process A, which is why there are two different diagrams drawn.

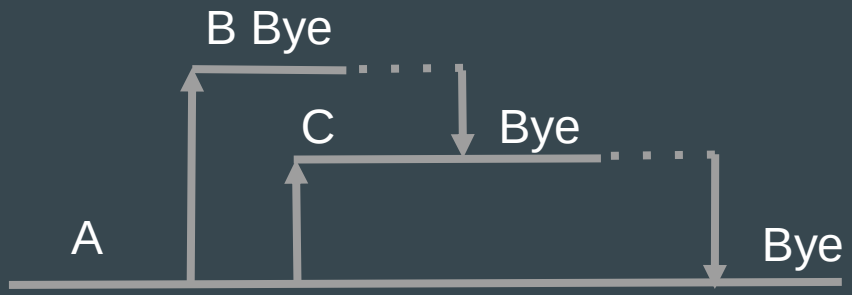


Which of the following are possible with fork?

a)

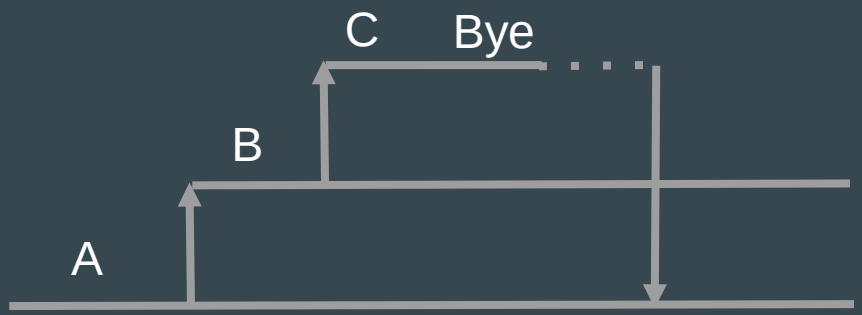


b)



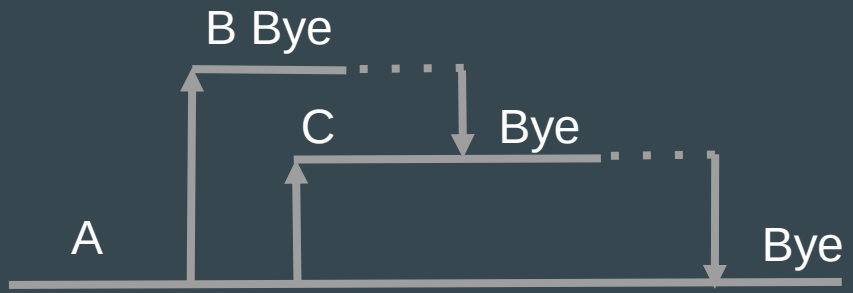
Which of the following are possible with fork?

a)



Not possible. Wait can only wait for child processes, and C is not our child.

b)



Also not possible for the same reason.

Exec

execv()

execvp()

execve()

execvpe()

execl()

execlp()

execle()

<u>These explain what parameters the exec command takes in:</u>

l – arguments directly in call (list)
v – arguments in array (vector which is null terminated)
p – use PATH to find program (“acts” like terminal)
e – provide environment definition

pid_t wait(int* wstatus);

- Suspends current process until **a** child terminates
- Writes info on terminated child into wstatus (if not NULL)
  - Many macros exist to get detail from this terminated child from wstatus,
    - e.g. WIFEXITED(wstatus), WEXITSTATUS(wstatus)
  - `man wait` for more
- Returns process ID of terminated child on success, -1 on error (also sets errno). Example of an error - parent has no children

```
fork();

wait(NULL);
return 0;
```

int pipe(int pipefd[2]);

Need: #include <unistd.h>

Creates a pipe between two new file descriptors, which are stored into the parameter array.

Read end: pipefd[0]

Write end: pipefd[1]

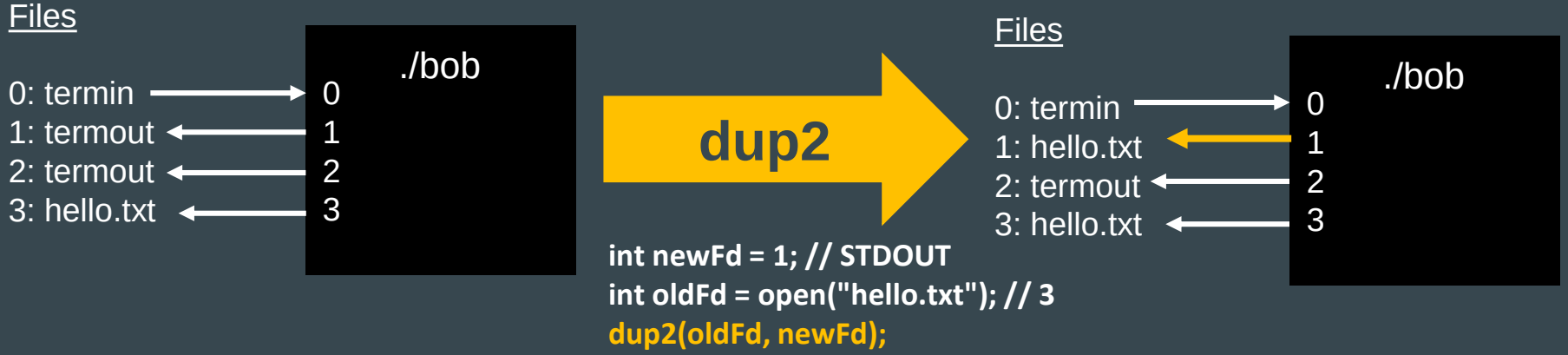
Only have 1 process using each end - don’t try to read/write from multiple places.

dup2(oldFd, newFd)

Std file descriptors:

0: stdin	./bob < hello.txt	(pipe hello.txt into stdin of program bob)
1: stdout	./bob > hello.txt	(pipe the stdout of bob into hello.txt)
2: stderr	./bob 2> save.err	(pipe the stderr of bob into save.err)

Redirecting stdout to file




```
int newFd = 1; // STDOUT
int oldFd = open("hello.txt"); // 3
dup2(oldFd, newFd);
```

https://www.youtube.com/watch?v=EqndHT606Tw

But fd’s aren’t friendly - please use FILE *

Open your file descriptors as FILE pointers:

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

Now use your friendly set of:

fprintf, fflush, fgetc, fgets, ...

And remember to flush

```
int oldFd = 1; // stdout
FILE* file = fdopen(oldFd, “w”); // Can now write to stdout
```

testa3.sh says I have long running processes?

- Sometimes things get left as zombies, or get left running
- Be considerate to your fellow students
- Kill them, please
- It might make your moss login faster

Exercises

1) Write a program that forks and prints the following messages to the display

parent's stderr: "Hello i am the parent"
childs stdout: "My name is bob"

It may be handy to run your program as follows `./test > child.txt 2> parent.txt`

2) Write a program that when executed will run the ls program will all the parameters given to your program.
3) Write a program that when executed will run the ls program with all the parameters given to your program but the output is put into a file called log.txt. (Note: don't add any new parameters to ls, you'll have to use dup2)

A3 Exercise (Individual learning focused for Assignment 3 practice): **(Hint: see the files and pipes lectures)**

1) Write a program called `mock` which when launched will start a child process that will exec into the `lolcat` program which will read from stdin and print to stdout. The parent of the program then should read from stdin and send every character to the child with every second character capitalised.
2) Write a program called `quick` which when launched starts a child process that will exec into the `sand` program, which will read from stdin and print to stdout. `quick` should read from stdin and send every character to `sand`. `sand` should send all characters back capitalised, and `quick` prints them to it's own stdout.