15-213 Recitation
Processes and Shells

Your TAs

Friday, Oct 31st

Reminders

- `malloc` deadlines:
  - Checkpoint: ***Oct 28th (last Tuesday)***
  - Final: ***Nov 4th (Tuesday)***
- `tshlab` will be released on ***Nov 4th***
- Code Reviews:
  - Checkpoint: Heap Checker Quality
  - Final: All of `mm.c`

Agenda

- Shell Lab Preview
  - Shell Demo
- Processes
  - Process Lifecycle
  - Process Graphs
- Error-handling

Shell Lab

Shell Lab

- `tshlab` will be released on ***Nov 4th***
- You'll write a simple shell, complete with:
  - Foreground and background jobs
  - Process management
  - I/O redirection
- Getting Started:
  - CS:APP Chapter 8

Review: Foreground vs Background

- The shell waits for **foreground** jobs to complete, such that all user input goes to the task while the task is still running.
  - eg) when you run **`./mdriver`** to test **`malloclab`**
- **Background** jobs run independently of the main application, allowing for users to interact with the shell while the task runs
  - eg) your browser running when minimized
  - eg) you could also run **`./mdriver`** as a background task!

Shell Demo

If you want to follow along…

- Log into a Shark machine, then type:

```
$ wget http://www.cs.cmu.edu/~213/activities/f25-rec10.tar
$ tar -xvf f25-rec10.tar
$ cd rec10-handout
$ make
```

Shell Demo: Recap

What did we see?

Process Lifecycle

- `./demo` – created a new process, and reaped it on exit.
- `ctrl + z` – pauses foreground process
- `./demo … &` – "`&`" runs process in background

Today

I/O Redirection

- `./demo < in.txt` – take input from a file
- `./demo > out.txt` – created a new file, and wrote output to it!

Next time

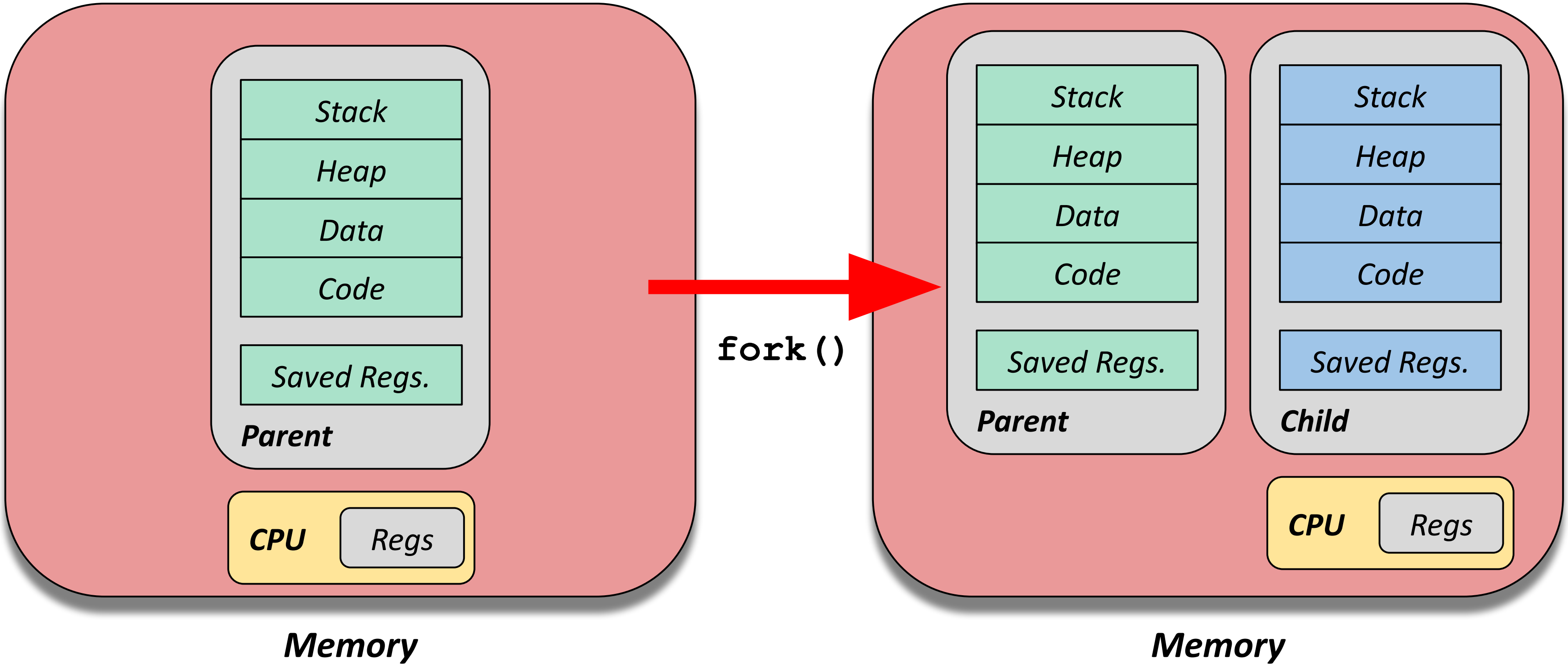
You'll be implementing all of these features in Shell Lab!

Processes

Life Cycle of a Process

- `fork()`
  - Creates a new child process
- `execve()`
  - Load and run a new program, replacing the current one
- [... Do some work]
- `exit()`
  - End the running program
- `waitpid()`
  - Parent reaps terminated children

`fork()`: Creating a New Process



- Child gets *duplicate* but *separate* copy of address space.
- File descriptors are still shared!

fork() Example

```
int main(int argc, char ** argv) {
    pid_t pid;
    int *x = malloc(sizeof(int));
    *x = 1;

    pid = Fork();
    if (pid == 0) {
        *x += 1;
        printf("[%d] child: x = %p, *x = %d\n", getpid(), x, *x);
        return 0;
    }

    *x -= 1;
    printf("[%d] parent: x = %p, *x = %d\n", getpid(), x, *x);
    return 0;
}
```

- Suppose **`x`** is stored at address **`A`**. What are the different possible outputs?

`fork()` Example: Solution

```
[<child pid>] child: x = A, *x = 2
[<parent pid>] parent: x = A, *x = 0
```

or

```
[<parent pid>] parent: x = A, *x = 0
[<child pid>] child: x = A, *x = 2
```

- In this example, calls to **`printf`** can happen in any order.
- Child and parent have different PIDs
- Same virtual address, different values.

`execve()`: Loading and Running a Program

```
int execve(char *pathname, char *argv[], char *envp[]);
```

- Loads and runs program specified by **`pathname`**:
  - With arguments **`argv`**, environment **`envp`**
- If successful:
  - Overwrite code, data, stack, and start executing!
  - *Calls once, never returns!*
- On failure, return **`-1`**, and set **`errno`**.

`execve()`: Example

```
int main(void) {
    char *args[3] = {
        "/bin/echo", "Hi 18213!", NULL
    };

    execve(args[0], args, environ);
    printf("Hi 15213!\n");
    exit(0);
}
```

- What does this program print? Assume **`/bin/echo`** exists.
    - ○ **`“Hi 18213!”`**

execve(): Example

```
int main(void) {
    char *args[3] = {
        "/bin/blahblah", "Hi 15513!", NULL
    };

    execve(args[0], args, environ);
    printf("Hi 14513!\n");
    exit(0);
}
```

- What does this program print? Assume **/bin/blahblah** does *not* exist.
  - ○ **“Hi 14513!”**

Recall: Terminating and Reaping

- **`void exit(int status)`**
  - Terminates the current program
  - Called once, never returns
- Terminated processes still consume system resources!
- Parent process is responsible for *reaping* them:
  - **`wait`**
  - **`waitpid`**

```
$ ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640

$ ps
PID TTY TIME CMD
6585 ttyp9 00:00:00 tcsh
6639 ttyp9 00:00:03 forks
6640 ttyp9 00:00:00 forks
<defunct>
6641 ttyp9 00:00:00 ps
```

`wait()` vs. `waitpid()`

Textbook: p743

```
pid_t wait(int *status)

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

- **`wait`**
  - Blocks until *any* child exits.
  - Returns PID of child, stores exit status at specified address.
- **`waitpid`**
  - **`pid = -1`** – wait for *any* child
  - **`pid > 0`** – wait for *specific* child
  - Can use **`options`** argument to configure behavior, e.g. to return immediately if there are no children to reap.

Exit Values Convey Information

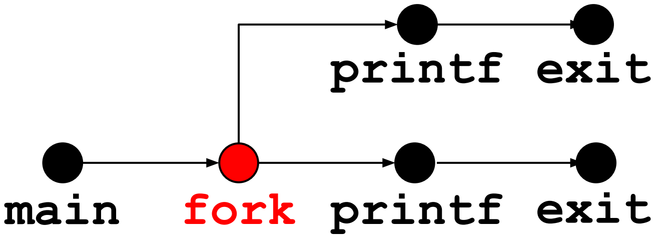
```
int main(void) {
    pid_t pid = fork();
    if (pid == 0) { exit(0x213); }
    else {
        int status = 0;
        waitpid(pid, &status, 0);
        printf("0x%x exited with 0x%x\n", pid, WEXITSTATUS(status));
    }

    exit(0);
}
```

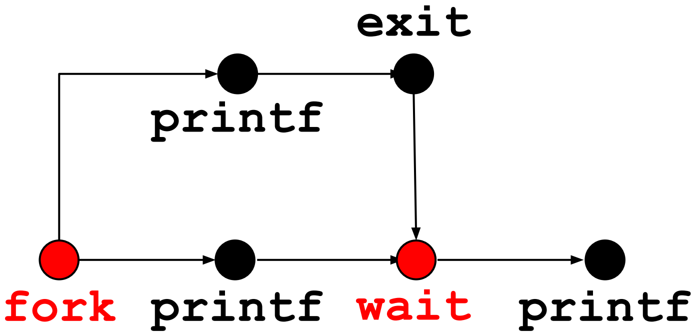
- What does this program print?
  - `“0x7b54 exited with 0x13”`
  - **`WEXITSTATUS()`** returns only 1 byte

Process Graphs

Process Graphs

- *Process Graphs* allow us to reason about the ordering of events across different processes.
- Vertices: execution of an event
- Directed Edge (a -> b): a occurs before b
- **`fork()`** creates a *branch*
- **`wait()`** creates a *join*

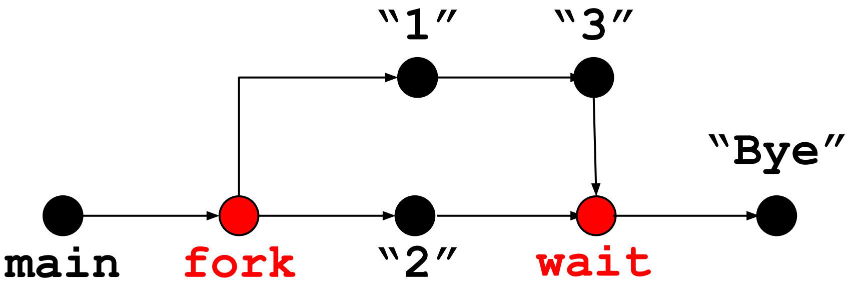




Process Graphs: Example

```
int main() {
    pid_t pid;
    int child_status;

    pid = Fork();
    if (pid == 0) {
        printf("1\n");
        printf("3\n");
        return 0;
    }

    printf("2\n");
    wait(&child_status);

    printf("Bye\n");
}
```

- What does the process graph for this program look like?

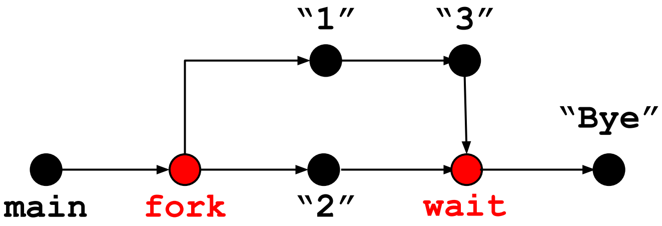


- Now we want to use the graph to answer questions:
  - e.g. "Can this program output 213?"

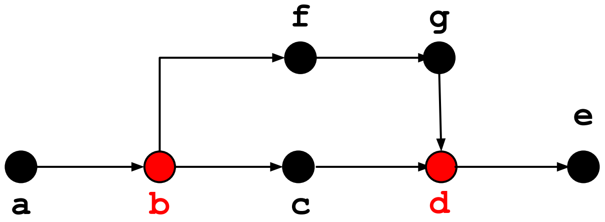
Process Graphs: Reasoning about Orderings

Q: “Is this ordering feasible?”

A: Use the graph!



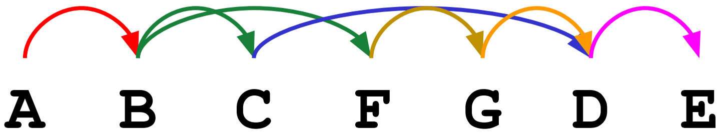
1. *Relabel graph*



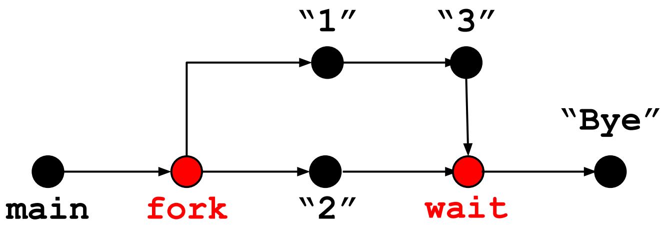
2. *Write out the ordering you want to try:*

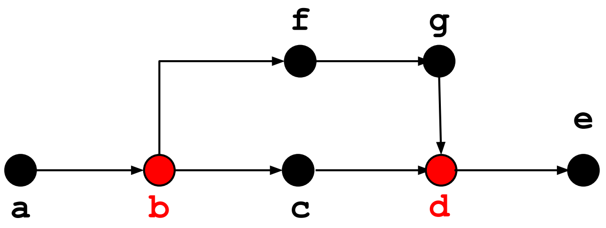
A B C F G D E

3. *Add edges from graph, then check for backward arrows*

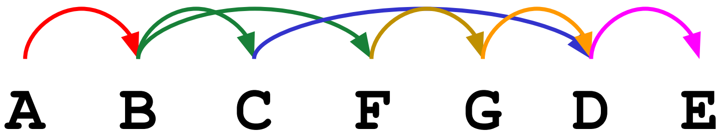


Process Graphs: Reasoning about Orderings





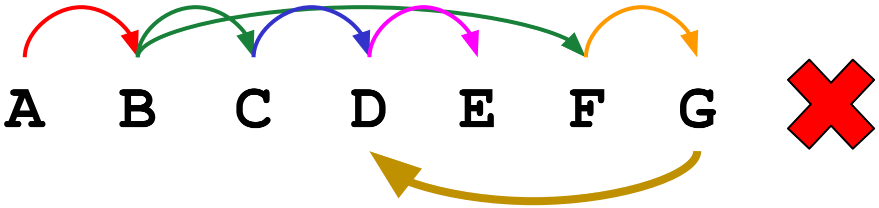
Feasible: no backward arrows



```
2
1
3
Bye
```

What about:

```
2
Bye
1
3
```

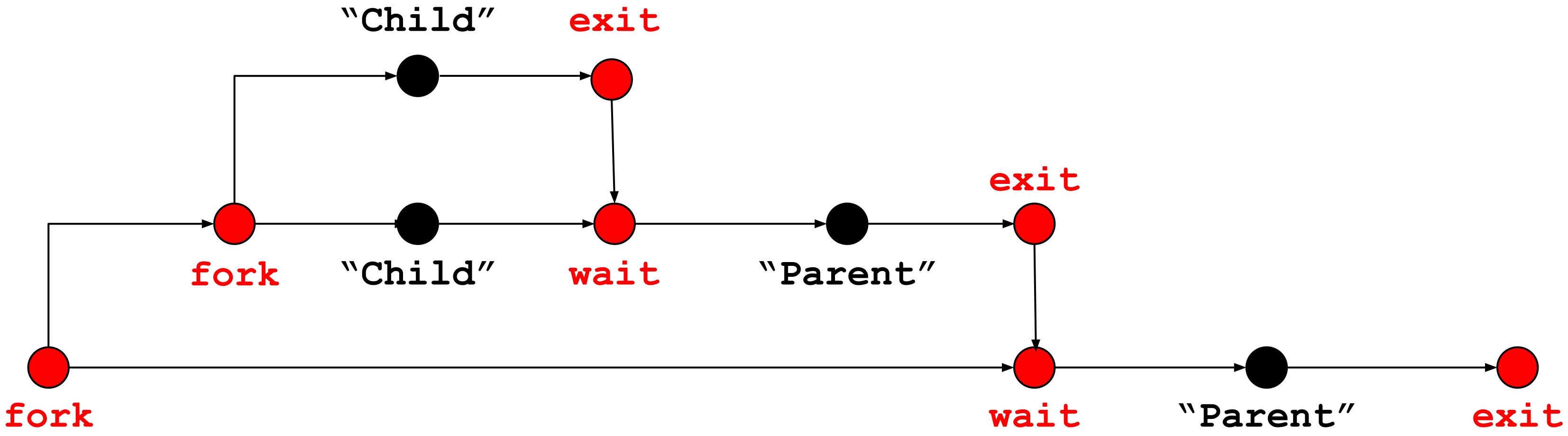


Process Graphs: Harder Example

```
int main(void) {
    int status;
    if (fork() == 0) {
        pid_t pid = fork();
        printf("Child: %d\n", getpid());
        if (pid == 0) {
            exit(0);
        }
        // Continues execution…
    }
    pid_t pid = wait(&status);
    printf("Parent: %d\n", pid);
    exit(0);
}
```

- What does the process graph look like for this example?
- How many unique combinations can be printed?

Process Graphs: Harder Example



```
"Child" (Grandchild)
"Child"
"Parent" (Child)
"Parent"
```

or

```
"Child"
"Child" (Grandchild)
"Parent" (Child)
"Parent"
```

Error Handling

Error Handling

```
int main() {
    int fd = open("213Grades.txt", O_RDWR);
    // Change grades to As or Fs
}
```

- Can syscalls fail?
- How can we tell when they fail?

Error Handling

```
int main() {
    int fd = open("213Grades.txt", O_RDWR);

    if (fd < 0) {
        fprintf(stderr, "Failed to open\n”);
        exit(-1);
    }
    // Change grades to As or Fs
}
```

- Syscalls return **`-1`** on failure, and set **`errno`**.
- How can we tell what specifically went wrong?

Error Handling

```
int main(void) {
    int fd = open("213Grades.txt", O_RDWR);
    if (fd < 0) {
        fprintf(
            stderr,
            "Failed to open %s: %s\n",
            "213Grades.txt",
            strerror(errno)
        );
        exit(1);
    }
    // Change grades to As or Fs
}
```

- **`strerror`** – turns errno codes into printable messages
- **`perror`** (print error) is a handy shorthand

Wrapping Up

- `malloc` Final:
  - Due ***Nov 4th (Tuesday)***
- Getting started on `tshlab`:
  - Textbook, write-up, `man` pages!
- Watch your inbox for code review sign ups.
- Good luck on `malloc` Final :)

The End