

Recitation Week #4

Project 2: A Thorough Overview

Alejandro Cabrera
Operating Systems
COP4610 / CGS5765

Today's Recitation

- Project 2: Assigned
 - Specifications Overview
- Command Line Parsing
- Process Management
 - Process Creation
 - Foreground vs. Background Execution
 - Environment Variables
- Further Advice

Project 2: Implementing a Shell

- Tasked with implementing a shell interface that behaves like tcsh.
- Project divided into multiple parts, each encompassing a logical component of functionality.

Project 2: Specification Overview

Program Main Structure

- The shell is an infinite loop.
- On each iteration, you will:
 - Parse user input.
 - Determine whether anything special needs to happen (e.g., error-handling, I/O redirection, piping, process execution, quitting, etc.)
 - Make that something happen.
 - Reset the state of the shell, so that your buffers are clean for the next iteration.

Program Main Structure

```
while(1)
{
    /* get user input */
}
}
```

Program Main Structure

```
while(1)
{
    /* get user input */
    /* exit if an exit condition is
    triggered */
}
```

Program Main Structure

```
while(1)
{
    /* get user input */
    /* exit if an exit condition is
    triggered */
    /* do something with the input */
}
```

Program Main Structure

```
while(1)
{
    /* get user input */
    /* exit if an exit condition is
    triggered */
    /* do something with the input */
    /* reset shell state */
}
```

Command Line Parsing

- You should first make sure that you are able to read in the user input.
- Double-check that you're reading in what you expect by printing it out.
- You should be reading your command line input from standard in.

Parsing Technique: Multiple Passes

- Parsing is far more difficult if you attempt to do it all in one pass.
- It's much easier if you break it into logical phases.
- Some possible phases:
 - Whitespace stripping
 - Command building phase
 - Path resolution phase
 - Environment variable expansion phase
 - I/O redirection phase
 - Final execution preparation phase

Parsing Example

```
cabrera:/home/cabrera#      ls      -l &  
→      ls      -l &
```

Parsing Example

```
cabrera:/home/cabrera#      ls      -l &
```

```
→      ls      -l &
```

```
(after whitespace stripping)
```

```
→ ls -l &
```

Parsing Example

```
cabrera:/home/cabrera#      ls      -l &
```

```
→      ls      -l &
```

```
(after whitespace stripping)
```

```
→ ls -l &
```

```
(after path resolution)
```

```
→ /bin/ls -l &
```

```
...
```

Parsing Advice

- Use `calloc()` instead of `malloc()`
 - `calloc()` zero-initializes the memory it allocates.
 - If your command line buffer contains garbage values, they will affect the execution of an iteration of your shell.
 - This could have subtle, disastrous effects.
- Treat user-input like it's dangerous (because it is).
- This means:
 - Parse thoroughly – eliminate un-needed whitespace.
 - Parse judiciously – catch and signal errors as early as possible.
- If something gets by that you didn't expect, this will crash your shell.

Process Management

- Process management refers to all the functions available to you in order to create new processes, manage their execution, and terminate them cleanly.
- This includes facilities for communicating between processes.

Process Creation: An Overview

- How do you create a new process without overwriting the existing process?
- You `fork()`, `exec()`, and `waitpid()`.
- `fork()` creates a copy of the shell process with its own address space and its own process ID.
- `exec()` replaces the image of a running process with a new process.
- `waitpid()` has the parent wait until the child is done.

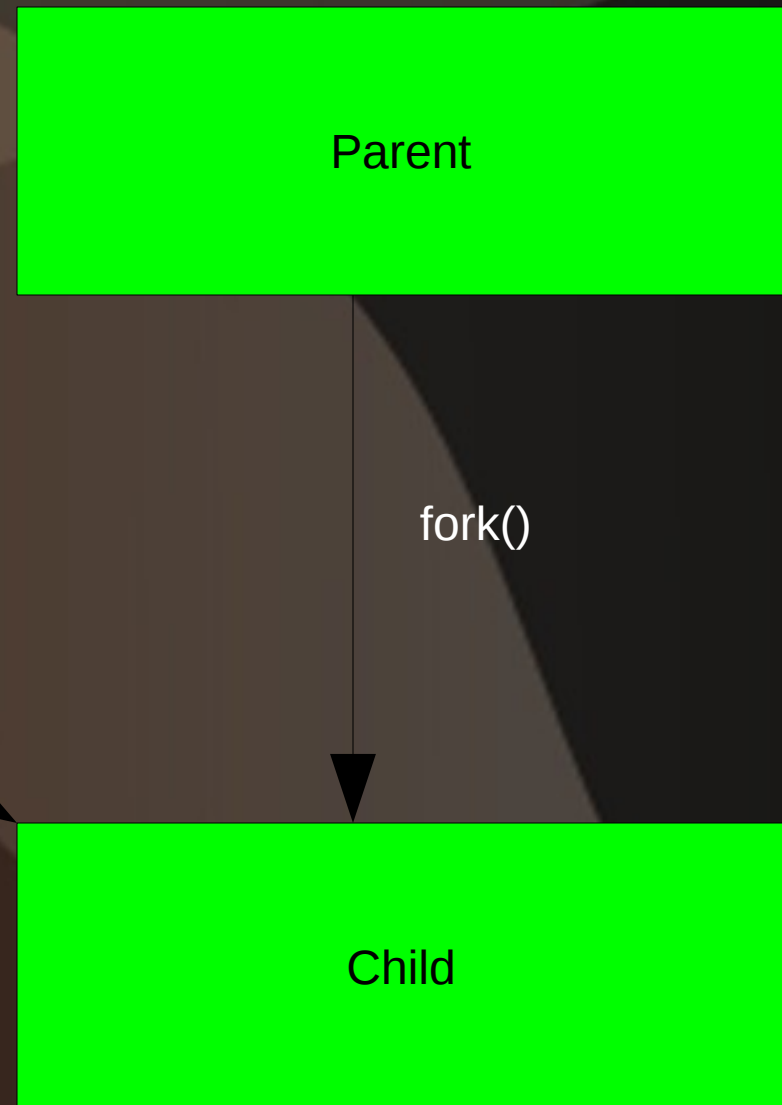
Process Creation: fork()

```
int fork();
```

- Returns pid of child process.
- Has three return values to worry about:
 - $\text{int} > 0$: This is the parent.
 - $\text{int} == 0$: This is the child – it has no child pid.
 - $\text{int} < 0$: An error occurred.

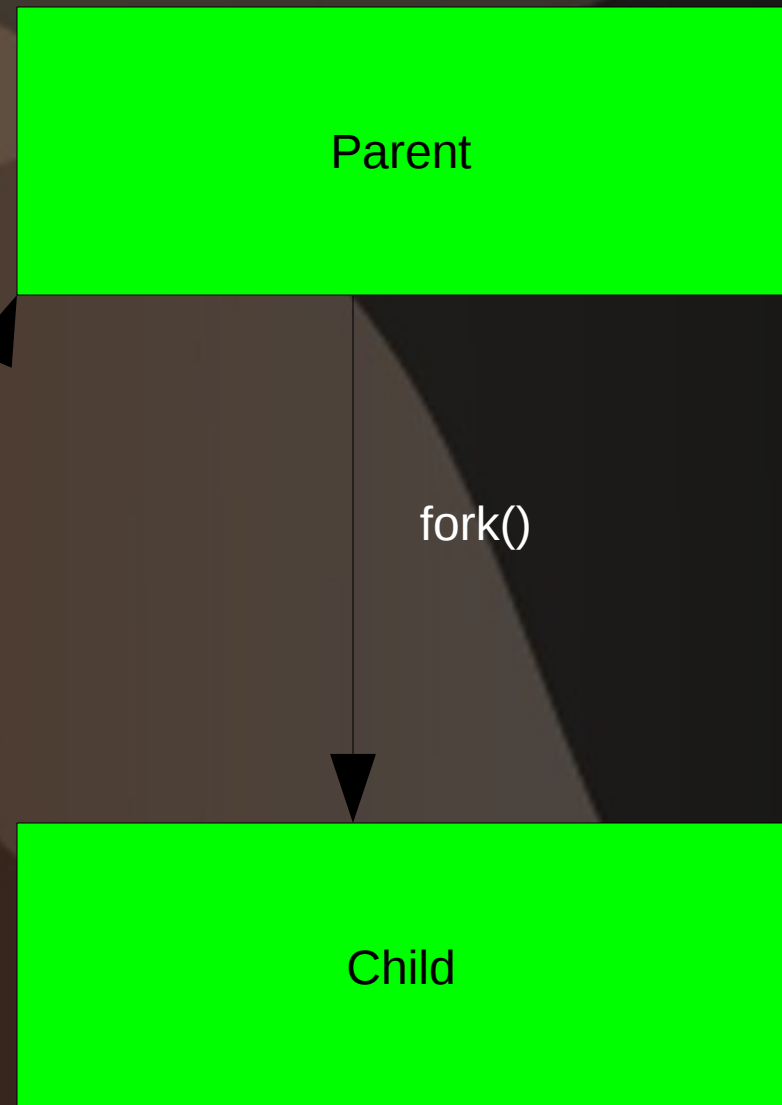
fork() Walkthrough

```
int main()
{
    switch(fork())
    {
        case 0:
            /*child*/
            break;
    }
    . . .
}
```



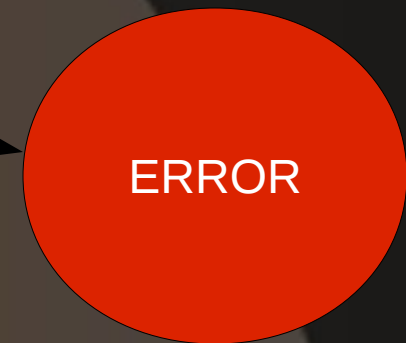
fork() Walkthrough

```
int main()  
{  
    switch(fork())  
    {  
        case 0:  
            /*child*/  
            break;  
        default:  
            /*parent*/  
    }  
    . . .  
}
```



fork() Walkthrough

```
int main()
{
    switch(fork())
    {
        case -1:
            /*error*/
            break;
        case 0:
            /*child*/
            break;
        default:
            /*parent*/
    }
    ...
}
```



exit(1)



Process Creation: `exec*()`

```
int execl(const char *pathname, const char *arg0, ... , (char *) 0);
int execv(const char *pathname, char *const argv[]);
int execlp(const char *pathname, const char *arg0, ...,
           (char *) 0, char *const envp[]);
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ..., (char *) 0);
int execvp(const char *filename, char *const argv[]);
```

- An entire family of execute functions!
- For this project, we only worry about `execv()`.

Process Creation: `execv()`

- `execv()` takes two arguments:
 - A pathname: this means an **absolute** pathname. You must construct this on your own.
 - An argument vector, following C standard.
- Returns 0 on success, 1 otherwise.

Calling `execv()`

```
char *command = "/bin/ls";  
char *argv[] =  
    {"/bin/ls", "-l", (char *) 0};  
  
execv(command, argv);
```

Calling `execv()` **Incorrectly**

```
char *command = "ls";
```

```
char *argv[] =
```

```
    {"/bin/ls", "-l", (char *) 0};
```

```
execv(command, argv);
```

```
...
```

```
ls: no such file or directory.
```

Calling `execv()` **Incorrectly**

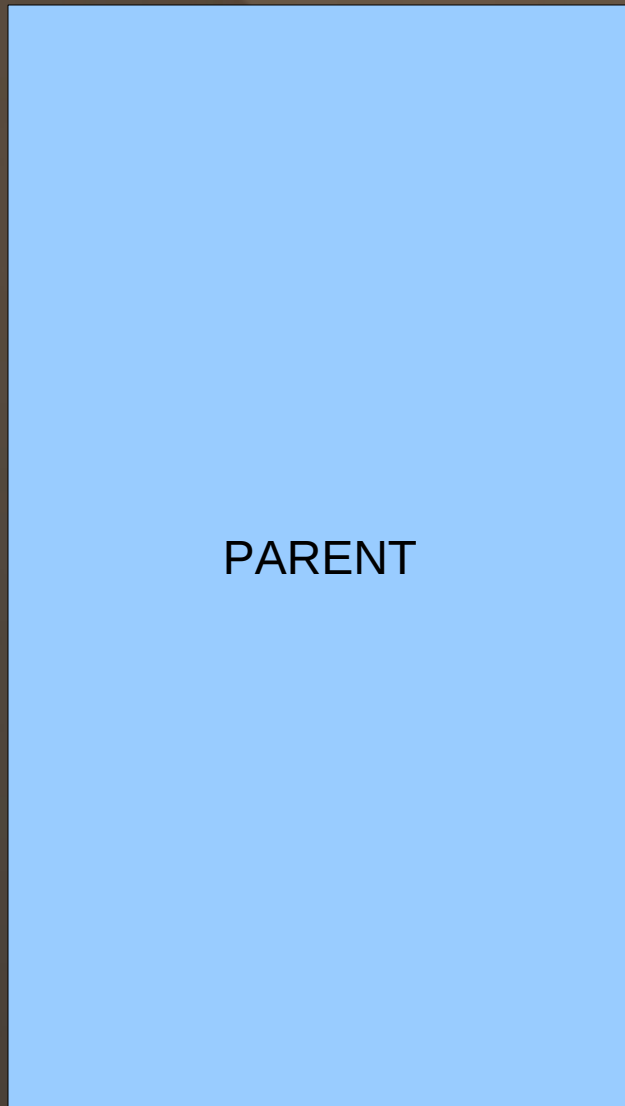
```
char *command = "/bin/ls";  
char *argv[] =  
    {"/bin/ls", "-l"}; /*Missing "\0"*/  
  
execv(command, argv);  
...  
execv: Bad address.
```

Process Creation: waitpid()

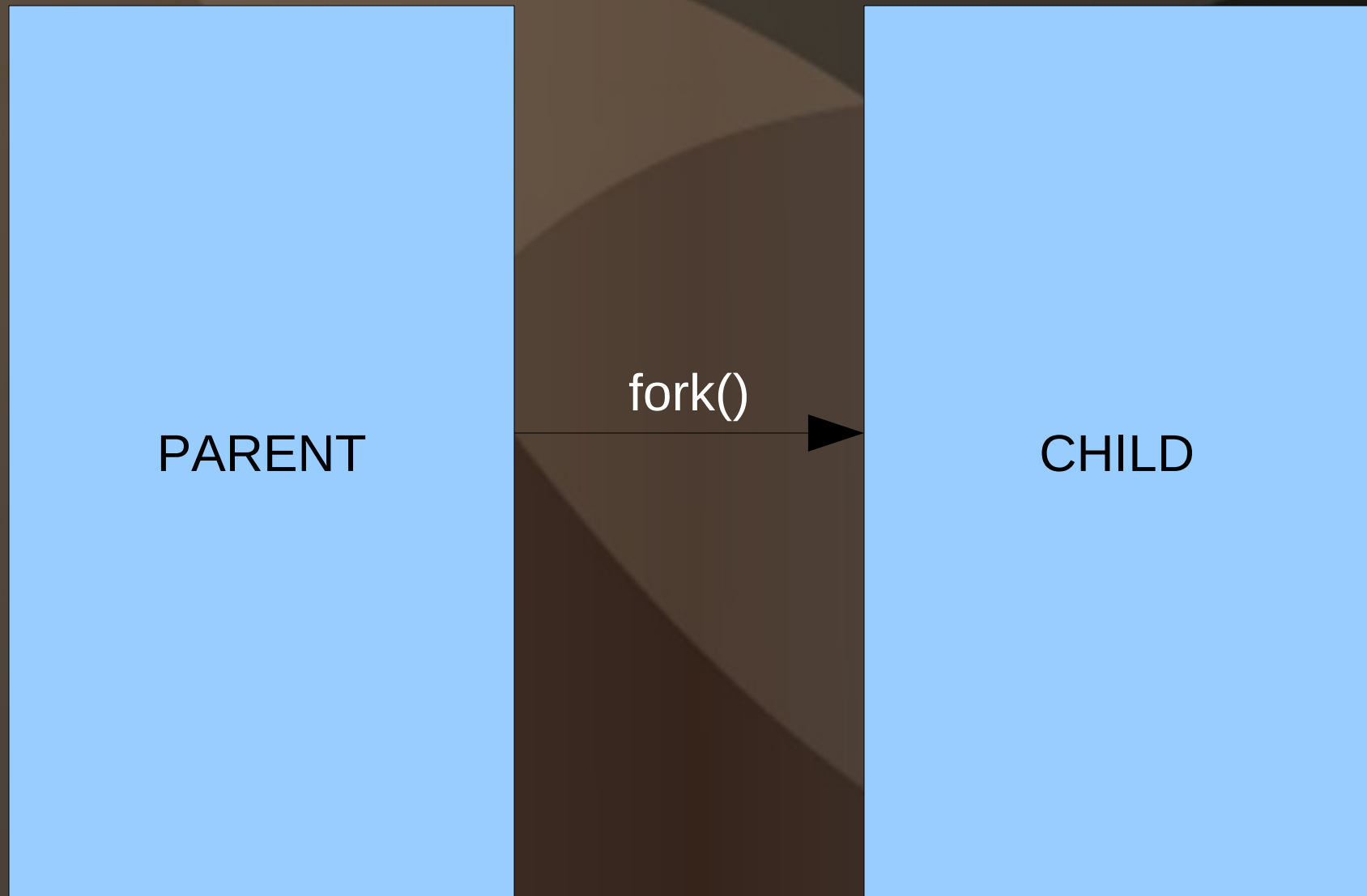
```
int waitpid(const int gpid,  
            int *status,  
            const int FLAGS);
```

- `gpid` – The children the parent will wait for.
 - For this project, we are only interested in `gpid == 0`, the parent waits for only its children.
- `status` – The exit status of the child on termination.
- `FLAGS` – Controls blocking behavior of `waitpid`.

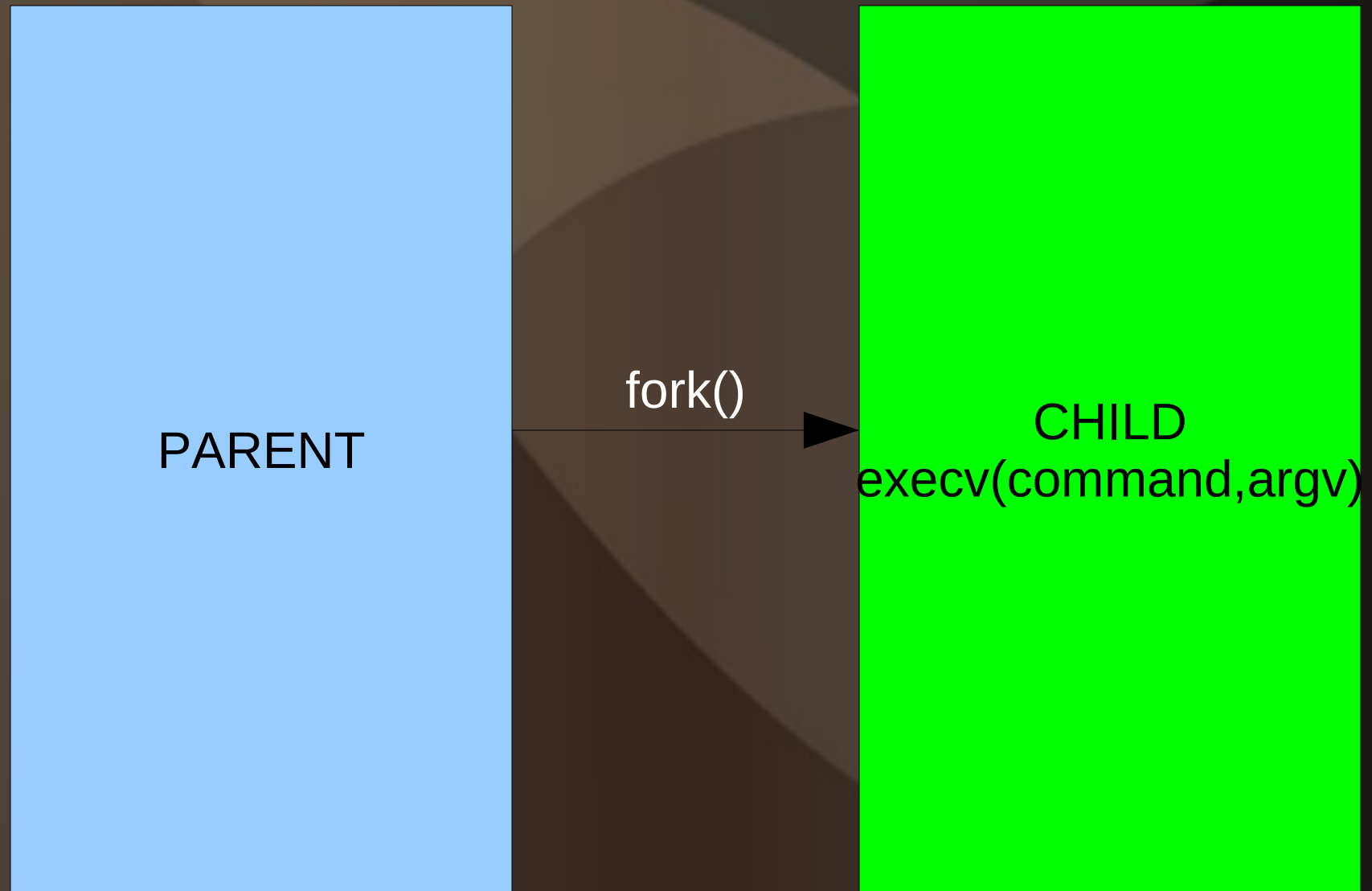
Visualizing Process Creation



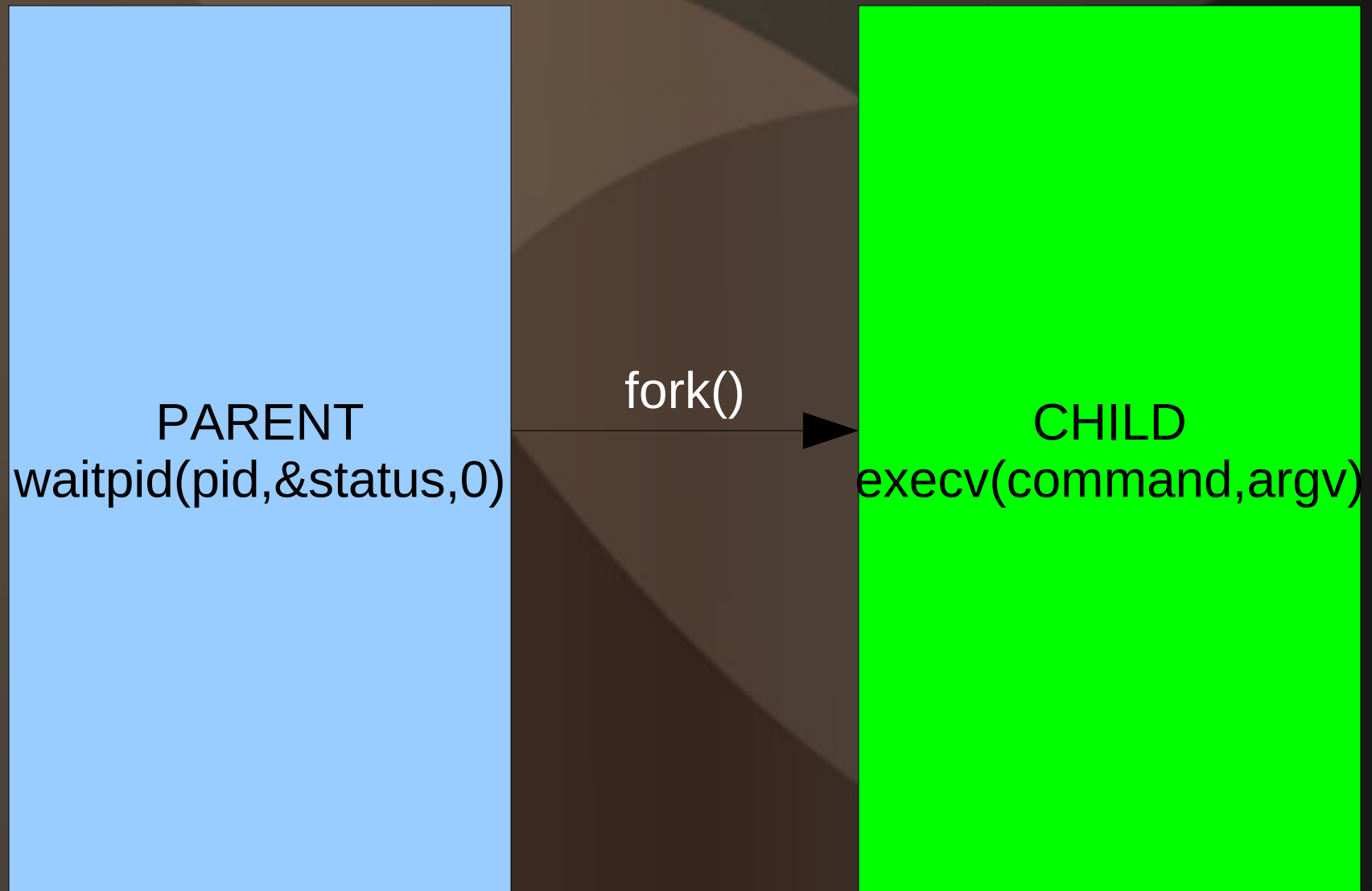
Visualizing Process Creation



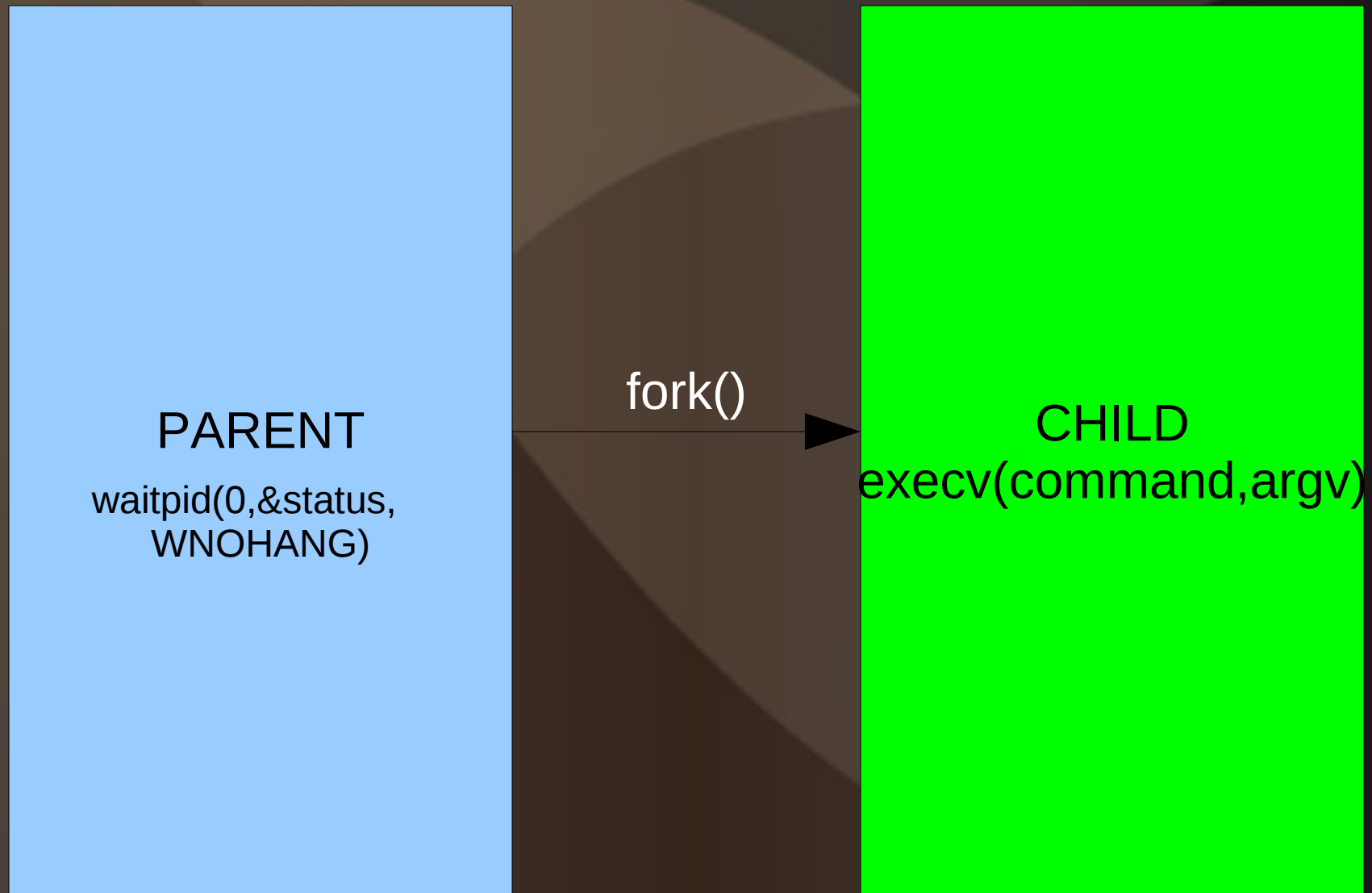
Visualizing Process Creation



Visualizing Process Creation



Visualizing Process Creation Background Execution



Environment Variables

- You'll need to be able to recognize environment variables when you see them.
- However, when you do recognize them, how do you obtain their value?

Environment Variables: Recognizing Them

```
cabrera:/home/cabrera# echo $USER
```

```
→ echo $USER
```

```
(after expanding $USER)
```

```
→ echo cabrera
```

- Environment variables always begin with '\$'.

Environment Variables: A Common List

- \$HOME
- \$PATH
- \$USER
- \$SHELL
- \$PWD
- ...

Obtaining the Value of an Environment Variable

```
char *getenv(const char *name);
```

- Returns the value of an environment variable, if found.
- Else, returns null.

Usng getenv()

```
char *var_value;  
  
var_value = getenv("PATH");  
if(var_value != NULL)  
    printf("PATH: %s\n", var_value);
```

Advice

- Start early (as usual)
 - This project is moderately intricate.
 - There are many corner cases to be aware of.
- Use `calloc()` instead of `malloc()` for allocation.
- Separate parsing into multiple phases.
- Write small programs to play around with `execv()`, `fork()`, and `waitpid()`.
- Play around with `tcsh` and come up with test cases.
- Think about what state the shell might need to keep.

Next Time:

- I/O Redirection
- Pipes
- More Details on Background Execution
- Environment Variables
- Zombies: How to Avoid them or How to Kill Them
- More hints

Any Questions?