Executing
External Process
Executing External Commands

```c
void my_execute(char **cmd) {
    execv(cmd[0], cmd);
}
```

• Problems???
execv

- `#include <unistd.h>`
- `int execv(char *absolute_pathname, char **arguments)`
  - `execv(cmd[0], cmd);`
- Returns **only if** execution fails
  - In which case it is -1
- Otherwise, the new process **overwrites** the images of the existing process
  - The shell if this case
- Note, you **have** to use execv
  - You can not use system or any of the other exec calls
Executing External Commands

void my_execute(char **cmd) {
  execv(cmd[0], cmd);
}

• Problems???
  – The command (ls in the example) will execute
  – The command's process will replace the shell's
  – When the command completes, it will appear as if the shell crashed/exited to the user

• Solutions?
  – Fork
fork

- `#include <sys/types.h>`
- `#include <unistd.h>`
- `pid_t fork()`
- Spawns a new process
  - Original is parent process
  - New is child process
- Returns three types of values
  - Failure
    - -1
  - In parent process
    - pid of child process
  - In child process
    - 0
pid

- Process identifier
  - Unique number representing the process
- Can get using getpid
  - `#include <sys/types.h>`
  - `#include <unistd.h>`
  - `pid_t getpid()`
- Can get parent's pid using getppid()
  - Same libraries
- How to get childrens' pid?
  - Have to save from fork calls
Executing External Commands

```c
void my_execute(char **cmd) {
    pid_t pid = fork();
    if (pid == -1) {
        //Error
        Exit(1);
    }
    else if (pid == 0) {
        //Child
        execv(cmd[0], cmd);
        //???
    }
    else {
        //Parent
        //???
    }
}
```

- Where to go now???
  - Child's command might not execute
    - Need to process error
      - Nothing to do in this case
    - Need to inform user
  - Parent will loop before child finishes
    - Messes up the prompt display
    - Need to wait for child to finish
void my_execute(char **cmd) {
    pid_t pid = fork();
    if (pid == -1) {
        //Error
        Exit(1);
    } else if (pid == 0) {
        //Child
        execv(cmd[0], cmd);
        fprintf("Problem executing %s\n", cmd[0]);
        exit(1);
    } else {
        //Parent
        //wait???
    }
}

• Why the exit(1)
  – Child process doesn't terminate at the end of the if statement
  – Will result in two shell processes executing in the same process group
• As for how to wait...
waitpid

- `#include <sys/types.h>`
- `#include <sys/wait.h>`
- `#include <unistd.h>`
- `pid_t waitpid(pid_t pid, int *status, int options)`
  - `pid` is the process to wait on
    - -1, 0 is for a single process
  - Wait differently depending on the option code
    - 0 to wait until the child terminates
  - Returns id of process who's state has changed
    - -1 on error
  - The status field is simply a second return value
    - Can pass in NULL to not receive this
    - Look in the man page for more information
void my_execute(char **cmd) {
    int status;
    pid_t pid = fork();
    if (pid == -1) {
        //Error
        Exit(1);
    } else if (pid == 0) {
        //Child
        execv(cmd[0], cmd);
        fprintf("Problem executing %s\n", cmd[0]);
        exit(1);
    } else {
        //Parent
        waitpid(pid, &status, 0);
    }
}
What if...

void my_execute(char **cmd) {
    pid_t pid = fork();
    if (pid == 0) {
        //Child
        //Do nothing
    }
    else {
        //Parent
        exit(0)
    }
}

• Parent exits while child is still running
  - Child becomes an orphan process
  - Child's parent becomes the init process (pid = 1)
void my_execute(char **cmd) {
    pid_t pid = fork();
    if (pid == 0) {
        //Child
        exit(0);
    }
    else {
        //Parent
        //Do nothing
    }
}

- Parent does not wait on child and child exits
  - Child becomes an zombie process
  - Child's resources are reclaimed but it still takes up a slot in the process table
  - Requires init to eventually remove it