

# CSc 352

## exec and fork

Benjamin Dicken

# Exec Functions

- Replaces the currently-running executable with a new one in the same process
- Several varieties

**\$ man exec**

```
#include <stdio.h>
#include <unistd.h>
int main() {
    printf("hi!\n");
    char* arguments[] = {NULL};
    char* environment[] = {NULL};
    int value = execve("./a.out", arguments, environment);
    if (value == -1) {
        printf("Failed to execute\n");
    }
    return 0;
}
```

# Fork function

- Create a new process as a duplicate of the current one
- Parent / Child relationship
- Returns PID of child to the parent, 0 to the child

**\$ man fork**

```
#include <stdio.h>
#include <unistd.h>
int main() {
    printf("hi!\n");
    int pid = fork();
    if (pid != 0) {
        printf("Parent: hello\n");
        sleep(10);
        printf("Parent: goodbye\n");
    } else {
        printf("Child: hello\n");
        sleep(10);
        printf("Child: goodbye\n");
    }
    return 0;
}
```

# Fork and Exec

Why are these system calls useful?

How could these be used *\*together\** to do interesting things?

```
#include <stdio.h>
#include <unistd.h>
int main() {
    printf("hi!\n");
    int pid = fork();
    if (pid != 0) {
        printf("I'm the parent\n");
        sleep(10);
    } else {
        printf("I'm the child\n");
        char* arguments[] = {NULL};
        char* environment[] = {NULL};
        int value = execve("./some_other_executable", arguments, environment);
        if (value == -1) {
            printf("Failed to execute\n");
        }
    }
    return 0;
}
```

```
#include <stdio.h>
#include <unistd.h>
int main() {
    printf("hi!\n");
    char* arguments[] = {NULL};
    char* environment[] = {NULL};
    sleep(2);
    fork();
    int value = execve("./proc", arguments, environment);
    if (value == -1) {
        printf("Failed to execute\n");
    }
    sleep(100);
    return 0;
}
```



# Other functions

kill: send signals to processes based on pid

```
$ man kill
```

getpid: get pid of current process

```
$ man getpid
```

waitpid: wait for process with pid to complete

```
$ man waitpid
```

```
int main(int argc, char* argv[]) {

    char* file_name = argv[1];
    char* search_term = argv[2];

    // Figure out file size
    FILE* to_search = fopen(file_name, "r");
    fseek(to_search, 0, SEEK_END);
    uint64_t size = ftell(to_search);
    rewind(to_search);

    uint64_t read = 0;
    uint64_t count = 0;
    uint64_t lines = 0;
    char line[256];
    while( fgets(line, 256, to_search) ) {
        lines += 1;
        if ( strstr(line, search_term) != NULL ) {
            count += 1;
        }
        read += strlen(line);
        if (read > size) { break; }
    }

    printf("Matching line count: %lu\n", count);
    printf("Total lines processed: %lu\n", lines);

    return 0;
}
```

# What does this do?

```
int main(int argc, char* argv[]) {

    char* file_name = argv[1];
    char* search_term = argv[2];

    // Figure out file size
    FILE* to_search = fopen(file_name, "r");
    fseek(to_search, 0, SEEK_END);
    uint64_t size = ftell(to_search);
    rewind(to_search);

    uint64_t read = 0;
    uint64_t count = 0;
    uint64_t lines = 0;
    char line[256];
    while( fgets(line, 256, to_search) ) {
        lines += 1;
        if ( strstr(line, search_term) != NULL ) {
            count += 1;
        }
        read += strlen(line);
        if (read > size) { break; }
    }

    printf("Matching line count: %lu\n", count);
    printf("Total lines processed: %lu\n", lines);

    return 0;
}
```

How can we make  
it perform better  
with fork and exec?

# Communicating Between Processes

- Use the **pipe** function.
- **pipe** accepts an array of two file descriptors
  - one is a read-end, one is a write end
- One process can write to the pipe, another can read.

```
$ man pipe
```

Process A

`int fd[2];`  
`pipe(fd)`

`fork()`

Process B

`close(fd[1])`

`read(fd[0],...)`

`read(fd[0],...)`

`close(fd[0])`

`close(fd[0])`

`write(fd[1],...)`

`write(fd[1],...)`

`close(fd[1])`

Process A

```
int fd[2];  
pipe(fd)
```

fork() (x2)

Process B

Process C

close(fd[1])

close(fd[0])

close(fd[0])

read(fd[0],..)

write(fd[1],..)

read(fd[0],..)

write(fd[1],..)

read(fd[0],..)

write(fd[1],..)

read(fd[0],..)

write(fd[1],..)

close(fd[0])

close(fd[1])

close(fd[1])

# Communicate between parent and child processes

```
int pipe_descriptors[2];
int status = pipe(pipe_descriptors);
if (status < 0) {
    fprintf(stderr, "BAD.\n");
    return 1;
}

int process_id = fork();

if (process_id != 0) { // Parent
    close(pipe_descriptors[1]); // Close write
    char message[10];
    read(pipe_descriptors[0], message, sizeof(char)*10); // Receive message
    printf("Message from child: %s\n", message);
    close(pipe_descriptors[0]); // Close read
} else { // Child
    close(pipe_descriptors[0]); // Close read
    char send[10] = "LOLZ\0";
    write(pipe_descriptors[1], send, sizeof(char)*10); // Send message
    close(pipe_descriptors[1]); // Close write
}
```



Matching line count: 7129  
Matching line count: 7129  
Matching line count: 7121  
Matching line count: 7121



Matching line count: 28500

How can we go  
from this to this  
with multiple  
processes?



# Timing

How does C program performance compare to other languages?

Python? Java?