15-213 Recitation 8

Processes, Signals, Tshlab

22 October 2018

Outline

- Cachelab Style
- Process Lifecycle
- Signal Handling

Cachelab Style Grading

- **Style grades will be available "soon"**
  - Click on your score to view feedback for each rubric item
  - Make sure points are added correctly!
  - File regrade requests on Piazza if we made a mistake.
- **Common mistakes**
  - Missing descriptions at the top of your file and functions
  - Error-checking for `malloc` and `fopen`
  - Writing everything in main function without helpers.
  - Lack of comments in general.
- **Keep style in mind as you work on tshlab!**
  - Error-checking is particularly important to consider

Shell Lab

- **Due date:** next Tuesday (October 30th)
- Simulate a Linux-like shell with I/O redirection

- **Review the writeup carefully.**
  - Review once before starting, and again when halfway through
  - This will save you a lot of style points and a lot of grief!

- **Read Chapter 8 in the textbook:**
  - Process lifecycle and signal handling
  - How race conditions occur, and how to avoid them
  - **Be careful not to use code from the textbook without understanding it first.**

Process “Lifecycle”

- `fork()`

  Create a duplicate, a “child”, of the process

- `execve()`

  Replace the running program

- `exit()`

  End the running program

- `waitpid()`

  Wait for a child process to terminate

Notes on Examples

- **Full source code of all programs is available**
  - TAs may demo specific programs
- **In the following examples, `exit()` is called**
  - We do this to be explicit about the program's behavior
  - Exit should generally be reserved for terminating on error
- **Unless otherwise noted, assume all syscalls succeed**
  - Error checking code is omitted.
  - Be careful to check errors when writing your own shell!

Processes are separate

- How many lines are printed?
- If pid is at address `0x7fff2bcc264c`, what is printed?

```
int main(void) {
    pid_t pid;
    pid = fork();
    printf("%p - %d\n", &pid, pid);
    exit(0);
}
```

Processes are separate

- How many lines are printed?
- If pid is at address `0x7fff2bcc264c`, what is printed?

```
int main(void) {
    pid_t pid;
    pid = fork();
    printf("%p - %d\n", &pid, pid);
    exit(0);
}
```

```
0x7fff2bcc264c - 24750
0x7fff2bcc264c - 0
```

The order and the child's PID (printed by the parent) may vary, but the address will be the same in the parent and child.

Processes Change

- What does this program print?

```
int main(void) {
    char *args[3] = {
        "/bin/echo", "Hi 18213!", NULL
    };
    execv(args[0], args);
    printf("Hi 15213!\n");
    exit(0);
}
```

Processes Change

- What does this program print?

```
int main(void) {
    char *args[3] = {
        "/bin/echo", "Hi 18213!", NULL
    };
    execv(args[0], args);
    printf("Hi 15213!\n");
    exit(0);
}
```

```
Hi 18213!
```

Processes Change

- What about this program? What does it print?

```
int main(void) {
    char *args[3] = {
        "/bin/blahblah", "Hi 15513!", NULL
    };
    execv(args[0], args);
    printf("Hi 14513!\n");
    exit(0);
}
```

Processes Change

- What about this program? What does it print?

```
int main(void) {
    char *args[3] = {
        "/bin/blahblah", "Hi 15513!", NULL
    };
    execv(args[0], args);
    printf("Hi 14513!\n");
    exit(0);
}
```

```
Hi 14513!
```

On Error

- What should we do if `malloc` fails?

```c
const size_t HUGE = 1 * 1024 * 1024 * 1024;
int main(void) {
    char *buf = malloc(HUGE * HUGE);




    printf("Buf at %p\n", buf);
    free(buf);
    exit(0);
}
```

On Error

- What should we do if `malloc` fails?

```
const size_t HUGE = 1 * 1024 * 1024 * 1024;
int main(void) {
    char *buf = malloc(HUGE * HUGE);

    if (buf == NULL) {
        fprintf(stderr, "Failure at %u\n", __LINE__);
        exit(1);
    }

    printf("Buf at %p\n", buf);
    free(buf);
    exit(0);
}
```

Exit values can convey information

- Two values are printed. Are they related?

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

Exit values can convey information

- Two values are printed. Are they related?

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

```
0x7b54 exited with 0x54
```

They're the same!... almost.
Exit codes are only one byte in size.

Processes have ancestry

- What's wrong with this code? (assume that fork succeeds)

```
int main(void) {
    int status = 0, ret = 0;
    pid_t pid = fork();
    if (pid == 0) {
        pid = fork();
        exit(getpid());
    }

    ret = waitpid(-1, &status, 0);
    printf("Process %d exited with %d\n", ret, status);

    ret = waitpid(-1, &status, 0);
    printf("Process %d exited with %d\n", ret, status);
    exit(0);
}
```

Processes have ancestry

- What's wrong with this code? (assume that fork succeeds)

```
int main(void) {
    int status = 0, ret = 0;
    pid_t pid = fork();
    if (pid == 0) {
        pid = fork();
        exit(getpid());
    }

    ret = waitpid(-1, &status, 0);
    printf("Process %d exited with %d\n", ret, status);

    ret = waitpid(-1, &status, 0);
    printf("Process %d exited with %d\n", ret, status);
    exit(0);
}
```

`waitpid` will reap only children, not grandchildren, so the second `waitpid` call will return an error.

Process Graphs

- How many different sequences can be printed?

```
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);
}
```

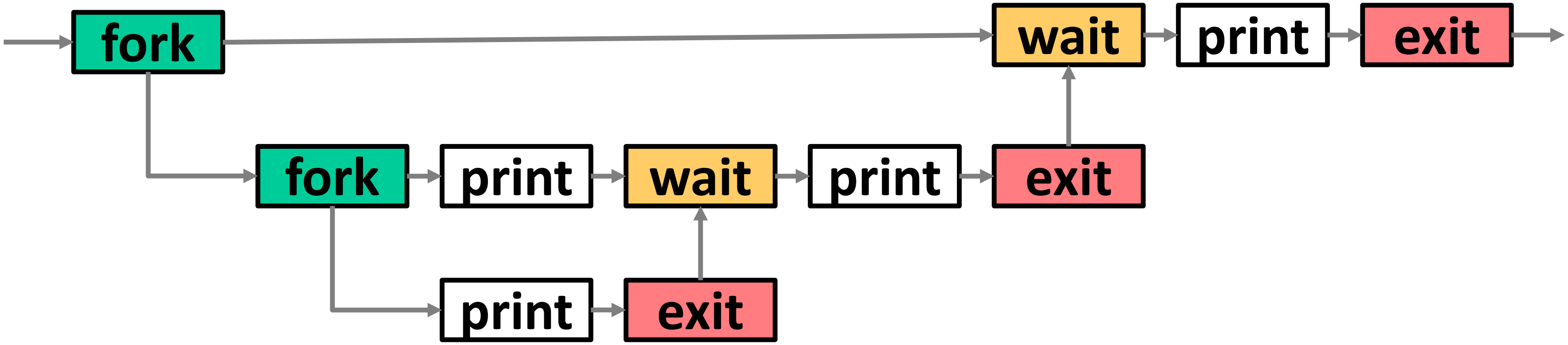
Process Graphs

- How many different sequences can be printed?

```
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);
}
```

Two different sequences. See the process graph on the next slide.

Process Diagram



Process Graphs

- **How many different lines are printed?**

```
int main(void) {
    char *tgt = "child";
    pid_t pid = fork();
    if (pid == 0) {
        pid = getppid(); // Get parent pid
        tgt = "parent";
    }
    kill(pid, SIGKILL);
    printf("Sent SIGKILL to %s:%d\n", tgt, pid);
    exit(0);
}
```

Process Graphs

- **How many different lines are printed?**

```c
int main(void) {
    char *tgt = "child";
    pid_t pid = fork();
    if (pid == 0) {
        pid = getppid(); // Get parent pid
        tgt = "parent";
    }
    kill(pid, SIGKILL);
    printf("Sent SIGKILL to %s:%d\n", tgt, pid);
    exit(0);
}
```

Anywhere from 0-2 lines. The parent and child try to terminate each other.

Signals and Handling

- **Signals can happen at any time**
  - Control when through blocking signals

- **Signals also communicate that events have occurred**
  - What event(s) correspond to each signal?

- **Write separate routines for receiving (i.e., signals)**

Counting with signals

- **Will this code terminate?**

```
volatile int counter = 0;
void handler(int sig) { counter++; }

int main(void) {
    signal(SIGCHLD, handler);
    for (int i = 0; i < 10; i++) {
        if (fork() == 0) { exit(0); }
    }
    while (counter < 10) {
        mine_bitcoin();
    }
    return 0;
}
```

Counting with signals

- **Will this code terminate?**

```
volatile int counter = 0;
void handler(int sig) { counter++; }

int main(void) {
    signal(SIGCHLD, handler);
    for (int i = 0; i < 10; i++) {
        if (fork() == 0) { exit(0); }
    }
    while (counter < 10) {
        mine_bitcoin();
    }
    return 0;
}
```

(Don't use `signal`, use `Signal` or `sigaction` instead!)

(Don't busy-wait, use `sigsuspend` instead!)

It might not, since signals can coalesce.

Proper signal handling

- **How can we fix the previous code?**
  - Remember that signals will be coalesced, so the number of times a signal handler has executed is **not** necessarily the same as number of times a signal was sent.
  - We need some other way to count the number of children.

Proper signal handling

- **How can we fix the previous code?**
  - Remember that signals will be coalesced, so the number of times a signal handler has executed is **not** necessarily the same as number of times a signal was sent.
  - We need some other way to count the number of children.

```
void handler(int sig) {
    pid_t pid;
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
        counter++;
    }
}
```

(This instruction isn't atomic. Why won't there be a race condition?)

If you get stuck

- **Read the writeup!**
- **Do manual unit testing before `runtrace` and `sdriver`!**

- **Read the writeup!**
- **Post private questions on Piazza!**

- **Think carefully about error conditions.**
  - Read the man pages for each syscall when in doubt.
  - What errors can each syscall return?
  - How should the errors be handled?

Appendix: Blocking signals

- **Surround blocks of code with calls to `sigprocmask`.**
  - Use SIG_BLOCK to block signals at the start.
  - Use SIG_SETMASK to restore the previous signal mask at the end.
- **Don't use SIG_UNBLOCK.**
  - We don't want to unblock a signal if it was already blocked.
  - This allows us to nest this procedure multiple times.

```
sigset_t mask, prev;
sigemptyset(&mask, SIGINT);
sigaddset(&mask, SIGINT);
sigprocmask(SIG_BLOCK, &mask, &prev);
// ...
sigprocmask(SIG_SETMASK, &prev, NULL);
```

Appendix: Errno

`#include <errno.h>`

- **Global integer variable used to store an error code.**
  - Its value is set when a system call fails.
  - Only examine its value when the system call's return code indicates that an error has occurred!
  - Be careful not to call make other system calls before checking the value of `errno`!
- **Lets you know why a system call failed.**
  - Use functions like `strerror`, `perror` to get error messages.
- **Example: assume there is no "foo.txt" in our path**

```
int fd = open("foo.txt", O_RDONLY);
if (fd < 0) perror("open");
// open: No such file or directory
```

Appendix: Writing signal handlers

- **G1. Call only async-signal-safe functions in your handlers.**
  - Do not call `printf`, `sprintf`, `malloc`, `exit`! Doing so can cause deadlocks, since these functions may require global locks.
  - We've provided you with `sio_printf` which you can use instead.
- **G2. Save and restore `errno` on entry and exit.**
  - If not, the signal handler can corrupt code that tries to read `errno`.
  - The driver will print a warning if `errno` is corrupted.
- **G3. Temporarily block signals to protect shared data.**
  - This will prevent race conditions when writing to shared data.
- **Avoid the use of global variables in tshlab.**
  - They are a source of pernicious race conditions!
  - You do not need to declare any global variables to complete tshlab.
  - Use the functions provided by `tsh_helper`.