

15-213 Recitation

Processes, Signals, Tshlab

November 7th, 2022

Your TAs

Outline

- Logistics
- Process Lifecycle
- Error Handling

Learning Objectives

■ Expectations:

- Basic understanding of signals & processes

■ Goals:

- Better understanding of signals & processes
- Understand what a shell does and how to interact with it
- Understand how to properly handle errors

Logistics

- Malloc Final due Nov 8th
 - **TOMORROW (TUESDAY)**
 - Can use up to 2 late days!
 - Style grading mm.c (not checkheap)
 - Sign up for malloc final code reviews

Post Mid-Semester Feedback Form

- **Please Take 5 minutes to Fill this out:**
 - <https://forms.gle/z2CrzaSXWynB2gLLA>

- **TA Hiring For the Next Semester hasn't been started by the Department yet, we shall be announcing so when it does.**
 - All hiring will be done through the CSD portal, not via email.

Shell Lab

- **Due date:** November 22nd
- Simulate a Linux-like shell

- **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.**



Shell demo

■ Process Lifecycle

- `$ ps -a`
 - This reports a snapshot of all the current processes. You can identify them by PID

```

PID TTY          TIME CMD
3435 pts/18      00:00:01 vim
4856 pts/22      00:00:00 vim
4894 pts/19      00:00:00 vim
6260 pts/17      00:00:00 vim
6737 pts/23      00:00:00 rlwrap
7075 pts/25      00:00:00 dbus-launch

```

- `$ ctrl+z` sends SIGTSTP and stops the current foreground process
 - `fg/bg` to run the most recently stopped process in the foreground/background
- `$./long_binary_with_lots_of_io &`
 - Appending **&** to the end of a command runs it in the background

■ I/O redirection

- `$./hex2raw < exploit.txt > exploit-raw.txt`
 - **<** to redirect input and **>** to redirect output to the specified file

Shell Demo

- Login to shark machine
- `wget http://www.cs.cmu.edu/~213/activities/rec10.tar`
- `tar -xvf rec10.tar`
- `cd rec10`

Process “Lifecycle”

- `fork()`
Create a duplicate, a “child”, of the process
- `execve()`
Replace the running program
- ... [Complete Work]
- `exit()`
End the running program
- `waitpid()`
Wait for a child process to terminate

Processes are separate

- How many lines are printed?
- Will the pid address be different?
- Will the pid be different?

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

Processes are separate

- How many lines are printed?
- Will the pid address be different?
- Will the pid be different?

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

```
pid addr: 0x7fff2bcc264c - pid: 24750  
pid addr: 0x7fff2bcc264c - pid: 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  
    };  
    execve(args[0], args, environ);  
    printf("Hi 15213!\n");  
    exit(0);  
}
```

Processes Change

- What does this program print?

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

Hi 18213!

Processes Change

- What about this program? What does it print?
- Assume that `/bin/blahblah` does **not** exist.

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

Processes Change

- What about this program? What does it print?
- Assume that `/bin/blahblah` does **not** exist.

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

Hi 14513!

Exit values can convey information

- Two values are printed. What are they?

```
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);
}
```


Exit values can convey information

- Two values are printed. What are they?

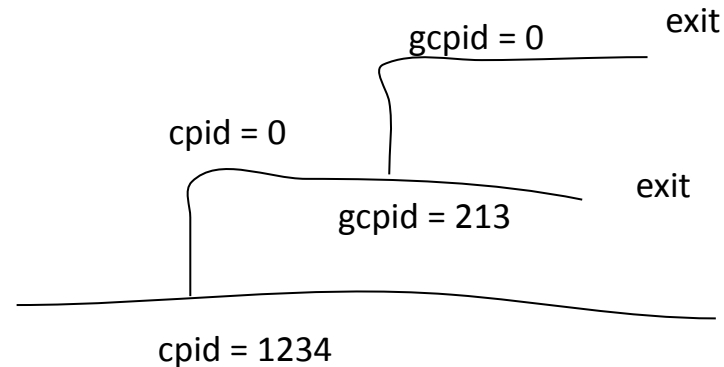
```
int main(void) {
    pid_t pid = fork();
    if (pid == 0) { exit(0x213); }
    else {
        int status = 0;
        waitpid(pid, &status, 0);
        printf("0x%x exited with 0x%x\n", pid,
              WEXITSTATUS(status));
    }
    exit(0);
}
```

0x7b54 exited with 0x13
WEXITSTATUS(status) will only return 1
byte of information

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());  
    }
```

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

```
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);
```

```
}
```

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);
}
```

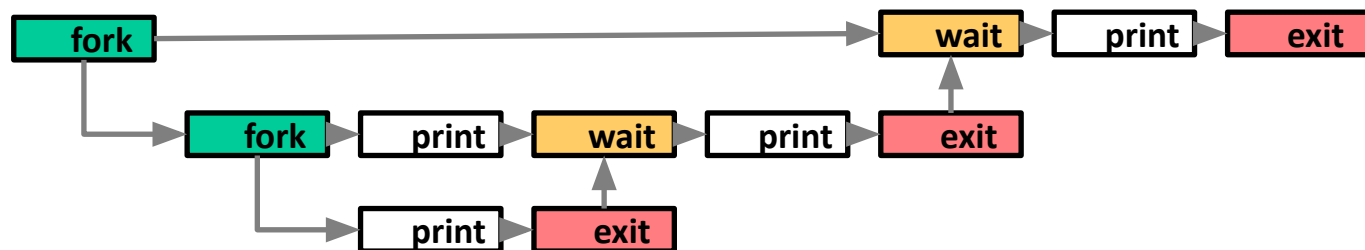
Process Graphs

- 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);
}

```



Error in UNIX - return value

- Can syscalls fail?
- How to tell the difference?

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

Error in UNIX - What error?

- Can syscalls fail?
- How to tell the difference?
 - Returned -1
- So, my fantastic syscalls failed.
- How can I tell what went wrong?

```
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  
}
```

Error in UNIX - What error?

- Can syscalls fail?
- How to tell the difference?
 - Returned -1
- So, my fantastic syscalls failed.
- How can I tell what went wrong?
 - **errno** is a global variable that syscalls store information in when they fail
 - **strerror** turns errno codes into printable messages
 - **perror (print error)** is a handy shorthand

```

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
}

```

Always print `strerror(errno)` **and** the names of filenames involved in failing system calls

Process Graphs

■ How many different lines are printed?

```
int main(void) {
    char *tgt = "child";
    sigset_t mask, old_mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigprocmask(SIG_SETMASK, &mask, &old_mask); // Block
    pid_t pid = fork();
    if (pid == 0) {
        pid = getppid(); // Get parent pid
        tgt = "parent";
    }
    kill(pid, SIGINT);
    sigprocmask(SIG_SETMASK, &old_mask, NULL); // Unblock
    printf("Sent SIGINT to %s:%d\n", tgt, pid);
    exit(0);
}
```

Process Graphs

■ How many different lines are printed?

```
int main(void) {
    char *tgt = "child";
    sigset_t mask, old_mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigprocmask(SIG_SETMASK, &mask, &old_mask); // Block
    pid_t pid = fork();
    if (pid == 0) {
        pid = getppid(); // Get parent pid
        tgt = "parent";
    }
    kill(pid, SIGINT);
    sigprocmask(SIG_SETMASK, &old_mask, NULL); // Unblock
    printf("Sent SIGINT to %s:%d\n", tgt, pid);
    exit(0);
}
```

0 or 1 line. 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.

sigsuspend

```
int sigsuspend(const sigset_t *mask);
```

- Suspend current process until a signal is received, you can specify which one using a mask

This is an atomic version of:

```
sigprocmask(SIG_SETMASK, &mask, &prev)
```

```
pause();
```

```
sigprocmask(SIG_SETMASK, &prev, NULL);
```

- This still doesn't fix the issue of signals coalescing!

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?)

Error and signals : Recap

- **You can't expect people to block signals around all error handling logic**
- **Hence, your signal handler shouldn't interfere with them**
- **Solution:**
 - Do not make any system call that could set errno
 - Save and restore errno (store at beginning of handler and restore after)
 - Think about what would work for the case you are using, not one rule

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: Notes on Examples

- **Full source code of all programs is available**
 - TAs may demo specific programs

- **In the 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!

Appendix: Example Question: Possible outputs?

```
1  int main( ) {
2      int val = 2;
3      printf("%d", 0);
4      fflush(stdout);
5
6      if (fork() == 0) {
7          val++;
8          printf("%d", val);
9          fflush(stdout);
10     }
11     else {
12         val--;
13         printf("%d", val);
14         fflush(stdout);
15         wait(NULL);
16     }
17
18     val++;
19     printf("%d", val);
20     fflush(stdout);
21     exit(0);
22 }
```

■ **There is no deterministic interleaving of the parent and child after the call to fork()**

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("foo.txt");  
// foo.txt: 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`.