Lecture 5: Signals

Note: Reading these lecture notes is not a substitute for watching the lecture. I frequently go off script, and you are responsible for understanding everything I talk about in lecture unless I specify otherwise.

Pipes

Recall from last lecture that a pipe is one of the main forms of interprocess communication, allowing for freeform exchange of data between two processes. Before forking, we create two "virtual files" that are linked to each other via the `pipe` syscall. On fork, both processes inherit the same files, and can communicate with each other if one writes to the file and the other reads from it.

Implementing `subprocess`

Similar to `system`, subprocess launches an executable in a child process. However, instead of waiting for the child process to exit, it returns *immediately*, so that the parent can communicate with the child while it runs. It returns the PID of the child, as well as a file descriptor; if the parent writes to this file descriptor, the child will be able to read that data by reading from stdin.

```
typedef struct {
    pid_t pid;
    int infd;
} subprocess_t;

subprocess_t subprocess(const char *command);

// Example demonstrating how to use subprocess. We start "sort," feed it 4
// words (by writing to the input file descriptor, which is wired to stdin of
// "sort"), then close the input file descriptor (equivalent to pressing ctrl+D
// on the keyboard -- tells "sort" we're done feeding it words).
int main(int argc, char *argv[]) {
    subprocess_t sp = subprocess("/usr/bin/sort");
```

```c
    const char *words[] = {"pen", "pineapple", "apple", "pen"};
    for (size_t i = 0; i < sizeof(words) / sizeof(words[0]); i++) {
        dprintf(sp.infd, "%s\n", words[i]);
    }
    close(sp.infd);

    int status;
    pid_t pid = waitpid(sp.pid, &status, 0);
    return pid == sp.pid && WIFEXITED(status) ? WEXITSTATUS(status) : -1;
}

subprocess_t subprocess(const char *command) {
    int fds[2];
    pipe(fds);
    subprocess_t process = { fork(), fds[1] };
    if (process.pid == 0) {
        close(fds[1]);  // The child isn't writing to the pipe, so we can close
                        // the write end
        dup2(fds[0], STDIN_FILENO); // Rewires fd 0 to point to the read end of
                        // the pipe. The read end of the pipe now has 2 file
                        // descriptors pointing to it.
        close(fds[0]);  // Now that STDIN_FILENO points to the read end of the
                        // pipe, we can close this extra file descriptor.

        // Start the target exectuable:
        char *argv[] = {"/bin/sh", "-c", (char *) command, NULL};
        execvp(argv[0], argv);
    }

    // The parent isn't reading from the pipe, so we can close the read end:
    close(fds[0]);

    return process;
}
```

Two notes about this program:

- Closing pipes is extremely important! Generally, try to close pipes as early as you can, so that you don't forget to close them. If you forgot the `close(sp.infd)` line in `main` *or* the `close(fds[1])` line in `subprocess`, this program would hang, because the pipe would never reach EOF, so `sort` would never exit. Likewise, in `main`, if we moved the `close(sp.infd)` to be after the `waitpid`, the program would also hang.

- The output of `sort` is currently going to the terminal, because it has inherited the STDOUT file descriptor from the parent process. There is no way for the parent process to intercept the output of `sort` programmatically. To do so, we'd need a second pipe, where the child writes and the parent reads. You'll implement this in Assignment 2.

Tip: In assignments and exams, it's always a good idea to draw out all the file descriptor and file entry tables. As an example debugging strategy, if your program hangs, trace through line by line, updating your file descriptor/file entry tables as you go. Did you forget to close something somewhere? Is the refcount of the write end of a pipe more than zero, when you would have expected the pipe to have already reached EOF?

Signals

Signals are like fruits that processes throw at each other, where each fruit has a particular meaning. The operating system might throw a strawberry at me to tell me I divided by zero. Someone might throw a watermelon at me to tell me to quit. You can't tell who threw the fruit at you, and there is no attached information, but it communicates some very simple, predefined message.

There are many signals defined in Unix. Here are a few:

Signal	Number	Description
SIGHUP	1	Hangup (e.g. SSH session closed while program still running)
SIGINT	2	Interrupt (e.g. ctrl+c pressed on keyboard)
SIGILL	4	Illegal assembly instruction executed
SIGABRT	6	Abort (abort() called)
SIGFPE	8	Arithmetic Exception (e.g. divide by zero)
SIGKILL	9	Program forcefully killed
SIGSEGV	11	Segmentation Fault
SIGSYS	12	Bad System Call
SIGPIPE	13	Broken Pipe
SIGTERM	15	Terminated

SIGUSR1	16	User Signal 1 (you can assign this a custom meaning in your program)
SIGUSR2	17	User Signal 2 (you can assign this a custom meaning in your program)
SIGCHLD	18	A child process changed status
SIGTSTP	24	Stopped (e.g. ctrl+z pressed on keyboard)
SIGCONT	25	Continue after stopping

You absolutely should not memorize the signal numbers (there are #defines for that). However, by the time the midterm comes, you should be familiar with what SIGINT, SIGKILL, SIGTERM, SIGUSR1, SIGUSR2, SIGCHLD, SIGTSTP, and SIGCONT do and how you might use them.

Handling signals

You can install *signal handlers,* which are functions that get called when a program receives a signal. This code overrides the default SIGSEGV signal handler, so that we print “RIP segfault” instead of “Segmentation fault: core dumped.”

```
void handleSegfault(int sig) {
    printf("RIP segfault!\n");
    exit(11);
}

int main(int argc, char *argv[]) {
    signal(SIGSEGV, handleSegfault);
    *(int *)NULL = 0;
    printf("Goodbye\n");
}
```

SIGCHLD

The operating system sends SIGCHLD to a parent process when one of its child processes changes state (stops, continues, or exits). This is useful for asynchronously keeping track of child process: what if the parent wants to spawn some child processes, but it wants to do some useful work while keeping track of the child processes’ states? For example, a shell supporting background processes needs to spawn a process and keep track of that process so it can update its `jobs` list, but it needs to immediately print a shell prompt and allow you to type another command to run. We can install a SIGCHLD handler to be notified about when a child changes state, and to act appropriately when that happens.

Let's demonstrate this idea by writing a toy simulation. A father takes his 5 kids out to Disneyland, but he's tired, so he lets them go out to play while he takes a nap. Periodically, he wakes up to check how many kids have come back, and when they all come back, he goes home.

```c
static const size_t kNumChildren = 5;
static size_t numDone = 0;

static void reapChild(int sig) {
    waitpid(-1, NULL, 0);
    numDone++;
}

int main(int argc, char *argv[]) {
    printf("Let my five kids play while I take a nap.\n");
    signal(SIGCHLD, reapChild);
    for (size_t kid = 1; kid <= 5; kid++) {
        pid_t pid = fork();
        if (pid == 0) {
            sleep(3 * kid);
            printf("Kid #%zu tired... returns to dad.\n", kid);
            return 0;
        }
    }
    while (numDone < kNumChildren) {
        printf("Kids still playing, back to sleep.\n");
        sleep(5);
        printf("Dad wakes up!\n");
    }

    printf("All children accounted for. Let's go home!\n");
    return 0;
}
```

This code works if you run it, but if you change the `sleep` call to sleep for a constant amount of time (e.g. change it to `sleep(3)`), the program breaks. A bunch of kids come back, all at the same time, but dad only registers one kid as having come back. This is because if multiple signals come in at the same time, **the signal handler is only run once.** (If three SIGCHLD signals come in while a process is off the processor, the operating system only records the fact that at least one SIGCHLD came in; it doesn't record how many signals came in while the process was in the ready queue or blocked set.) To fix this, we need to call waitpid in a while loop:

```
static void reapChild(int sig) {
    while (true) {
        pid_t pid = waitpid(-1, NULL, 0);
        if (pid == -1) break;
        numDone++;
    }
}
```

But this breaks the original program. Let's say only one child has exited, and the other children are still out playing. The parent will call waitpid once – it will return fine – but then it will call waitpid a second time, which will wait for another child to exit instead of letting dad get back to his nap.

We need to tell waitpid to reap children that have already exited, but to not block if there are more children that *haven't* exited. We can use the WNOHANG flag for this:

```
static void reapChild(int sig) {
    while (true) {
        pid_t pid = waitpid(-1, NULL, WNOHANG);
        if (pid <= 0) break;
        numDone++;
    }
}
```

Note that we also changed the `if (pid <= 0)` condition. A return value of -1 (probably) means that there are no child processes left; now, additionally, when WNOHANG is specified, a return value of 0 means there *are* still child processes, and we would have waited for them normally, but we're returning immediately because of WNOHANG.

***Every* SIGCHLD handler has the structure of `reapChild` above.**

Synchronization problems

Signal handling introduces new problems for synchronization, because the code in a signal handler might be executed at any time, even if we're in the middle of doing something important that shouldn't be interrupted. To illustrate this problem, we'll write a super simple "shell." It's not even a shell – it doesn't ask the user for any input – but it runs `date` three times, telling the user that `date` has been added to the "shell's" job list at the beginning and then printing that `date` completed once it exits.

```
static void reapChild(int sig) {
    while (true) {
        pid_t pid = waitpid(-1, NULL, WNOHANG);
        if (pid <= 0) break;
        printf("Job %d removed from job list.\n", pid);
    }
}

int main(int argc, char *argv[]) {
    signal(SIGCHLD, reapChild);
    for (size_t i = 0; i < 3; i++) {
        pid_t pid = fork();
        if (pid == 0) {
            char *args[] = {"date", NULL};
            execvp(args[0], args);
            printf("execvp failed\n");
            return -1;
        }

        // Let's say this shell uses some ridiculously inefficient data structure
        // to implement the job list, and adding processes to this job list takes
        // whole second. (Yikes.)
        sleep(1);

        printf("Job %d added to job list.\n", pid);
    }

    return 0;
}
```

This code prints "Job removed from job list" *before* "Job added to job list." How can we fix this?

- We can remove the `sleep` call, akin to improving the efficiency of the job list data structure. But then we're at the mercy of the OS scheduler, praying that it schedules the parent process before the child finishes.
- We could call `waitpid` in `main` instead of using a signal handler… But then how would we know *when* to call `waitpid`? We'd essentially be busy waiting, calling `waitpid` every once in a while until a child process exits.
- We could override the signal handler to be an empty signal handler while we're creating the process and updating the job list, then restore it once we're finished. But what if a signal comes in during that time? We'd lose track of that signal.

What we *really* want to do is to *defer handling* of SIGCHLD signals that come in while we're updating the job list. *Signal masks* let us tell the operating system, "hey, I don't want to be bothered with SIGCHLD at the moment – can you hang on to that and call my signal handler later?"

```
int main(int argc, char *argv[]) {
    signal(SIGCHLD, reapChild);

    // Create an empty set of signals:
    sigset_t mask;
    sigemptyset(&mask);
    // Add SIGCHLD to the set:
    sigaddset(&mask, SIGCHLD);

    for (size_t i = 0; i < 3; i++) {
        // Tell the OS to defer handling of signals in the set (remember, the
        // set currently contains only SIGCHLD):
        sigprocmask(SIG_BLOCK, &mask, NULL);

        pid_t pid = fork();
        if (pid == 0) {
            // Signal masks are preserved across fork and execvp boundaries! We
            // need to unblock SIGCHLD so we don't interfere with date
            sigprocmask(SIG_UNBLOCK, &mask, NULL);

            char *args[] = {"date", NULL};
            execvp(args[0], args);
            printf("execvp failed\n");
            return -1;
        }
        sleep(1);
        printf("Job %d added to job list.\n", pid);

        // Tell the OS it's now okay to call my signal handler
        sigprocmask(SIG_UNBLOCK, &mask, NULL);
    }

    return 0;
}
```

Sending signals

You can send a signal to another process using the `kill` syscall:

```
int kill(int pid, int sig);
```

If you want to send *yourself* a signal, you can use `raise`:

```
int raise(int sig);
```

This is just shorthand for `kill(getpid(), sig)`.