Lecture 6: Multiprocessing Recap

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.

I needed to rush through the past two weeks in order to cover the material necessary for assignments 1 and 2, but now we can take a short breather and digest some of the multiprocessing material before moving onto multithreading!

You should feel very proud of yourself for coming this far. Multithreading usually begins after 3.5 weeks of a normal quarter, but we've managed to cover all the prior material in just two weeks. Treat yourself!

Why my sigint handler didn't prevent the program from exiting

In lecture last Friday, I gave the following code:

```
void handleSigint(int sig) {
    printf("haha not exiting! :)\n");
}

int main(int argc, char *argv[]) {
    signal(SIGINT, handleSigint);
    sleep(100);
    return 0;
}
```

It was supposed to prevent the program from exiting on Ctrl+C, but it didn't work. I did some investigating after lecture, and found this is because signals force some syscalls to return early. `sleep` is one of them (and a complete list is here, section "Interruption of system calls and library functions by signal handlers"). From the man page of `sleep`:

```
Return value:
```

```
Zero if the requested time has elapsed, or the number of seconds left
to sleep, if the call was interrupted by a signal handler.
```

So, there isn't anything necessarily wrong with our signal handling of SIGINT, but `sleep` returned when I didn't expect it to, causing the program to exit. I wrote a `delay` function that behaves closer to what we might expect:

```
void delay(unsigned int seconds) {
    unsigned int remaining = seconds;
    while (remaining > 0) {
        remaining = sleep(remaining);
    }
}
```

I won't expect you to understand the semantics of syscalls being interrupted by signals. If I can't get it right, I don't expect you to either! This is just interesting info that I learned myself this weekend.

What's with this EOF business?

When you call `read` on a file (a real file, on disk), it returns the number of bytes that were read from the file. You can call `read` several times, advancing the cursor each time, but when you reach the end of the file, `read` will return 0, indicating there is nothing left to read in the file. This shouldn't be surprising.

Pipes are files too, and `read` behavior is almost the same. If you call `read` on the read end of a pipe, it will return the number of bytes read from the pipe (or, if there hasn't been anything written yet, it will wait until some number of bytes have been written to the pipe, and then it will read those bytes). What's the "end of file" of a pipe? **When reading from a pipe, you reach the end of file when (1) you've read everything that has been written, and (2) it's not possible for anyone to write any more data to the pipe.** This has several implications:

- Until EOF, any `read` syscalls on a pipe will block/wait until they can return a positive number of bytes. `read` never returns 0 unless you're at the end of the file.
- You don't reach EOF until *every* file descriptor pointing to the write end of the pipe has been closed. If *any* process keeps a write file descriptor open, you won't reach EOF.

While working on assignment 2, you will probably experience your programs hanging at

least once. In pipeline/subprocess, the most common issue is forgetting to close a write file descriptor somewhere. Draw your file descriptor / file entry tables as you trace through your program, and make sure that there aren't any open descriptors when you expect otherwise. (Farm can hang for more complex reasons – it may be waiting for a worker to become available even though it already has workers available, or it might get stuck in `sigsuspend`, or other problems might occur.)

Signal handlers and scheduling

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

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

(Note that a program can send a signal to itself by calling `raise(int sig)`, which is equivalent to calling `kill(getpid(), sig)`.)

Consider the following code:

```
static pid_t pid;
static int counter = 0;

static void parentHandler(int unused) {
    counter++;
    printf("counter = %d\n", counter);
    kill(pid, SIGUSR1);
}

static void childHandler(int unused) {
    counter += 3;
    printf("counter = %d\n", counter);
}

int main(int argc, char *argv[]) {
    signal(SIGUSR1, parentHandler);
    if ((pid = fork()) == 0) {
        signal(SIGUSR1, childHandler);
        kill(getppid(), SIGUSR1);
        return 0;
```

```
    }

    waitpid(pid, NULL, 0);
    counter += 7;
    printf("counter = %d\n", counter);
    return 0;
}
```

What can this program output?

If the child process is around by the time `parentHandler` calls `kill`, then the output will be:

```
counter = 1
counter = 3
counter = 8
```

However, it's possible the child process might exit before the scheduler puts the parent back on the processor. In that case, the output will be:

```
counter = 1
counter = 8
```

Signal masks: blocking signals and `sigsuspend`

On Friday, I introduced `sigprocmask` as a way of deferring signals when it would be bad for a process to handle them. To recap, we can construct sets of signals:

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

In principle, the `sigset_t` is just a set of bits, where each bit position corresponds to a particular signal, and a bit will be `1` if the signal is in the set or a `0` if not. We can tell the OS to hold off on calling the signal handlers for any signals in a set:

```
sigprocmask(SIG_BLOCK, &mask, NULL);
```

And later, we can tell the OS that it's okay to deliver those signals now:

```
sigprocmask(SIG_UNBLOCK, &mask, NULL);
```

Signal masks

Those function invocations update the process's *signal mask*. The signal mask is a set of *blocked signals*, so `sigprocmask(SIG_BLOCK,...)` adds signals to the process's signal mask, and `sigprocmask(SIG_UNBLOCK,...)` removes signals from the process's signal mask. Whenever a signal comes in, the OS checks whether the signal is contained in the process's signal mask, and if it is, it defers delivery of the signal.

Any time you call `sigprocmask` to change the process's signal mask, you have the option to save the signal mask (before the change) to a variable via the third argument:

```
sigset_t additions, current;
sigemptyset(&additions);
sigaddset(&additions, SIGCHLD);

sigprocmask(SIG_BLOCK, &additions, &current);
// SIGCHLD has now been added to the process's signal mask. `current` is the
// set of signals that were blocked *before* SIGCHLD was added.
```

Waiting for signals to come in

Let's say we want to do something like the following:

```
while not all children have exited yet:
    wait for a SIGCHLD to be recieved (which might mean a child exited)
```

Let's say we have some function `sigsuspend()` that waits until a signal comes in. (There is a `sigsuspend` function, but it requires arguments, which I'll explain in a moment.) We could write the following:

```
while (numExitedChildren < numSpawnedChildren) {
    sigsuspend();
}
```

However, this has a problem. What if, immediately after the `numExitedChildren < numSpawnedChildren` test *but before the sigsuspend*, the last child exits? All child processes will have exited, so we won't have any more SIGCHLD signals come in, but we're about to call sigsuspend, putting us to sleep until a signal comes in. If this happens, the parent process will hang.

To avoid this issue, we need to block SIGCHLD while we're deciding whether to go to sleep:

```
sigset_t additions;
sigemptyset(&additions);
sigaddset(&additions, SIGCHLD);
sigprocmask(SIG_BLOCK, &additions, NULL);

while (numExitedChildren < numSpawnedChildren) {
    sigsuspend();
}

sigprocmask(SIG_UNBLOCK, &additions, NULL);
```

If we're deciding whether to go to sleep, the OS will postpone delivery of any SIGCHLD signals, avoiding the issue above. But this *still* hangs, because SIGCHLD is blocked while we're sleeping, so the parent doesn't wake up when its child exits. We need to unblock SIGCHLD once we have committed to going to sleep.

The real `sigsuspend` supports this. It takes a signal mask as a parameter, and it *atomically* sets the process's signal mask to that set *and* puts the process to sleep. (To do something atomically is to do something in a single, uninterruptible operation; this means that the process won't receive SIGCHLD in between unblocking it and going to sleep, which is the issue we originally had.) Before blocking SIGCHLD, we can save the process's signal mask, then restore it when going to sleep. (When coming out of sleep, `sigsuspend` will re-block SIGCHLD.)

```
sigset_t additions, current;
sigemptyset(&additions);
sigaddset(&additions, SIGCHLD);
sigprocmask(SIG_BLOCK, &additions, &current);

while (numExitedChildren < numSpawnedChildren) {
```

```
    sigsuspend(&current);
}

sigprocmask(SIG_UNBLOCK, &additions, NULL);
```