Lecture 17 - Inter-Process Communication (last updated: 11/09/22)

In the previous lecture, we saw how UNIX processes are created with fork(). With their memory address spaces separated, these processes cannot communicate with each other without the support of the operating system.

UNIX systems provide several mechanisms for inter-process communication (IPC). We have already seen one such mechanism: waitpid() allows a process to block until one of its children terminates, and can be used to read that child's exit status code. However, waitpid() is limited to transmitting a single integer from child to parent, and requires the child to terminate. In this lecture, we'll look at some other IPC mechanisms that allow concurrent processes to communicate with each other on UNIX systems.

UNIX Signals

UNIX processes can communicate with each other using signals. Signals are asynchronous, meaning a process can receive a signal at any point during its execution. They can be (but are not always) used to terminate processes under exceptional circumstances.

There are many different types of signals, some of which you should have already encountered; here are some that are useful to know about:

- SIGINT ("interrupt"): sent to foreground processes when you press Ctrl-C in the terminal, usually to manually kill a program; by default, SIGINT terminates the recipient process.

- SIGSEGV ("segmentation fault"): sent to processes that attempt to access invalid memory addresses (e.g., dereference a NULL pointer); by default, SIGSEGV terminates the recipient process.

- SIGABRT ("abort"): usually sent by a process to itself upon encountering a fatal error (e.g., failed assert()); by default, SIGABRT terminates the recipient process.

- SIGPIPE ("broken pipe"): sent to processes that write to file descriptors without any readers (e.g., writing to a severed socket connection); by default, SIGPIPE terminates the recipient process.

- SIGCHLD ("child terminated"): sent to the parent process when one its children terminates; by default, processes ignore SIGCHLD.

- SIGKILL ("kill"): forcibly terminate a process; SIGKILL cannot be overridden or ignored.

Each signal is identified by a signal number; for instance, SIGINT is 2 and SIGPIPE is 13. POSIX's system header files (e.g., unistd.h) #define signal names like SIGINT and SIGPIPE as constants according to their signal numbers.

Most signals are sent automatically by the kernel, but signals can also be sent using the kill() system call (not to be confused with SIGKILL, which is a specific signal):

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

You can also use the "kill" command to send signals from the command line.

For example, to forcibly terminate process number 61240, you can send it SIGKILL like this:

```
$ kill -SIGKILL 61240
```

If you do not know the PID of a process, you can also use the "pkill" command, which allows you to specify the name of the program/command the process is running. For example, to terminate all processes running a.out, run:

```
$ pkill -SIGKILL a.out
```

Note that processes only have permissions to send signals to processes of the same user, to prevent users from kill()ing each others' processes.

Handling signals

How a process handles a signal depends on its configured "disposition." For example, the default disposition for SIGINT is to terminate the recipient process, but many terminal text editors like Vim and Emacs override SIGINT's default disposition so that Ctrl+C can be used as an editor keybinding.

You can configure the disposition of a signal using the sigaction() system call:

```
int sigaction(int signum, const struct sigaction *action,
              struct sigaction *old_action);
```

The signum parameter specifies which signal to configure, while the action parameter specifies the configuration. The previous configuration for signum is written to old_action when old_action is non-NULL.

The sigaction configuration structure is defined approximately like this:

```
struct sigaction {
    void      (*sa_handler)(int);   // What to do upon receiving signal
    sigset_t  sa_mask;              // Ignore these signals while handling
    int       sa_flags;             // Additional options
    ...                             // Other fields not shown
};
```

The .sa_handler field specifies what action to take when the process receives a signal; if .sa_handler is set to a function's address, that function will be executed when the configured signal is received. .sa_handler can also be set to SIG_IGN to ignore the signal, or to SIG_DFL to restore the default disposition.

For example, we can configure a program to ignore SIGINT like this:

```
struct sigaction sa;

memset(&sa, 0, sizeof(sa));         // Zero-initialize sigaction structure
sigemptyset(&sa.sa_mask);           // Don't mask any signals
sa.sa_handler = SIG_IGN;            // Ignore the signal

sigaction(SIGINT, &sa, NULL);
```

After calling sigaction(), that process will no longer exit when you press Ctrl-C (though it can still be terminated using SIGKILL or SIGSEGV).

Reaping child processes asynchronously

Recall that when a process exits, its parent should use waitpid() to reap its terminated child. However, calling waitpid() in blocking manner prevents the parent from doing any other work in the meantime. Instead, the parent may periodically check for and reap terminated children by calling the following function, which polls waitpid() in a non-blocking manner:

```
void reap_children(int _unused_arg) {
    while (waitpid(-1, NULL, WNOHANG) > 0)
        ;
}
```

However, this approach can be wasteful and error-prone: calling reap_children() too often is unnecessary, while not calling reap_children() often enough may lead to an unchecked accumulation of zombies.

Instead, we can make use of SIGCHLD, which is sent to the parent process when a child terminates. Though this signal is ignored by default, we can install reap_children() as a SIGCHLD handler function to automatically reap child processes as soon as they terminate:

```
struct sigaction sa;

memset(&sa, 0, sizeof(sa));         // Zero-initialize sigaction structure
sigemptyset(&sa.sa_mask);           // Don't mask any signals
sa.sa_flags = SA_RESTART;           // Restart interrupted system calls
sa.sa_handler = &reap_children;     // Reap children upon receiving signal

sigaction(SIGCHLD, &sa, NULL);
```

By default, blocking system calls such as read() and accept() are unblocked when a signal handler is invoked. To prevent such behavior, we specify the SA_RESTART flag to ensure they are automatically "restarted" so that the parent process continues to block for as long as it needs.

One subtle detail here is that the handler must call waitpid() in a non-blocking loop, in order to account for situations where there are multiple terminated children. Signals do not stack: if multiple children happen to terminate in quick succession, only a single signal is delivered to the parent, so its SIGCHLD handler will only get to execute once.

Pipes (optional)

Although signals can be used to asynchronously communicate with arbitrary processes, they only convey a limited amount of information (the signal number). To send streams of arbitrary data, processes may instead communicate using a pipe, a special kind of file that represents a unidirectional communication channel. Pipes are created using the pipe() system call, which produces two file descriptors: one for the read end of the pipe, and one for the write end of the pipe:

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

Pipes are typically created prior to fork()ing, with one process writing data into one end of the pipe and reading data from the other end. In the following example, the parent sends "hello" to its child through a pipe:

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

if (fork() == 0) {
    // Child process

    close(pipefd[1]);       // Close write end of pipe

    char buf[8];

    int len = read(pipefd[0], buf, sizeof(buf) - 1);
    buf[len] = '\0';

    printf("Read from pipe: %s\n", buf);

    // ...

} else {
    // Parent process

    close(pipefd[0]);       // Close read end of pipe

    write(pipefd[1], "hello", 5);

    // ...

}
```

Though pipes may seem like an alien construct, you should have already encountered them in the command line! For instance, when you construct a shell pipeline like this:

```
$ yes | head
```

The shell constructs a pipe() that is shared between two child processes, which respectively exec() the "yes" and "head" commands. The pipe's file descriptors are redirected to the stdout and stdin of those child processes, allowing them to communicate via the pipe.

Incidentally, these kinds of pipelines are why SIGPIPE exists. Here, the "yes" command prints a never-ending stream of lines consisting of the "y" character, while the "head" command extracts just the first 10 lines before terminating. Once the "head" process exits, subsequent writes by the "yes" process will trigger a SIGPIPE, terminating the "yes" process as well rather than leaving it to write indefinitely into a now-broken pipe.

At a high level, pipes work very similarly to sockets: they are both special kinds of files that can be used for inter-process communications. However, pipes are strictly unidirectional and can only be used for IPC; on the other hand, sockets are bidirectional and can also be used for various forms of network I/O.