CS 24: Introduction to Computing Systems

Signals

Last time, we talked about how user code can ask the kernel to do things, but how does the kernel notify user code that something important has happened? This time we will cover the mechanism that makes this possible called *signals*.

Signals

A *signal* is a short message from the kernel that notifies a process that a system-level event has happened. Signals can be handled by user code or can apply a default action. Here is a short list of some signals:

- **Interrupt** (`SIGINT`). This signal is sent when you type Control-C on your keyboard. By default, this signal terminates the program, but it can be customized to do something different.
- **Kill** (`SIGKILL`). This signal is sent to a process with the intention of terminating it immediately. This signal cannot be caught or ignored. It is the ultimate way of killing a program.
- **Bad Arithmetic** (`SIGFPE`). This signal is sent when the processor has tried to execute an erroneous arithmetic instruction (e.g., division by zero).
- **Segmentation Violation** (`SIGSEGV`). This signal is sent when a program tries to access memory that it doesn't have access to.
- **Alarm** (`SIGALRM`). This signal is sent when a "timer" set with the `alarm` function expires.
- **Child** (`SIGCHLD`). This signal is sent to a parent process when one of its children terminates.

Signals do not "queue"; at all times, there is either no pending signal of a particular type or just one. Signal handlers need to be aware of this fact, as they can't expect to get one signal per event.

Causes of Signals

Signals can get sent as a result of various functions and system events. The most common are listed here:

- **System Exceptions.** Most signals can be sent to a process by the kernel as a result of an exception. In particular, `SIGFPE` and `SIGSEGV` tend to both be raised in this way.
- **The `kill` Function.** The `kill` function takes a signal and a process as arguments and send that signal to that process. In other words, you can explicitly send a signal to another process via the `kill` function.
- **Keyboard.** The terminal sends signals on certain keypress combinations (such as Control-C or Control-Z). This is the most common way `SIGINT` is raised.
- **The `alarm` Function.** The `alarm` function raises a `SIGALRM` when a timer expires. This can be useful to get a process to do something after a certain amount of time.

Some Simple Signal Handlers

A *signal handler* is just a function which returns `void` and takes a single `int`, which represents the signal number, as an argument. To *install* a signal handler in a process, you use the `signal` function which takes the signal and function pointer as arguments.

Segmentation Fault... Nah...

In this first example of a signal handler, we catch segmentation faults and print a message instead of ending the process.

```c
void haha(int sig) {
   // When we get a SIGSEGV, print out a message instead
   char *msg = "haha! segfault!\n";
   write(STDOUT_FILENO, msg, strlen(msg));
}

int main() {
   // This installs haha as a signal handler for SIGSEGV
   signal(SIGSEGV, haha);

   // This forces a dereference of NULL
   *(volatile char *)NULL;
}
```

If you run this program, you will see that it repeatedly prints out the message. This is because, after executing the signal handler, the program goes back to the instruction it failed on.

Timing Out a Program

In this second example, we run a program and kill it if it's still running after 5 seconds. This could be useful in an autograder, for example.

```c
static pid_t pid;

static void kill_child(int sig) {
   // Sends a SIGKILL to the child process to immediately kill it
   kill(pid, SIGKILL);
}

int main(int argc, char *argv[]) {
   pid = fork();

   if (pid == 0) {
      printf("execing program...\n");
      execvp(argv[1], &argv[1]);
      abort();
   }

   printf("parent running...\n");
   signal(SIGALRM, kill_child);
   alarm(5);

   wait(NULL);
   printf("child reaped.\n");
}
```

Signal Safety

Consider the following program:

```c
void sigint_handler(int sig) {
   malloc(10000);
}


int main() {
   signal(SIGINT, sigint_handler);
   while (true) {
      char *ptr = calloc(100000000, 1);
      free(ptr);
      printf("Going...\n");
   }
}
```

If you run this program on `labradoodle` and type control-c a bunch of times, you will see that the program hangs. Signals are a form of *asynchronous* execution, and, as a result, there are only a small number of functions allowed in signal handlers to avoid concurrency issues. In particular, `malloc` and `printf` are both not allowed, because they have state that might deadlock the program or output the wrong thing.

A Silly Example

Signal handlers *can* get more information than just the signal number if you use `sigaction` instead of `signal`. In particular, this program skips over a divide by zero by setting `%rip` to a label after it.

```
1  extern char label[];
2  const int zero = 0;
3
4  static void sigfpe_handler(int signum, siginfo_t *siginfo, void *context) {
5     ucontext_t *ucontext = (ucontext_t *)context;
6     ucontext->uc_mcontext.gregs[REG_RIP] = (greg_t)label;
7  }
8
9  int main() {
10    struct sigaction act = {
11       .sa_sigaction = sigfpe_handler,
12       .sa_flags = SA_SIGINFO
13    };
14    sigaction(SIGFPE, &act, NULL);
15
16    int what = 10 / zero;
17
18    asm volatile("label:");
19    printf("Hello from the dead!\n");
20 }
```