CSE 410: Systems Programming

POSIX Signals

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo

POSIX Signals

POSIX signals are another form of interprocess communication.

They are also a way to create concurrency in programs.

For these two reasons, they are rather complicated and subtle!

Signals provide a simple message passing mechanism.

©2018 Ethan Blanton / CSE 410: Systems Programming

Signals as Messages

POSIX signals are asynchronous messages.

Asynchronous means that their reception can occur at any time.[1]

The message is the reception of the signal itself.

Each signal has a number, which is a small integer.

POSIX signals carry no other data.

[1] Almost. We'll see how to control it later.

©2018 Ethan Blanton / CSE 410: Systems Programming

Signal Types

There are two basic types of POSIX signals:

- Reliable signals
- Real-time signals

Real-time signals are much more complicated.

In particular, they can carry data.

We will discuss only reliable signals in this lecture.

©2018 Ethan Blanton / CSE 410: Systems Programming

Asynchronous Reception

From the point of view of the application:

- Signals can be blocked or ignored
- Enabled signals may be received between any two processor instructions
- A received signal can run a user-defined function called a signal handler

This means that enabled signals and program code must very carefully manipulate shared or global data!

©2018 Ethan Blanton / CSE 410: Systems Programming

Signals

POSIX defines a number of signals by name and number.

A few of those are:

- `SIGHUP`, 1 (sent when a terminal disconnects)
- `SIGINT`, 2 (sent when you push Ctrl-C)
- `SIGKILL`, 9 (uncatchable, terminates the process)
- `SIGSEGV`, 11 (sent on invalid memory access)
- `SIGCHLD`, 17 (sent when a child process exitsx)

©2018 Ethan Blanton / CSE 410: Systems Programming

Signal Handlers

A process indicates that it wishes to receive a signal by installing a signal handler.

Each signal has a default handler that either:

- Ignores the signal
- Stops the process
- Continues the process
- Terminates the process
- Terminates the process and dumps core

©2018 Ethan Blanton / CSE 410: Systems Programming

Signal Handlers II

A process can install a signal handler for any signal except:

- SIGKILL
- SIGSTOP

A signal handler is a function.

That function is called when the signal is received.

Signal handlers are of type `sighandler_t`:

```
typedef void (*sighandler_t)(int);
```


©2018 Ethan Blanton / CSE 410: Systems Programming

Typedef

```
typedef void (*sighandler_t)(int);
```

A typedef declares a new type.

It looks like a variable declaration.
The name of the variable becomes the type.

`sighandler_t` is a function pointer.

It is a function returning `void` and accepting one `int` argument.

©2018 Ethan Blanton / CSE 410: Systems Programming

Installing a Handler

```
sighandler_t signal(int signum, sighandler_t handler);
```

The `signal` function accepts

- a signal number and
- a handler function

and binds the function to the signal.

Therefter, receipt of the signal will call the bound function.

It also returns the old signal handler.

©2018 Ethan Blanton / CSE 410: Systems Programming

Special Handlers

There are two special signal handlers:

- SIG_IGN: ignore a signal
- SIG_DFL: restore default behavior

These values may be passed to signal instead of a function.

```
signal(SIGCHLD, SIG_DFL);
```


©2018 Ethan Blanton / CSE 410: Systems Programming

Signal Portability

The `signal()` function has some portability problems:

- Some systems reset the handler to `SIG_DFL` upon receipt
- Some systems allow signals to arrive during a handler set by `signal()`

For this reason, there is a POSIX function `sigaction()`.

The behavior of `sigaction()` is more tightly defined.

Linux `signal()` semantics are appropriate for your shell project.

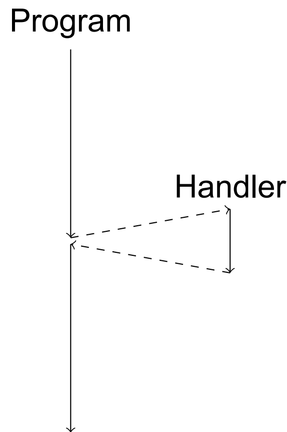
©2018 Ethan Blanton / CSE 410: Systems Programming

Signal Reception

The kernel may deliver a signal at any time.

When receiving a signal, a process will:

- Push its current program counter onto the stack
- Jump to the signal handler
- Execute the signal handler
- Pop the saved PC and return



©2018 Ethan Blanton / CSE 410: Systems Programming

Blocking Signals

A signal can be blocked by the program.

A blocked signal will be delivered when it is unblocked.

Signals may be:

- implicitly blocked because a handler for that signal is currently executing
- explicitly blocked by the programmer using `sigprocmask()`

Signal blocking allows the program to restrict signal reception, since it otherwise cannot predict when they will be received.

©2018 Ethan Blanton / CSE 410: Systems Programming

sigprocmask()

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set,
    sigset_t *oldset);
int sigemptyset(sigset_t *set);
int sigaddset(sigset_t *set, int signum)
```

`sigprocmask()` blocks or unblocks signals given by a `signal set`.

A `signal set` contains zero or more signals.

©2018 Ethan Blanton / CSE 410: Systems Programming

Using sigprocmask()

```
sigset_t mask, oldmask;

sigemptyset(mask);
sigaddmask(mask, SIGCHLD);
sigprocmask(SIG_BLOCK, &mask, &oldmask);

/* SIGCHLD is blocked here */

sigprocmask(SIG_SETMASK, &oldmask, NULL);

/* SIGCHLD restored to its state before the block */
/* If it is unblocked and pending, the handler will
   run now (or shortly). */
```


©2018 Ethan Blanton / CSE 410: Systems Programming

Shared Data

Signals introduce concurrency into programs.

Because signal handlers run at unpredictable times, accessing shared data from signal handlers is dangerous.

If data is in an inconsistent state when a handler accesses it, corruption or program errors might occur.

©2018 Ethan Blanton / CSE 410: Systems Programming

Signal Concurrency

```
void prepend(struct ListNode *node) {
    node->next = list;
    list = node;
}

void handler(int sig) { prepend(new_listnode()); }

int main(int argc, char *argv[]) {
    signal(SIGINT, &handler);
    prepend(new_listnode());

    return 0;
}
```


©2018 Ethan Blanton / CSE 410: Systems Programming

Corruption

```
1 void prepend(struct ListNode *node) {
2     node->next = list;
3     list = node;
4 }
```

If the signal arrives …	*The result is …*
Before line 2	The list contains 2 items
Between 2 & 3	The list contains only the main node
After 3	The list contains both nodes

If the handler node is lost, memory is leaked.

©2018 Ethan Blanton / CSE 410: Systems Programming

Sending Signals

Signals are sent to a process with the `kill()` function.

```
#include <sys/types.h>
#include <signal.h>

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

Note that kill may or may not actually kill the receiving process!

A process can generally only kill:

- itself
- other processes owned by the same user

©2018 Ethan Blanton / CSE 410: Systems Programming

Summary

- Signals are interprocess communication.
- Each signal is a message.
- Signals are handled by functions.
- Signal handlers introduce concurrency.
- Shared data must be manipulated carefully when signals are in use.

©2018 Ethan Blanton / CSE 410: Systems Programming

Next Time …

- …

© 2018 Ethan Blanton / CSE 410: Systems Programming

References I

Required Readings

[1] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 8: 8.5. Pearson, 2016.

Optional Readings

[2] "Overview of Signals". In: *Linux Programmer's Manual*. `man 7 signal`.

©2018 Ethan Blanton / CSE 410: Systems Programming

License

Copyright 2018 Ethan Blanton, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see https://www.cse.buffalo.edu/~eblanton/.