CS415 Module 2 Part D - Signals

Athens State University

Contents

1 Setting a signal handler

The `signal()` system call

- The POSIX standard defines two system calls for changing the disposition of signals: `signal()` and `sigaction()`
- `signal()` was the original interface and is the simpler of the two system calls
- But it is not consistently implemented across different versions of UNIX

The `signal()` system call

```
#include <signal.h>
void (*signal(int sig, void (*handler)(int)));
```

- Returns previous signal disposition on success or `SIG_ERR` on error
- The odd appearing syntax in the prototype is required because of the second parameter is a pointer to a function.
- Note that the return value is a function pointer that

The `handler` is the address of the function that needs to be called when this signal is delivered. Signal handlers take the following form:

```
void handler(int sig)
{
   /* Code for the handler goes here
}
```

The return value for the system call has to be a function pointer to the old handler so that the application can, if required, reset the handler back to the old function.

Resetting the signal handler

```
void (*oldHandler)(int);
oldHandler = signal(SIGINT, newHandler);
if (oldHandler == SIG_ERR)
   errExit("signal");
//Do something else here. During this time, if SIGINT is
//delivered, newHandler will be used to handle the signal.
if (signal(SIGINT, oldHandler) == SIG_ERR)
   errExit("signal")
```

Harness the power of `typedef`

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int sig, sighandler_t handler)
```

2 Signal Handler Internals

Signal Delivery and Handler Execution

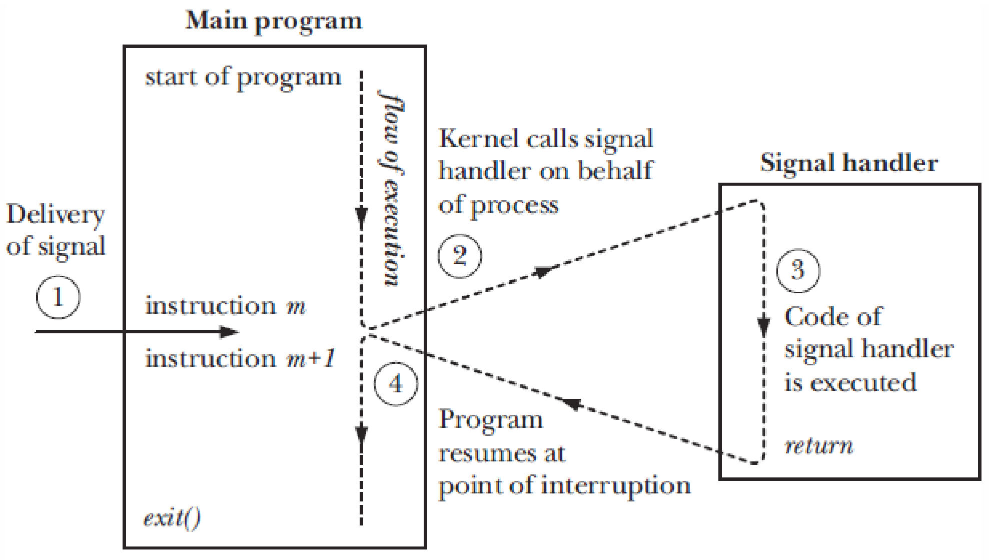
Main program
start of program
flow of execution
instruction m
instruction m+1
exit()
Delivery of signal
1
Kernel calls signal handler on behalf of process
2
Signal handler
3
Code of signal handler is executed
return
4
Program resumes at point of interruption

Figure 20-1: Signal delivery and handler execution

Case Study: Catching terminate signals

```
#include <signal.h>
void sigHandler(int sig) {
  static int count = 0;
  if (sig == SIGINT) {
    count++;
    printf("Caught SIGINT (%d)\n", count);
    return;
  }
  printf("Caught SIGQUIT - Punt\n");
  exit(EXIT_SUCCESS);
}

int main(int argc, char *argv[]) {
  if (signal(SIGINT, sigHandler) == SIG_ERR)
     errExit("signal SIGINT");
  if (signal(SIGQUIT, sigHandler) == SIG_ERR)
     errExit("Signal SIGQUIT");
  while(1) pause();
}
```

3 The `sigaction()` system call

The `sigaction()` system call

```
#include<signal.h>
int sigaction(int sig, const struct sigaction *act,
                         struct sigaction *oldact);

struct sigaction {
  void (*sa_handler)(int); /* Address of handler */
  sigset_t sa_mask; /* Signals blocked during handler invocation */
  int sa_flags; /* Flags controlling handler invocation */
  void (*sa_restorer)(void); /* Not for application use */
};
```

So, the use case here is first create an instance of `sigaction` struct with details about our signal handler and its behavior.

The `sa_handler` field is a function pointer to the handler function. The `sa_mask` field is a bitmask that will be bitwise-OR'ed with the process signal mask just before the signal handler is called and removed just after signal handler returns. The `sa_flags` field is a bitmask that allows one to adjust the behavior of handler.

4 Signal Handler Design Issues

Signals are not queued!

- Note that the default behavior of signals is that a signal is blocked while a signal handler is executing
- In that case, that single instance of a signal is set to pending
- But all other instances of a signal arriving while the signal handler is executing are dropped.
- Means that we can't reliably count the number of times a signal is generated
- And we need to code signal handlers to deal with the chance that multiple events of the type corresponding to the signal have occurred

Reentrant Functions

- Classical UNIX program have a single thread of execution
- Modern UNIX programs can have multiple threads of execution
- A function is `reentrant` if can be executed by multiple threads of execution in the same process

Async-signal-safe functions

- **Async-signal-safe function** A function that is either re-entrant or uninterruptible by a signal handler
  - This means that we can safely use such functions within a signal handler
  - The POSIX system calls and functions that async-signal-safe are listed in Table 21-1 in the Kerrisk textbook
- Note that some important functions are not in this category
  - The input/output functions (such as `printf`/`scanf`) family of functions)
  - The functions that manage the heap (`malloc()`/`free()`)

What this implies for signal handlers

- Work as much as possible to not call unsafe functions in signal handlers
- If you must, then note that you will have to block the delivery of signals while executing code in the main program calls the unsafe functions or modifies global data structures also used by the signal handler

Designing signal handlers

- Keep them simple... the KISS principle is definitely in effect!
- Two approaches
  - Set a global flag and exit. Let the main program do clean-up
  - Make the signal handler do what app clean-up it can and then terminate the process

Note that it's common with the second approach to use a non-local GOTO (`setjmp()` and `longjmp()`) to bounce back to the main part of the program. This is fraught with much danger and should be avoided unless absolutely required in your application.

Signal handlers in C++

```
#include <csignal>
#include <iostream>
namespace {
   volatile std:sig_atomic_t gSignalStatus;
}

void signalHandler(int signal) {
   gSignalStatus = signal;
}

int main() {
   std:signal(SIGINT, signalHandler);
   std::cout << "SigVal: " << gSignalStatus << std::endl;
   std::raise(SIGINT);
   std::cout << "SigVal: " << gSignalStatus << std::endl;
}
```

Signal handlers in C++

- Limitations imposed on signal handlers in C++:
  - The signal handler cannot raise a signal if called as result of a `std:abort()` or `std:raise()` system call
  - Do not use floating-point variables in C++ signal handler
  - The `sig:signal()` system call is not thread-safe.
  - Signal handlers must have C-linkage and only use features common to both C and C++.

5 Key Points

Key Points

- What are signals?
- How are they processed?
- What can we do with them?