Advanced Signal Handling

COMP 321

Dave Johnson

1

Review: Changing the Behavior for Some Signal

```
int  sigaction(int  signum,
      const struct sigaction * _Nullable restrict act,
      struct sigaction * _Nullable restrict oldact);
```

A process can change the behavior and/or check current behavior of a signal

- "signum" specifies which signal
- "act" points to a struct defining the desired new bahavior when that type of signal is received by this process
  - If NULL, no change to signal's behavior is made
- "oldact" points to a struct that returns current setting of this signal's behavior
  - If NULL, the current setting for this signal is not returned

2

Two Ways a Signal Handler Can Be Called

A function with 1 argument

```
void  handler(int  sig);
```

- The handler gets told which signal caused it to be called
- So its easy to have a single handler for more than one type of signal

A function with 3 arguments

```
void  handler(int  sig,  siginfo_t  *info,  void  *ucontext);
```

- info tells the handler a lot more information about what happened
- ucontext tells the handler even more about what happened

Review: Dealing with a struct sig_action

A struct sig_action has several fields, some of which overlap or conflict

- Best practice is to
  - Zero out the entire struct sig_action contents (e.g., with memset)
  - Then set just the fields in it that you need
- The most important fields are sa_handler and sa_action
  - Both define handling for that type of signal, but use only one, not both

The sa_handler and sa_action fields are the address of the handler function

- Don't try to use both in a single call to sigaction()

Controlling Which Way a Handler is Called

The sa_flags field in the struct sigaction controls which gets used

```
struct sigaction {
    void    (*sa_handler)(int);
    void    (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int     sa_flags;
    void    (*sa_restorer)(void);
};
```

approximate definition of struct sigaction

- If SA_SIGINFO is set in sa_flags, then sa_sigaction is used
- Otherwise, the simpler sa_handler is used

An Example for SIGSEGV

```
int
main(void)
{
        struct sigaction newact;

        memset(&newact, 0, sizeof(newact));
        newact.sa_flags = SA_RESTART | SA_SIGINFO;
        newact.sa_sigaction = example_handler;
        sigaction(SIGSEGV, &newact, NULL);

        *(int *)0x123 = 0x456;

        exit(0);
}
```

For testing: cause a segmentation fault!

An Example for SIGSEGV

```
void
example_handler(int sig, siginfo_t *info, void *uc)
{
        fprintf(stderr, "example_handler: signal %d code %d occurred\n",
                sig, info->si_code);
        fprintf(stderr, "example_handler: addr %p\n", info->si_addr);

        exit(1);
}
```

If we just returned, the SIGSEGV would reoccur!

This prints

```
example_handler: signal 11 code 1 occurred
example_handler: addr 0x123
```


Easily Printing Signal Information

```
void  psignal(int  sig,  const char  *s);
void  psiginfo(const siginfo_t  *pinfo,  const char  *s);
```

Prints information to stderr

- psignal() prints a message describing signal sig
- psiginfo() prints relevant information from the given siginfo_t
- Both prefix the information printed with the string from s (e.g., a label) and a colon and space
  - "s: signal information"
  - But if s == NULL, the colon and space (and s) are omitted

An Example for SIGSEGV

```
void
example_handler(int sig, siginfo_t *info, void *uc)
{
        psignal(sig, "example_handler");
        psiginfo(info, "example_handler");

        exit(1);
}
```

We really only want one of these two

If we just returned, the SIGSEGV would reoccur!

This prints

example_handler: Segmentation fault
example_handler: Segmentation fault (Address not mapped to object [0x123])

Even More Information in a Signal Handler

```
#define _GNU_SOURCE                    // Needed to get definition of REG_RIP, etc.
#include <sys/ucontext.h>

void example_handler(int sig, siginfo_t *info, void *uc)
{
        mcontext_t *mc = &(((ucontext_t *)uc)->uc_mcontext);

        fprintf(stderr, "program counter = %p\nstack pointer = %p\n",
                mc->gregs[REG_RIP], mc->gregs[REG_RSP]);
        fprintf(stderr, "rdi = %p\nrsi = %p\nrdx = %p\n",
                mc->gregs[REG_RDI], mc->gregs[REG_RSI], mc->gregs[REG_RDX]);
        fprintf(stderr, "rcx = %p\nr8 = %p\nr9 = %p\n",
                mc->gregs[REG_RCX], mc->gregs[REG_R8], mc->gregs[REG_R9]);

        exit(1);
}
```

A More Interesting Example

```
#define PAGES           16
void *  memory;
int pagesize;

int main(void)
{
        . . .
        sigaction(SIGSEGV, &newact, NULL);
        pagesize = getpagesize();
        memory = mmap(NULL, PAGES * pagesize, PROT_NONE,
                MAP_ANON | MAP_PRIVATE,  0, 0);
        *(int *)(memory + 8000) = 0x456;
        printf("it worked: 0x%x\n", *(int *)(memory + 8000));
        exit(0);
}
```


A More Interesting Example

```
void example_handler(int sig, siginfo_t *info, void *uc)
{
        void *addr = info->si_addr;

         psiginfo(info, "example_handler");

        if (addr < memory || addr >= memory + PAGES * pagesize) {
                fprintf(stderr, "Address out of range\n");
                exit(1);
        }

        addr = (void *)((uintptr_t)(addr) & ~(pagesize-1));    // round down to page

        if (mprotect(addr, pagesize, PROT_READ|PROT_WRITE) < 0)
                err(1, "mprotect");
}
```

What Stack Is Used to Call a Signal Handler

A signal handler is normally called using the regular stack

- There's no actual procedure call instruction there, but it is a procedure call
- The return address needs to be saved
- The handler needs to have local variables

But sometimes, you can't or don't want to use the regular stack, such as

- You can't use it, e.g., if it is currently not writable (see the previous example)
- Or you want to be sure not to modify even parts of the stack memory that are currently beyond what is now on the stack (based on the stack pointer)
  - Example: Memory still within the same last page of the stack space
  - Example: Even more cautious, leave no trace at all that handler was called

Handling Signals Using An "Alternate" Stack

```
int  sigaltstack(const stack_t  *_Nullable restrict ss,
        stack_t  *_Nullable restrict  old_ss);
```

Specifies an alternate stack to use for calling signal handlers

- ss specifies the new alternate signal handler stack, old_ss returns the current

```
typedef struct {
    void    *ss_sp;     // the base (i.e., lowest) address of this stack
    int      ss_flags;  // flags are normally just 0
    size_t   ss_size;   // the number of bytes in the stack
} stack_t;
```

- Must first call sigaltstack() to define the alternate signal stack
- And must specify SA_ONSTACK in sa_flags for sigaction() in setting up handler