System Programming for Linux Containers

Signals: Signal Handlers

Michael Kerrisk, man7.org © 2020

mtk@man7.org

February 2020

Outline

Outline

Keep it simple

- Signal handlers can, in theory, do anything
- But, complex signal handlers can easily have subtle bugs (e.g., race conditions)
  - E.g., if main program and signal handler access same global variables
- ⇒ Avoid using signals if you can
  - ⚠ Don't introduce them as a means of IPC
  - ⚠ Don't use as part of a library design
    - (That would imply a contract with main program about which signals library is allowed to use)
- But, in some cases, we must deal with signals sent by kernel
  - ⇒ Design the handlers to be as simple as possible

Keep it simple

- Some simple signal-handler designs:
  - Set a global flag and return
    - Main program periodically checks (and clears) flag, and takes appropriate action
  - Signal handler does some clean-up and terminates process
    - (TLPI §21.2)
  - Signal handler performs a nonlocal goto to unwind stack
    - *sigsetjmp()* and *siglongjmp()* (TLPI §21.2.1)
    - E.g., some shells do this when handling signals

Signals are not queued

- Signals are not queued
- A blocked signal is marked just once as pending, even if generated multiple times
- ⇒ **One signal may correspond to multiple "events"**
  - Programs that handle signals must be designed to allow for this
- Example:
  - `SIGCHLD` is generated for parent when child terminates
  - While `SIGCHLD` handler executes, `SIGCHLD` is blocked
  - Suppose **two** more children terminate while handler executes
  - Only one `SIGCHLD` signal will be queued
  - Solution: `SIGCHLD` handler should loop, checking if multiple children have terminated

Outline

Reentrancy

- Signal handler can interrupt a program at *any* moment
  - ⇒ handler and main program are *semantically* equivalent to two *simultaneous* flows of execution inside process
    - (Like two "threads", but not the same as POSIX threads)
- A function is **reentrant** if it can safely be simultaneously executed by multiple threads
  - Safe == function achieves same result regardless of state of other threads of execution

[TLPI §21.1.2]

Nonreentrant functions

Functions that update global/static variables are **not** reentrant:

- Some functions by their nature operate on global data
  - e.g., *malloc()* and *free()* maintain a global linked list of free memory blocks
    - Suppose main program is executing *free()* and is interrupted by a signal handler that also calls *free()*...
    - Two "threads" updating linked list at same time ⇒ chaos!
- Functions that return results in statically allocated memory are nonreentrant
  - e.g., *getpwnam()* and many other functions in C library
- Functions that use static data structures for internal bookkeeping are nonreentrant
  - e.g., *stdio* functions do this for buffered I/O

Nonreentrant functions

- C library is rife with nonreentrant functions!
  - Man pages usually note functions that are nonreentrant

Async-signal-safe functions

- An async-signal-safe function is one that can be safely called from a signal handler
- A function can be async-signal-safe because either
  - It is reentrant
  - It is not interruptible by a signal handler
    - (Atomic with respect to signals)
- POSIX specifies a set of functions required to be async-signal-safe
  - See *signal-safety(7)* or TLPI Table 21-1
  - Set is a *small* minority of functions specified in POSIX
- No guarantees about functions not on the list
  - ⚠ *stdio* functions are **not** on the list

[TLPI §21.1.2]

Signal handlers and async-signal-safety

- Executing a function inside a signal handler is unsafe only if handler interrupted execution of an unsafe function
- ⇒ Two choices:
  1. Ensure that signal handler calls only async-signal-safe functions
  2. Main program blocks signals when calling unsafe functions or working with global data also used by handler
- Second choice can be difficult to implement in complex programs
  - ⇒ Simplify rule: call only async-signal-safe functions inside a signal handler

Signal handlers can themselves be nonreentrant

- ⚠ Signal handler can also be nonreentrant if it updates global data used by main program
- A common case: handler calls functions that update *errno*
- Solution:

```
void
handler(int sig)
{
    int savedErrno;
    savedErrno = errno;

    /* Execute functions that might
       modify errno */

    errno = savedErrno;
}
```


The *sig_atomic_t* data type

- Contradiction:
  - Good design: handler sets global flag checked by *main()*
  - Sharing global variables between handler & *main()* is unsafe
    - Because accesses may not be atomic

[TLPI §21.1.3]

The *sig_atomic_t* data type

- POSIX defines an integer data type that can be safely shared between handler and *main()*:
  - *sig_atomic_t*
  - Range: `SIG_ATOMIC_MIN..SIG_ATOMIC_MAX` (`<stdint.h>`)
  - Read and write guaranteed atomic
  - ⚠ Other operations (e.g., `++` and `--`) **not** guaranteed atomic (i.e., not safe)
  - Specify `volatile` qualifier to prevent optimizer tricks

```
volatile sig_atomic_t flag;
```


Exercises

1. Examine the source code of the program `signals/unsafe_printf.c`, which can be used to demonstrate that calling *printf()* both from the main program and from a signal handler is unsafe. The program performs the following steps:
   - Establishes a handler for the `SIGINT` signal (the control-C signal). The handler uses *printf()* to print out the string "`sssss\n`".
   - After the main program has established the signal handler, it pauses until control-C is pressed for the first time, and then loops forever using *printf()* to print out the string "`mmmmm\n`"

   Before running the program, start up two shells in separate terminal windows as follows (the *ls* command will display an error until the `out.txt` file is actually created):

```
$ watch ps -C unsafe_printf
```

```
$ cd signals
$ watch ls -l out.txt
```

Exercises

In another terminal window, run the *unsafe_printf* program as follows, and then hold down the control-C key **continuously**:

```
$ cd signals
$ ./unsafe_printf > out.txt
^C^C^C
```

Observe the results from the *watch* commands in the other two terminal windows. After some time, it is likely that you will see that the file stops growing in size, and that the program ceases consuming CPU time because of a deadlock in the *stdio* library. Even if this does not happen, after holding the control-C key down for 15 seconds, kill the program using control-\.

Inside the `out.txt` file, there should in theory be only lines that contain "`mmmmm\n`" or "`sssss\n`". However, because of unsafe executions of *printf()*, it is likely that there will be lines containing other strings. Verify this using the following command:

```
$ egrep -n -v '^(mmmmm|sssss)$' < out.txt
```


Exercises

2. Examine the source code of `signals/unsafe_malloc.c`, which can be used to demonstrate that calling *malloc()* and *free()* from both the main program and a signal handler is unsafe. Within this program, a handler for `SIGINT` allocates multiple blocks of memory using *malloc()* and then frees them using *free()*. Similarly, the main program contains a loop that allocates multiple blocks of memory and then frees them.

   In one terminal window, run the following command:

   ```
   $ watch -n 1 ps -C unsafe_malloc
   ```

   In another terminal window, run the *unsafe_malloc* program, and then hold down the control-C key until either:

   - you see the program crash with a corruption diagnostic from *malloc()* or *free()*; or
   - the *ps* command shows that the amount of CPU time consumed by the process has ceased to increase, indicating that the program has deadlocked inside a call to *malloc()* or *free()*.

Outline

Interrupted system calls

- What if a signal handler interrupts a blocked system call?
- Example:
  - Install handler for (say) `SIGALRM`
  - Perform a *read()* on terminal that blocks, waiting for input
  - `SIGALRM` is delivered
  - What happens when handler returns?
- *read()* fails with `EINTR` ("interrupted system call")
- Can deal with this by manually restarting call:

```
while ((cnt = read(fd, buf, BUF_SIZE)) == -1
                && errno == EINTR)
    continue;   /* Do nothing loop body */
if (cnt == -1)  /* Error other than EINTR */
    errExit("read");
```

[TLPI §21.5]

Automatically restarting system calls: SA_RESTART

- Specifying SA_RESTART in *sa_flags* when installing a handler causes system calls to *automatically* restart
  - SA_RESTART is a per-signal flag
- More convenient than manually restarting, but...
  - Not all system calls automatically restart
  - Set of system calls that restart varies across UNIX systems
  - (Origin of variation is historical)

Automatically restarting system calls: SA_RESTART

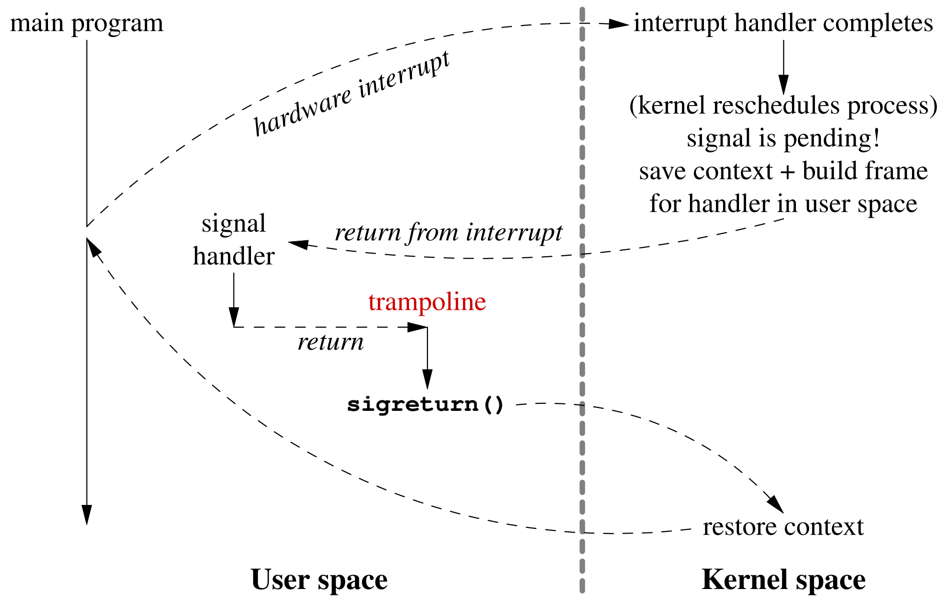
- Most (all?) modern systems restart at least:
  - *wait()*, *waitpid()*
  - I/O system calls on "slow devices"
    - i.e., devices where I/O can block (pipes, sockets, ...)
    - *read()*, *readv()*, *write()*, *writev()*
- On Linux:
  - Certain other system calls also automatically restart
  - Remaining system calls never restart, regardless of SA_RESTART
  - See TLPI §21.5 and *signal(7)* for details
- **Bottom line**: If you need cross-system portability, omit SA_RESTART and always manually restart

Outline

The problem

- Before executing signal handler, kernel must modify some *kernel-maintained* process context
  - Signal mask, signal stack (*sigaltstack()*)
  - (Registers will also be modified during handler execution, and so must be saved)
  - *Easy, because kernel has control at this point*
- Upon return from signal handler, previous context must be restored
  - *But, at this point we are in user mode; kernel has no control*
- **How does kernel regain control in order to restore context?**
  - ⇒ the “signal trampoline”

The "signal trampoline"



The kernel uses the signal trampoline to arrange that control is bounced back to kernel after execution of signal handler

When is a signal delivered?

- In a moment, we consider what's required to execute a signal handler
- But first of all, when is a signal delivered?
  - Signals are asynchronously delivered to process, but...
  - Only on transitions from kernel space back to user space

Steps in the execution of a signal handler

The following steps occur in the execution of a signal handler:

- A hardware interrupt occurs
  - E.g., scheduler timer interrupt, or syscall trap instruction
  - Process is scheduled off CPU
  - Kernel gains control & receives various process context info, which it saves
    - E.g., register values (program counter, stack pointer, etc.)
- Upon completion of interrupt handling, kernel chooses a process to schedule, and discovers it has a pending signal

Steps in the execution of a signal handler

- To allow signal to be handled, the kernel:
  - Saves process context information onto user-space stack
    - Context == CPU registers (PC, SP), signal mask, and more
    - Saved context will be used later by *sigreturn()*...
    - See, e.g., *struct rt_sigframe* definition in `arch/x86/include/asm/sigframe.h`
    - Saved context information is visible via third argument of `SA_SIGINFO` handler, which is really *ucontext_t **; see also *ucontext_t* definition in `<sys/ucontext.h>`
  - Constructs frame on user-space stack for signal handler
    - Sets return address in frame to point to "signal trampoline"
  - Rearranges trap return address so that upon return to user space, control passes to signal handler
- Control returns to user space
  - Handler is called; handler returns to trampoline

Steps in the execution of a signal handler

- Trampoline code calls *sigreturn(2)*
  - **Now, the kernel once more has control!**
  - *sigreturn()* restores signal context
    - Signal mask, alternate signal stack
  - *sigreturn()* restores saved registers
    - Including program counter ⇒ next return to user space will resume execution where handler interrupted main program
  - Info needed by *sigreturn()* to do its work was saved earlier on user-space stack
    - For example, see code of, and calls to, *setup_sigcontext()* and *restore_sigcontext()* in kernel source file `arch/x86/kernel/signal.c`
  - Trampoline code is in user space (in C library or *vdso(7)*)
    - If in C library, address is made available to kernel via *sa_restorer* field (done by *sigaction()* wrapper function)

sigreturn()

- *sigreturn()*:
  - Special system call used only by signal trampoline
  - Uses saved context to restore state and resume program execution at point where it was interrupted by handler