Advanced Programming in the UNIX Environment

Week 07, Segment 5:
Reentrant and interrupted functions

Department of Computer Science
Stevens Institute of Technology

Jan Schaumann
jschauma@stevens.edu
https://stevens.netmeister.org/631/

Terminal — 80×24

```
        (void)printf("Now in sig_int, s=%d. Returning immediately.\n", ++s);
}

int
main(void) {
        printf("\n=> Establishing initial signal hander via signal(3).\n");
        if(signal(SIGQUIT, sig_quit) == SIG_ERR) {
                (void)fprintf(stderr, "can't set signal handler\n");
                exit(EXIT_FAILURE);
        }

        if(signal(SIGINT, sig_int) == SIG_ERR) {
                (void)fprintf(stderr, "can't set signal handler\n");
                exit(EXIT_FAILURE);
        }

        (void)sleep(SLEEP);

        (void)printf("\n=> Time for a second interruption.\n");
        (void)sleep(SLEEP);
        (void)printf("Now exiting.\n");
        exit(EXIT_SUCCESS);
}
apue$
```

Terminal — 80×24

```
apue$ vim signals4.c
apue$ cc -Wall -Werror -Wextra signals4.c
apue$ ./a.out
=> Waiting for a signal... done.
apue$ ./a.out 1
^\=> Waiting for a signal...SIGQUIT caught.
 done.
apue$ vim signals4.c
apue$ cc -Wall -Werror -Wextra signals4.c
apue$ ./a.out 1
^\SIGQUIT caught.
=> Waiting for a signal... done.
apue$
```

```
no 'root' found!

SIGALRM
SIGALRM
SIGALRM
SIGALRM
SIGALRM
SIGALRM
SIGALRM
SIGALRM
SIGALRM
return value corrupted: pw_name = root
[1]   Abort trap (core dumped) ./a.out

SIGALRM
SIGALRM
SIGALRM
[1]   Segmentation fault (core dumped) ./a.out

SIGALRM
SIGALRM
user jschauma not found!

^C
apue$
```

Terminal — 80×32

Signal-safe functions

Only functions that are guaranteed to be async-signal-safe can safely be used in signal handlers. These are functions that are either reentrant or non-interruptible. (These functions are also the only functions that may be used in a child process after doing fork(2) in a threaded program.)

The following functions are async-signal-safe. Any function not listed below is unsafe to use in signal handlers.

_Exit(2), _exit(2), abort(3), accept(2), access(2), alarm(3), bind(2), cfgetispeed(3), cfgetospeed(3), cfsetispeed(3), cfsetospeed(3), chdir(2), chmod(2), chown(2), clock_gettime(2), close(2), connect(2), creat(3), dup(2), dup2(2), execle(3), execve(2), fchmod(2), fchown(2), fcntl(2), fdatasync(2), fork(2), fpathconf(2), fstat(2), fsync(2), ftruncate(2), getegid(2), geteuid(2), getgid(2), getgroups(2), getpeername(2), getpgrp(2), getpid(2), getppid(2), getsockname(2), getsockopt(2), getuid(2), kill(2), link(2), listen(2), lseek(2), lstat(2), mkdir(2), mkfifo(2), open(2), pathconf(2), pause(3), pipe(2), poll(2), pthread_mutex_unlock(3), raise(3), read(2), readlink(2), recv(2), recvfrom(2), recvmsg(2), rename(2), rmdir(2), select(2), sem_post(3), send(2), sendmsg(2), sendto(2), setgid(2), setpgid(2), setsid(2), setsockopt(2), setuid(2), shutdown(2), sigaddset(3), sigdelset(3), sigemptyset(3), sigfillset(3), sigismember(3), sleep(3), signal(3), sigpause(3), sigpending(2), sigprocmask(2), sigset(3), sigsuspend(2), sockatmark(3), socket(2), socketpair(2), stat(2), symlink(2), sysconf(3), tcdrain(3), tcflow(3), tcflush(3), tcgetattr(3), tcgetpgrp(3), tcsendbreak(3), tcsetattr(3), tcsetpgrp(3), time(3), timer_getoverrun(2), timer_gettime(2), timer_settime(2), times(3), umask(2), uname(3), unlink(2), utime(3), wait(2), waitpid(2), write(2).

--More--(88%)

Blocking functions

Some system calls can block for long periods of time (or forever). These include things like:

- `read(2)` from files that can block (pipes, networks, terminals)
- `write(2)` to the same sort of files
- `open(2)` of a device that waits until a condition occurs (for example, a modem)
- `pause(3)`, which purposefully puts a process to sleep until a signal occurs
- certain `ioctl(2)`s
- certain IPC functions
- file- or record-locking mechanisms

Blocking functions

If a signal handler is invoked while a system call or library function call is blocked, then:

- the call may be forced to terminate with the error `EINTR`
- the call may return with a data transfer shorter than requested
- the call is automatically restarted after the signal handler returns

Which of these behaviors occurs depends on the interface and whether or not the signal handler was established using the `SA_RESTART` flag (see `sigaction(2)`).

Note: much of this is OS specific, and on some platforms only some calls may be restarted, while others always will fail when interrupted.

Terminal — 80×24

```
                }
        } else if (restarted) {
                (void)printf("\nread call was restarted\n");
        }

        printf("|%c|\n", c);
        return EXIT_SUCCESS;
}
apue$ cc -Wall -Werror -Wextra eintr.c
apue$ ./a.out
a
|a|
apue$
apue$ ./a.out
a^C
read call was interrupted
||
apue$ ./a.out
a^\a

read call was restarted
|a|
apue$
apue$
```

Reentrant and interrupted functions

- Only functions that are guaranteed to be async-signal-safe can safely be used in signal handlers.
- POSIX guarantees a set of such async-signal-safe functions; different OS (versions) may add others.
- Even async-signal-safe functions may set `errno`, so best to save and reset it.
- Interrupted system calls may fail, return short, or be restarted, again very much subject to variance across OS flavors and versions.