CH 10: Signals Notes
Advanced Programming In The Unix Environment 3e

Tyler Ryan

February 6, 2023

Contents

1 Chapter 10 Summary

Signals are used in most nontrivial applications. An understanding of the hows and whys of signal handling is essential to advanced UNIX programming.

This chapter is a long and thorough look at the UNIX System signals. You will start by looking at the warts in previous implementations of signals and how manifest themselves.

You will then proceed to the POSIX.1 reliable-signal concept and all the related functions. Once you've covered all these details, you will able to provide implementations of the POSIX.1 **abort()**, **system()**, and **sleep()** functions.

The chapter will finish with a look at the job control signals and the ways that you can convert between signal names and signal numbers.

2 Practical Key Takeaways (IMO)

- Be able to manage a group of signals using **signal sets** and the following functions.
- Understand **Signal Blocking and Unblocking**:
  - **sigemptyset()**: Clears out a signal set.
  - **sigaddset()**: Adds a signal to a signal set.
  - **sigdelset()**: Deletes a signal from a signal set.
  - **sigprocmask()**: Examines and changes a signal mask (Blocked Signals)
  - **sigismember()**: **TODO!**
  - **sigpending()**: Holds all currently blocked *and* pending signals.
  - **sigaction()**: Better version of **signal()**. **USE INSTEAD OF SIGNAL()**
  - **sigsetjmp()**: Use instead of **setjmp** when dealing with signals.

    * Saves stack environment **and signal mask**.
    * **setjmp()** saves stack environment **but not the signal mask**.
  - **siglongjmp()**: Use instead of **longjmp()** when dealing with signals.
  - **sigsuspend()**: Unblocks signals, suspend process, and wait for signal to occur.

There are other functions in this chapter, but they are rather obvious.

3 38 Unix Signals

Name	Description	Default action
SIGABRT	abnormal termination (abort)	terminate+core
SIGALRM	timer expired (alarm)	terminate
SIGBUS	hardware fault	terminate+core
SIGCANCEL	threads library internal use	ignore
SIGCHLD	change in status of child	ignore
SIGCONT	continue stopped process	continue/ignore
SIGEMT	hardware fault	terminate+core
SIGFPE	arithmetic exception	terminate+core
SIGFREEZE	checkpoint freeze	ignore
SIGHUP	hangup	terminate
SIGILL	illegal instruction	terminate+core
SIGINFO	status request from keyboard	ignore
SIGINT	terminal interrupt character	terminate
SIGIO	asynchronous I/O	terminate/ignore
SIGIOT	hardware fault	terminate+core
SIGJVM1	Java virtual machine internal use	ignore
SIGJVM2	Java virtual machine internal use	ignore
SIGKILL	termination	terminate
SIGLOST	resource lost	terminate
SIGLWP	threads library internal use	ignore
SIGPIPE	write to pipe with no readers	terminate
SIGPOLL	pollable event (poll)	terminate
SIGPROF	profiling time alarm (setitimer)	terminate
SIGPWR	power fail/restart	terminate/ignore
SIGQUIT	terminal quit character	terminate+core
SIGSEGV	invalid memory reference	terminate+core
SIGSTKFLT	coprocessor stack fault	terminate
SIGSTOP	stop	stop process
SIGSYS	invalid system call	terminate+core
SIGTERM	termination	terminate
SIGTHAW	checkpoint thaw	ignore
SIGTHR	threads library internal use	terminate
SIGTRAP	hardware fault	terminate+core
SIGTSTP	terminal stop character	stop process
SIGTTIN	background read from control tty	stop process
SIGTTOU	background write to control tty	stop process
SIGURG	urgent condition (sockets)	ignore
SIGUSR1	user-defined signal	terminate
SIGUSR2	user-defined signal	terminate
SIGVTALRM	virtual time alarm (setitimer)	terminate
SIGWAITING	threads library internal use	ignore
SIGWINCH	terminal window size change	ignore
SIGXCPU	CPU limit exceeded (setrlimit)	terminate+core/ignore
SIGXFSZ	file size limit exceeded(settrlimit)	terminate+core/ignore
SIGXRES	resource control exeeded	ignore

Table 1: All 38 Signals With Default Actions

4 Job Control Signals

Of the signals shown in Table 1, POSIX.1 considers 6 to be job control signals:

Table 2: 6 Job Control Signals

Name	Description	Default action
SIGCHLD	change in status of child	ignore
SIGCONT	continue stopped process	continue/ignore
SIGSTOP	stop	stop process
SIGTTIN	background read from control tty	stop process
SIGTTOU	background write to control tty	stop process

- Except for SIGCHLD, most applicatoin programs don't handle these signals.
  - Iteractive shells usually do all the work requred to handle these signals.
- When any of the 4 stop signals (SIGTSTP, SIGSTOP, SIGTTIN, SIGTOU) is genereated for a process, any pending SIGCONT signal for that process is discarded.
  - Similarly, when the SIGCONT signal is generated for a process, any pending stop signals for that same process are discarded.

5 sigaction() Function

5.0.1 Function Prototype

Listing 1: Sigaction Function Prototype

```
#include <signal.h>

// sigaction() uses the sigaction struct. See next section.
int sigaction(

// The signal number whose action is to be examined or modified.
int signo,

// If *act is non-null, we are modifying the action.
const struct sigaction * restrict act,

// If *oact is non-null, the system returns the previous action for the
// signal through the oact pointer.
struct sigaction * restrict oact
);
                        Returns: 0 if OK, -1 on error
```

5.0.2 sigaction Struct

Listing 2: Sigaction Struct

```
#include <signal.h>

struct sigaction
{
    // Signal Handler Function
    // or SIG_IGN or SIG_DFL
    void (* sa_handler)(int);

    // Signal Set
    sigset_t sa_mask;

    // Signal Options
    int sa_flags;

    // Alternate signal handler used when the SA_SIGINFO flag is used
    void (* sa_sigaction)(int, siginfo_t *, void *);
}
```

5.0.3 siginfo_t Struct

Listing 3: siginfo_t Struct

```
#include <signal.h>

/* Contains info about why the signal was generated.
 * Used when your signal handler has the following signature.
 *
 * void handler(int signo, siginfo_t * info, void *context);
 */
struct siginfo
{
    int      si_signo;       // signal number
    int      si_errno;       // if nonzero, errno value from <errno.h>
    int      pid_t si_pid;   // additional info (depends on signal)
    uid_t    si_uid;         // sending PID
    vaoid *  si_addr;        // sending process UID
    int      si_status;      // address that caused the fault.
    long     si_band;        // band number for SIGPOLL
    // Possible other values
}
```

5.0.4 Signal Action Options For The sigaction Struct

Option	Description
SA_INTERRRUPT	System calls interrupted by this signal are not automatically restarted The XSI default for **sigaction**.
SA_NOCLDSTOP	If **signo** is SIGCHLD, do not genereate this signal when a child process stops (job control). This signal is still generated, when a child terminates. As an XSI extension, SIGCHLD won't be sent when a stopped child continues if this flag is set.
SA_NOCLDWAIT	If **signo** is SIGCHLD, this option prevents the system from creating zombie processes when the children of the calling process terminate. If it subsequently calls **wait()**, the calling process blocks until all its child processes have terminated and then returns -1 with **errno** set to ECHILD.
SA_NODEFER	When this signal is caught, the signal is not automatically blocked by the system while the signal-catching function executes (unless the signal is also included in **sa_mask**). **Note that this type of operation corresponds to the earlier unreliable signals.**
SA_ONSTACK	If an alternate stack has been declared with **sigaltstack(2)**, this signal is delivered to the process on the alternate stack.
SA_RESETHAND	The disposition for this signal is reset to SIG_DFL, and the SA_SIGINFO flag is cleared on entry to the signal-catching function. **Note that this type of operation corresponds to the earlier unreliable signals.** The disposition for the 2 signals SIGILL and SIGTRAP can't be reset automatically. Setting this flag causes **sigaction** to behave as if SA_NODEFER is also set.
SA_RESTART	System calls interrupted by this signal are automatically restarted.
SA_SIGINFO	This option provides additional information to a signal handler: a pointer to a **siginfo** structure and a pointer to an identifier for the process context.

5.0.5 si_code Values Values For The siginfo_t Struct

Signal	Code	Reason
SIGILL	ILL_ILLOPC	illegal opcode
	ILL_ILLOPN	illegal operand
	ILL_ILLADR	illegal addressing mode
	ILL_ILLTRP	illegal trap
	ILL_PRVOPC	privileged opcode
	ILL_PRVREG	privileged register
	ILL_COPROC	coprocessor error
	ILL_BADSTK	internal stack error
SIGFPE	FPE_INTDIV	integer divide by zero
	FPE_INTOVF	integer overflow
	FPE_FLTDIV	floating-point divide by zero
	FPE_FLTOVF	floating-point overflow
	FPE_FLTUND	floating-point underflow
	FPE_FLTRES	floating-point inexact result
	FPE_FLTINV	invalid floating-point operation
	FPE_FLTSUB	subscript out of range
SIGSEGV	SEGV_MAPERR	address not mapped by object
	SEGV_ACCERR	invalid permissions for mapped object
SIGBUS	BUS_ADRALN	invalid address alignment
	BUS_ADRERR	nonexistent pysical address
	BUS_OBJERR	object-specific hardware error
SIGTRAP	TRAP_BRKPT	process breakpoint trap
	TRAP_TRACE	process trace trap
SIGCHLD	CLD_EXITED	child has exited
	CLD_KILLED	child has terminated abnormally (no core)
	CLD_DUMPED	child has terminated abnormally (with core)
	CLD_TRAPPED	traced child has trapped
	CLD_STOPPED	child has stopped
	CLD_CONTINUED	stopped child has continued
SIGPOLL	POLL_IN	data an be read
	POLL_OUT	data can be written
	POLL_MSG	input message available
	POLL_ERR	I/O error
	POLL_PRI	high-priority message available
	POLL_HUP	device disconected
Any	SI_USER	signal sent by **kill()**
	SI_QUEUE	signal sent by **sigqueue** (real-time extension)
	SI_TIMER	expiration of timer set by **timer_settime** (real-time extension)
	SI_ASYNCIO	completion of asynchronous I/O request (real-time extension)
	SI_MESGQ	arrival of a message on a message queue (real-time extension)

5.1 About sigaction()

- Allows us to examine or modify (or both) the action associated with a particular signal.
- It supersedes the **signal()** function.
- Unlike signal, if and when the signal-catching function returns, **the signal mask of the process is reset to its previous value.**
  - This way, we are able to block certain signals whenever a signal handler is invoked.
- The operating system includes the signal being delivered in the signal mask when the handler is invoked.
  - Hence, we are guaranteed that whenever we are processing a given signal, another occurrence of that same signal is blocked until we're finished procesing the first occurrence.

  - If the signal occurs five times while it is blocked, when we unblock the signal, the signal-handling function for that signal will usually be invoked only one time.

- Additional occurrences of the same signal are usually not queued.
- **siginfo_t:**
  - If the signal is SIGCHLD, then the **si_pid**, **si_status**, and **si_uid** field will be set.
  - If the signal is SIGILL or SIGSEGV, then the **si_addr** field contains the address responsible for the fault, although this address might not be accurate.
  - If the signal is SIGPOLL, then the **si_band** field wil contain the prority band for STREAMS messages that generate the POLL_IN, POLL_OUT, or POLL_MSG events.
  - The **si_errno** field contains the rror number orresponding to the condition that caused the signal to be generated, although its use is implementation defined.

5.2 Recreate signal() with sigaction()

Listing 4: An implemtation of signal using sigaction

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void my_handler(int signo) { printf("Signal Handled!\n"); }

__sighandler_t new_signal(int signo, __sighandler_t func);

int main()
{
    if (new_signal(SIGQUIT, my_handler) == SIG_ERR)
    {
        printf("Error: Catching signal\n");
        exit(EXIT_FAILURE);
    }

    sleep(5);

    exit(EXIT_SUCCESS);
}

__sighandler_t new_signal(int signo, __sighandler_t func)
{
     struct sigaction    act, oact;
     act.sa_handler =    func;

     sigemptyset(&act.sa_mask);
     act.sa_flags = 0;

     if (signo == SIGALRM)
     {
         /* Don't want SIGALRM restarted to allow us to set a timeout
          * for I/O operations
          */
#ifdef SA_INTERRUPT
         act.sa_flags |= SA_INTERRUPT;
#endif
```

```
    }
    // Setting SA_RESTART flag for all signals other than SIGALRM
    else
    {
#ifdef SA_RESTART
        act.sa_flags |= SA_RESTART;
#endif
    }

    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);

    return(oact.sa_handler);
}
```

6 sigsetjmp() and siglongjmp() Functions

6.1 Function Prototypes

Listing 5: sigsetjmp() and siglongjmp() Function Prototypes

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);
Returns: 0 if called directly, nonzero if returning from a call to siglongjmp()

void siglongjmp(sigjmp_buf env, int val);
```

6.2 Overview

- **setjmp()** and **longjmp()** functions can be used for **nonloal branching**.
  - **longjmp()** is often called from a signal handler to return to the main loop of a program, instead of return from the handler.
  - The problem wth **longjmp()** is that when a signal is caught, the signal-catching function is entered with the current signal automatically being added to the sigmal mask of the process.
  - This prevents subsequent occurences of that signal from interrupting the signal handler.
- POSIX.1 does not specify the effect of **setjmp()** and **longjmp()** on signal masks.
- Instead, 2 new functions **sigsetjmp()** and **longjmp** were defined.
  - These 2 functions should **always** be used when branching from a signal handler.
- The only difference between these functions and the **setjmp()** and **longjmp()** functions is that **sigsetjmp()** has an additional argument (**savemask**)
- If **savemask** is nonzero, then **sigsetjmp()** also saves the current sgnal mask of the process in **env**.
- When **siglongjmp()** is called, if the **env** argument was saved by a call to **sigsetjmp()** with a nonzero **savemask**, then **siglongjmp** restores the saved signal mask.
- **sig_atomic_t:** Defined by the ISO C standard to be the type of variable that can be written without being interrupted.
  - This type should not extend across page boundries on a ssystem with virtual memory.
  - Can be accessed with a single machin instruction.

  - Alway include the ISO type qualifier **volatile** for these data types since the variable is being accessed by 2 different threads: **main()** and the executing signal handler.

7 sigsuspend() Function

7.1 Function Prototype

Listing 6: sigsuspend() Function Prototype

```
#include <signal.h>

int sigsuspend(const sigset_t * sigmask);
            Returns: −1 with errno set to EINTR
```

7.2 Overview

- Used to unblock a signal and then pause, waiting for the previously blocked signal to occur.
- Allows us to both reset the signal mask and put the process to sleep in a single **atomic operation**.
- **sigsuspend()**:
  - The signal mask of the process is set to the value pointed to by **sigmask**.
  - Then the process is suspended until a signal is caught or until a signal occurs that terminates the process.
  - If the signal is caught and if the signal handler returns, then **sigsuspend()** returns and the signal mask of the process is set to its value before the call to **sigsuspend()**.
  - **Note: There is no successfull return from this function. If it returns to the caller, it always returns -1 with *errno* set to EINTR (indicating an interrupted system call).**
- Uses:
  - Protect a critical region of code from a specific signal.
  - Wait for a signal handler to set a global variable.
  - Synchronizing a parent and a child.

7.3 Examples

7.3.1 Protect A Critical Region of Code From A Signal

Listing 7: Protect Critical Regions From Specific Signals

```
[Remember to put include statements here]
/* pr_mask() is annoyingly not defined here.
 * You will have to write your own function for this.
 */
static void sig_int(int);

int main(void)
{
    sigset_t    newmask, oldmask, waitmask;

    pr_mask("program start: ");

    if (signal(SIGINT, sig_int) == SIG_ERR)
    {
        printf("error: signal(SIGINT)\n");
```

```
            exit(EXIT_FAILURE);
        }

        sigemptyset(&waitmask);
        sigaddset(&waitmask, SIGUSR1);
        sigemptyset(&newmask);
        sigaddset(&newmask, SIGINT);

        // Block SIGINT and save current signal mask
        if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        {
            printf("error: SIG_BLOCK\n");
            exit(EXIT_FAILURE);
        }

        // Critical region of code.
        pr_mask("in critical region: ");

        // Pause, allowing all signals except SIGUSR1
        if (sigsuspend(&waitmask) != -1)
        {
            printf("error: sigsuspend\n");
            exit(EXIT_FAILURE);
        }

        pr_mask("after return from sigsuspend: ");

        // Reset signal mask which unblocks SIGINT

        if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        {
            printf("error: SIG_SETMASK\n");
            exit(EXIT_FAILURE);
        }

        // And continue processing
        pr_mask("program exit: ");

        exit(EXIT_SUCCESS);
    }

static void sig_int(int signo)
{
    pr_mask("\nin sig_int: ");
}
```

Output:

```
$ ./a.out
program start:
in critical region: SIGINT
^?
in sig_int: SIGINT SIGUSR1
after return from sigsuspend: SIGINT
program exit:
```

7.3.2 Using Signals To Sync Parent and Child Processes

Listing 8: Routines to allow a parent and child to synchronize

```
[Remember to include the headers]

// Set nonzero by sig handler
static volatile sig_atomic_t sigflag;
static sigset_t newmask, oldmask, zeromask;

static void sig_usr(int signo)
{
    sigflag = 1;
}

void TELL_WAIT(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
    {
        printf("error: signal(SIGUSR1)\n");
        exit(EXIT_FAILURE);
    }
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
    {
        printf("error: signal(SIGUSR2)\n");
        exit(EXIT_FAILURE);
    }

    sigemptyset(&zeromask);
    sigemptyset(&newmask);

    sigaddset(&newmask, SIGUSR1);
    sigaddset(&newmask, SIGUSR2);

    // Block SIGUSR1 and SIGUSR2, and save current signal mask
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
    {
        printf("error: SIG_BLOCK\n");
        exit(EXIT_FAILURE);
    }
}
void TELL_PARENT(pid_t pid)
{
    // Tell parent we are done
    kill(pid, SIGUSR2);
}

void WAIT_PARENT(void)
{
    while (sigflag == 0)
    {
        // wait for parent
        sigsuspend(&zeromask);
    }

    // Reset signal mask to original value
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
    {
            printf("error: SIG_SETMASK\n");
            exit(EXIT_FAILURE);
    }
}

void TELL_CHILD(pid_t pid)
{
    // Tell child we are done
    kill(pid, SIGUSR1);
}

void WAIT_CHILD(void)
{
    while (sigflag == 0)
    {
        // wait for child
        sigsuspend(&zeromask);
    }

    sigflag = 0;

    // Reset signal mask to original value
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
    {
            printf("error: SIG_SETMASK\n");
            exit(EXIT_FAILURE);
    }
}
```

8 abort() Function

8.1 Function Prototype

Listing 9: abort() Function Prototype

```
#include <stdlib.h>

void abort(void);
        This function NEVER RETURNS
```

8.2 Overview

- Sends the SIGABRT signal to the caller.
- Processes should not ignore this signal.
- ISO C specifications:
  - states that calling **abort()** will deliver an unsuccessful termination notification to the host environment by calling **raise(SIGABTR)**.
  - requires that if the signal is caught and the signal handler returns, **abort()** still doesn't return to its caller.
  - If this signal is caught, the only way the signal handler can't return is if it calls **exit()**, **_exit**, **_Exit**, **longjmp()**, or **siglongjmp()**.
  - It also specifies that **abort()** overrides the blocking or ignoring of the signal by the process.

  - leaves it up to the implementation as to whether ouput streams re flushed and whether temporary files are deleted.
  - Requires tat if the call to **abort()** terminates the process, then the effect on the open standard I/O streams in the process wil be the same as if the process had called **fclose()** on each stream before terminating.

- The intent of letting the process catch the SIGABRT signal is to allow it to perform any cleanup that itwants to d obefore the process terminates.
  - If the proess doesn't terminate itself fro mthis signal handler, POSIX.1 states that, when the signal handler returns, **abort()** terminates the process.

9 system() Function Revisited

9.1 Nothing Here

10 sleep() Function

10.1 Function Prototype

Listing 10: sleep() Function Prototype

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);

    Returns: 0 or number of unslept seconds.
```

11 Additional Features

This section describes additional implementation-dependent features of signals.

11.1 Signal Names

11.1.1 Function Prototypes

Listing 11: Implementation-Dependent Signal Names

```
#include <signal.h>


extern char * sys_siglist[];
// msg is normally the same name as the program.
// This signal is similar to perror()
void psignal(int signo, const char * msg);

#include <string.h>

char * strsignal(int signo);
            Returns: a pointer to a string describing the signal.
```

11.2 Signal Mappings

11.2.1 Function Prototypes

Listing 12: Solaris Signal Mappings

```
#include <signal.h>

int sig2str(int signo, char *str);
int str2sig(const char * str, int * signop);

    Both return: 0 if OK, −1 on error
```

12 Book Exercises

12.1 Questions

12.1.1 Q4

What is wrong with this code?

```
signal(SIGALRM, sig_alrm);
alarm(60)
if (setjmp(env_alrm) != 0)
{
    // Handle timout
    ...
}
...
```

Solution Here: 12.2.2

12.1.2 Q6

Write a program to test the parent-child synchronization functions in Listing [].

The process creates a file and writes the integer 0 to the file. The process then calls **fork()**, and the parent and child alternate incrementing the counter in the file.

Each time the counter is incremented, print which process (parent or child) is doing the increment.

12.2 Solutions

Solutions not obtained from the back of the book were obtained from:
https://github.com/adalton/apue3/tree/master

Note: They may or may not be correct!

12.2.1 A1

The program terminates the first time we send it a signal. The reason is that the **pause()** function returns whenever a signal is caught.

12.2.2 A4

We again have a race condition; this time between the first call to **alarm()** and the call to **setjmp()**.

If the process is blocked by the kernel between these two function calls, the alarm goes off, the signal handler is called, and **longjmp()** is called.

But since **setjmp()** was never called, the buffer **env_alrm** is not set.

The operation of **longjmp()** is undefined if its jump buffer has not been initialized by **setjmp()**.

12.2.3 A6

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

static volatile sig_atomic_t sigflag; /* set nonzero by sig handler */
static sigset_t newmask, oldmask, zeromask;

static void
sig_usr(int signo)  /* one signal handler for SIGUSR1 and SIGUSR2 */
{
    sigflag = 1;
}

static void
TELL_WAIT(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        perror("signal(SIGUSR1) error");

    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        perror("signal(SIGUSR2) error");

    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGUSR1);
    sigaddset(&newmask, SIGUSR2);

    /* Block SIGUSR1 and SIGUSR2, and save current signal mask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        perror("SIG_BLOCK error");
}

static void
TELL_PARENT(void)
{
    // Tell parent we're done
    kill(getppid(), SIGUSR2);
}

static void
WAIT_PARENT(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask);  /* and wait for parent */
    sigflag = 0;
    /* Reset signal mask to original value */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
```

```
        perror("SIG_SETMASK error");
}

static void
TELL_CHILD(pid_t pid)
{
    kill(pid, SIGUSR1); // tell child we are done
}

static void
WAIT_CHILD(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask);  /* and wait for child */
    sigflag = 0;
    /* Reset signal mask to original value */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        perror("SIG_SETMASK error");
}

static int
increment_value(FILE* const file)
{
    int value = 0;

    fseek(file, 0, SEEK_SET);
    fread(&value, sizeof(value), 1, file);

    ++value;

    fseek(file, 0, SEEK_SET);
    fwrite(&value, sizeof(value), 1, file);
    fflush(file);

    return value;
}

int
main(void)
{
    const int NUM_ITERATIONS = 100;

    FILE* const file = fopen("/tmp/data", "w+");
    if (file == NULL) {
        perror("fopen");
        return 1;
    }

    TELL_WAIT();
    int i;
    const pid_t pid = fork();

    if (pid < 0) {
        perror("fork");
        return 1;
    }
```

```
        if (pid == 0) {
            for (i = 0; i < NUM_ITERATIONS; ++i) {
                printf(" child incrementing, value: %3d\n",
                       increment_value(file));

                TELL_PARENT();
                WAIT_PARENT();
            }
        } else {
            for (i = 0; i < NUM_ITERATIONS; ++i) {
                WAIT_CHILD();

                printf("parent incrementing, value: %3d\n",
                       increment_value(file));

                TELL_CHILD(pid);
            }
        }

        fclose(file);

        return 0;
    }
```

Partial output:

```
$ ./a.out
 child incrementing, value:   1
parent incrementing, value:   2
 child incrementing, value:   3
parent incrementing, value:   4
 child incrementing, value:   5
parent incrementing, value:   6
...
 child incrementing, value: 195
parent incrementing, value: 196
 child incrementing, value: 197
parent incrementing, value: 198
 child incrementing, value: 199
parent incrementing, value: 200
```

12.2.4 A12

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static void
sig_handler(const int signo)
{
#define MESSAGE_TEXT "signal received\n"
    write(STDERR_FILENO, MESSAGE_TEXT, sizeof(MESSAGE_TEXT) - 1);
#undef MESSAGE_TEXT
}

#define BUFFER_SIZE (1024 * 1024 * 1024)

int
main(void)
{
    // Allocate a 1GB buffer
    char* const buffer = calloc(sizeof(char), BUFFER_SIZE);
    if (buffer == NULL) {
        perror("calloc");
        return 1;
    }

    FILE* const file = fopen("/tmp/ex12.out", "w");
    if (file == NULL) {
        perror("fopen");
        free(buffer);
        return 1;
    }

    signal(SIGALRM, sig_handler);
    alarm(1);

    const size_t written = fwrite(buffer, BUFFER_SIZE, 1, file);
    if (written != 1) {
        fprintf(stderr, "Failed to write buffer\n");
    }

    fclose(file);
    free(buffer);

    return 0;
}
```

13 Personal Exercises

These are just some problems I came up with while trying to figure this stuff out.

13.1 Problems

13.1.1 Q1

Recreate a basic signal() function using sigaction and implement it with a signal and handler of your choosing.

- Be aware of the **__sigset_t** type, but don't use it in this problem.

13.1.2 Q2

Write a program that does the following:

- Recreates the signal() function with sigaction.
- Empties a signal set.
- Adds 2 signals to that signal set.
- Blocks these signals using a signal mask.
  - sleep() for X seconds to allow the user to test this.
  - Use **kill -USR1 [pid]** in another terminal to test this.
- Unblock 1 signal.
  - Allow the user to test this for ONLY 10 seconds.
  - Do not use a loop or the sleep() function.
  - You **must** use a signal (hint SIGALRM).

13.1.3 Q3

Write a program that allows a user to send signals to a parent process from a child process via the terminal.

The main process should fork a new terminal where the user can type **kill -USR1 [ppid]** to send the desired signals to the parent.

13.2 Solutions

13.2.1 A1

Recreate a basic signal() function using sigaction and implement it with a signal and handler of your choosing.

- Be aware of the **__sigset_t** type, but don't use it in this problem.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

// The second parameter just takes in a function.
static void * my_signal(int, void (*)(int));
static void usr1_handler(int);

int
main()
{
    if (my_signal(SIGUSR1, usr1_handler) == SIG_ERR)
    {
        printf("error: catching signal\n");
        exit(EXIT_FAILURE);
    }

    // run "kill -USR1 <pid>" in another terminal
    printf("PID: %d\n", getpid());

    while (1) { }

    exit(EXIT_SUCCESS);
}

static void
usr1_handler(int signo)
{
    printf("Handled SIGUSR1\n");
}

static void *
my_signal(int signo, void (* func)(int))
{
    struct sigaction sa, osa;
    sa.sa_handler = func;
    sa.sa_flags   = 0;

    sigemptyset(&sa.sa_mask);

    sigaction(signo, &sa, &osa);

    return osa.sa_handler;
}
```

13.2.2 Q2

Write a program that does the following:

- Recreates the signal() function with sigaction.
- Empties a signal set.
- Adds 2 signals to that signal set.
- Blocks these signals using a signal mask.
  - sleep() for X seconds to allow the user to test this.
  - Use **kill -USR1 [pid]** in another terminal to test this.
- Unblock 1 signal.
  - Allow the user to test this for ONLY 10 seconds.
  - Do not use a loop or the sleep() function.
  - You **must** use a signal (hint SIGALRM).

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <signal.h>
    #include <stdbool.h>

    static void * my_signal(int, void (*)(int));
    void validate_mask(sigset_t *);

    static void usr1_handler(int signo)
    {
        printf("SIGUSR1 handled!\n");
    }
    static void usr2_handler(int signo)
    {
        printf("SIGUSR2 handled!\n");
    }

    static void alrm_handler(int signo)
    {
        printf("The program has run for 10 seconds.\n");
        exit(EXIT_SUCCESS);
}

int
main(void)
{
            // Add signals to block
        sigset_t ss, osa;
        sigemptyset(&ss);
        sigaddset(&ss, SIGUSR1);
        sigaddset(&ss, SIGUSR2);
        sigprocmask(SIG_BLOCK, &ss, &osa);

        validate_mask(&ss);

        // Watch for sigusr1
        if (my_signal(SIGUSR1, usr1_handler) == SIG_ERR)
        {
            printf("error: catching signal\n");
            exit(EXIT_FAILURE);
        }

            // Watch for sigusr2
        if (my_signal(SIGUSR2, usr2_handler) == SIG_ERR)
        {
            printf("error: catching signal\n");
            exit(EXIT_FAILURE);
        }

            // Watch for sigalrm
        if (my_signal(SIGALRM, alrm_handler) == SIG_ERR)
        {
            printf("error: catching signal\n");
            exit(EXIT_FAILURE);
        }

        printf("Enter the following command within 10 seconds\n");
        printf("kill -USR1 %d\n", getpid());
        sleep(10);

        // Remove SIGUSR2 from the block list
        sigdelset(&ss, SIGUSR2);
        // Set the new mask (without sigusr2)
        sigprocmask(SIG_SETMASK, &ss, NULL);

        alarm(10);
        printf("Unblocked SIGUSR1\n");
```

```
    printf("Alarm started. 10 seconds until program terminates\n");

    /* Didn't end up getting this function to work correctly.
     * At this point SIGUSR2 is no longer in ss, but
     * this function states that is is...
     */
    validate_mask(&ss);

    // Just to keep the program running and allow
    // user input.
    while (true)
    {
    }

    exit(EXIT_SUCCESS);
}


static void *
my_signal(int signo, void (*func)(int))
{
    struct sigaction sa, osa;
    sa.sa_handler = func;
    sa.sa_flags   = 0;

    sigaction(signo, &sa, &osa);

    return osa.sa_handler;
}

void
validate_mask(sigset_t * signal_set)
{
    // Check if signals are in mask (used to block signals)
    // This function doesn't really seem to be that reliable
    bool usr1 = sigismember(signal_set, SIGUSR1);
    bool usr2 = sigismember(signal_set, SIGUSR1);

    if (usr1 && usr2)
        printf("Both signals are in mask\n");
    else if (usr1)
        printf("Only SIGUSR1 is in mask\n");
    else if (usr2)
        printf("Only SIGUSR2 is in mask\n");
    return;
}
```

13.2.3 A3

Write a program that allows a user to send signals to a parent process from a child process via the terminal.

The main process should fork a new terminal where the user can type **kill -USR1** [**ppid**] to send the desired signals to the parent.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <wait.h>

/*
 * NOTE: IDK if this is the correct way to go about
 * this. Upon trying to test things out, there seems
 * to be a few issues.
 *
 * When the child sends a Signal to the parent, it
 * causes an error with wait().
 */

static void sig_handler(int);
static void * my_sigaction(int, void (*)(int));

int
main(void)
{
    pid_t pid;
    int status;

    // watch for signal.
    if (my_sigaction(SIGUSR1, sig_handler) == SIG_ERR)
    {
        printf("error: catching signal\n");
        exit(EXIT_FAILURE);
    }

    pid = fork();

    if (pid < 0)
    {
        printf("error: forking\n");
        exit(EXIT_FAILURE);
    }

    else if (pid == 0)
    {
            printf("Child PID: %d\n", getpid());
            status = execlp("alacritty", "terminal", NULL);
            if (status < 0)
            {
                printf("error: executing\n");
                exit(EXIT_FAILURE);
            }

            exit(EXIT_SUCCESS);
        }

        printf("Parent PID: %d\n", getpid());
        printf("Parent Doing something else\n");

        // This wait() function would fail due to an
        // Interrupted system call
        //
        // wait(&status);

        while (1) {}
    }

    static void
    sig_handler(int signo)
    {
        printf("SIGUSR1 was sent\n");
    }

    static void *
    my_sigaction(int signo, void (*func)(int))
    {
        struct sigaction sa, osa;
        sa.sa_handler = func;
        sa.sa_flags = 0;
        sigemptyset(&sa.sa_mask);

        sigaction(signo, &sa, &osa);

        return osa.sa_handler;
    }
```