Low-Level I/O & Signals

CSC209H5: Software Tools & Systems Programming

Robert (Rupert) Wu
rupert.wu@utoronto.ca

Department of Computer Science
University of Toronto

Feb 27, 2023

Lesson Plan

1. Bitwise algebra and manipulation
2. Deep dive into low-level file IO
3. Error handling
4. Signals
5. Break (~15min)
6. Midterm Test at 7pm

Acknowledgements

Part of the slides are borrowed from Andi Bergen.

Section 1

Bit Manipulation

Bitwise Operators: Basics

All data is composed of bits. Numerics and numerical macros/aliases can be manipulated with bitwise operators.

- Flip/negation (NOT): `~x`
- Union (inclusive-OR): `x|y`
- Intersection (AND): `x&y`
- Exlusive-OR (XOR): `x^y`

```
unsigned x = 0;
printf("x = %u\n", x);
printf("~x = %u, %u\n", ~x, UINT_MAX);

unsigned y = 7;
printf("7|9 = %u\n", y | 9);
printf("7&9 = %u\n", y & 9);
printf("7^9 = %u\n", y ^ 9);
```

Bitwise Operators: Shifts

All data is composed of bits. Numerics and numerical macros/aliases can be manipulated with bitwise operators.

- Bit-Shifts: `x<<y`, `x>>y`
  - Right bit-shift >> is logical on unsigned numbers.
  - Right bit-shift >> is arithmetic on signed numbers.
  - You can (cheaply) scale numbers to powers of 2 this way.

```
for (unsigned z = 1; z < UPPER; z++) {
    printf("Computing powers of 2 scaled by %u...\n", z);
    for (unsigned i = 0; i < UPPER; i++)
        if (i == 11) printf("%u*(2^%u) = %u\n", z, i, z << i);
}
int s = -12815;
printf("s>>2 = %d\n", s >> 2);          // s/4
printf("s>>2>>3 = %d\n", s >> 2 >> 3);  // s/4/8
unsigned u = 12815;
printf("u>>1 = %u\n", u >> 1);          // u/2
printf("u>>7>>3 = %u\n", u >> 7 >> 3);  // u/7/3
```

Section 2

Low-Level File I/O

File I/O: File Descriptors

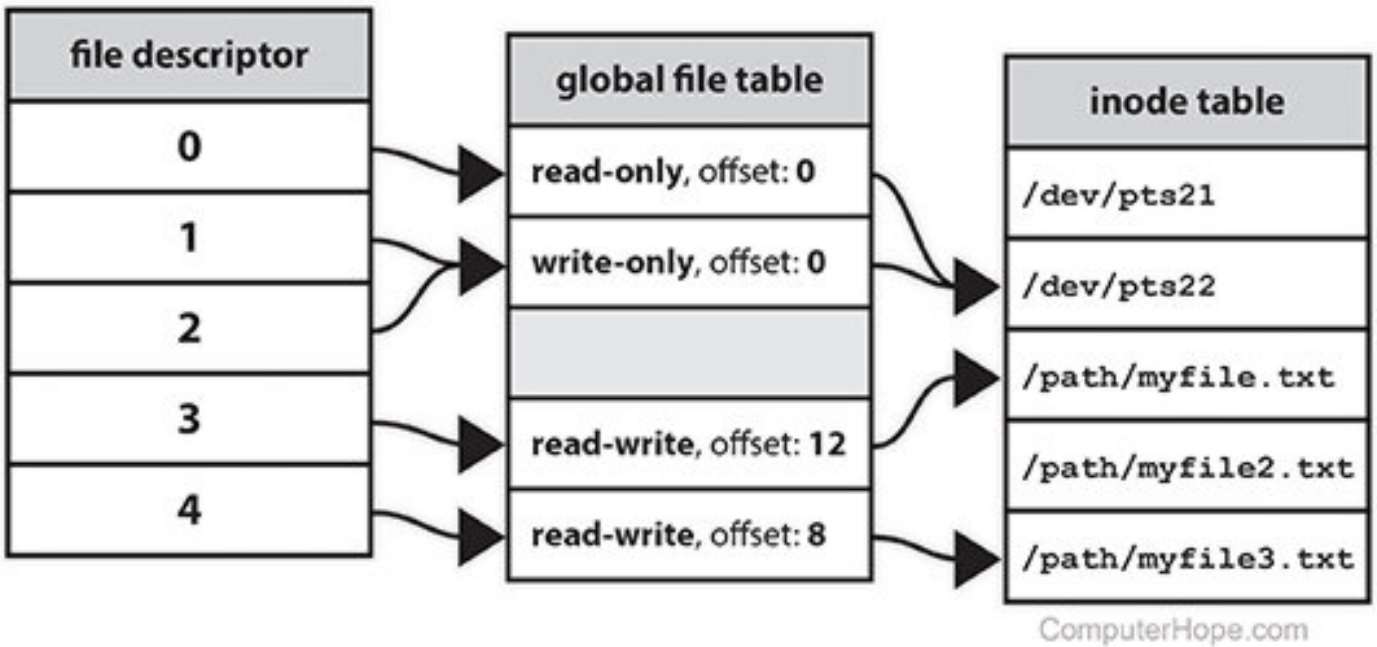

Figure 1: Example of FDs pointing to inodes.

A file descriptor (FD) is a number (non-negative integer) that uniquely identifies an open file in a computer's operating system. It describes a data resource, and how that resource may be accessed.

`www.computerhope.com/jargon/f/file-descriptor.htm`

File I/O: Streams, File Descriptors & Pointers

Streams are files to which data is written or from which data is read. They're accessed through file pointers (`FILE *` from `<stdio.h>`) that wrap around FDs. The following default streams (FDs) are provided by `<stdio.h>`.

- `stdin` (0): default input; typically from user keyboard or pipes.
- `stdout` (1): default output; usually to terminal screen or pipes.
- `stderr` (2): default error; also to terminal screen.
- Use > to *redirect* `stdout`, and 2> to redirect `stderr`
  - > overwrites the output file, >> appends to it.

File I/O: Opening & Closing

```
FILE *fopen(const char *filename, const char * mode);
```

A file `filename` is opened with `fopen()` in a `mode` `{r|w|a}{|+}` to perform the following operations. Returns a file pointer that wraps the FD. The pointer is `NULL` if we fail to open the file (often because the file doesn't exist or your process doesn't have permission).

action\mode	r	w	a	r+	w+	a+
read	yes	no	no	yes	yes	yes
write	no	yes	no	yes	yes	yes
append	no	no	yes	no	no	no
file exists	ok	ok	ok	ok	truncate	append
doesn't exist	fail	create	create	fail	ok	create

```
int fclose(FILE *stream);
```

- `stream`: a `FILE *` opened by `fopen()`/`freopen()`.
- returns: `0` if closed properly, `EOF` otherwise.

File I/O: Reading & Writing

Reading

1. `getchar()`: read a character from `stdin`.
2. `fgetc()`: read a single character from the file.
3. `fgets()`: read strings from files.
4. `fscanf()`: formatted input from a file.
5. `fread()`: block of raw bytes from files; useful for binary files.

Examples: `e1.c`, `e2.c`.

Writing

You can use `putchar()` to write a character to `stdout`.

```
size_t fwrite(const void *ptr, size_t size,
    size_t nmemb, FILE *stream)
```

Alternatively, use `fwrite()` to write `nmemb` elements (each `size` large) from `*ptr` to `stream`.

Low-Level I/O: A Deep Dive

The aforementioned library functions from `stdio.h` are high-level conveniences! But sometimes we need to work with lower-level functions. When we want to do low-level I/O, to bypass the buffering and abstractions provided by the C standard library, we must use *system calls*, namely:

1. `open()`
2. `close()`
3. `read()`
4. `write()`

Many different flags, errors, corner cases, etc. to consider.

- See `man 2 <sys-call>` for each.

Low-Level I/O: Be Careful!

Proper usage of low-level I/O often requires looping and handling error conditions.

```
ssize_t ret;
while (len != 0 && (ret = read(fd, buf, len)) != 0) {
    if (ret == -1) {
        if (errno == EINTR)
            continue;
        perror("read");
        break;
    }
    len -= ret;
    buf += ret;
}
```

Low-Level I/O: File Descriptors

As before, we use *file descriptors (FDs)*, serving as indices for open files.

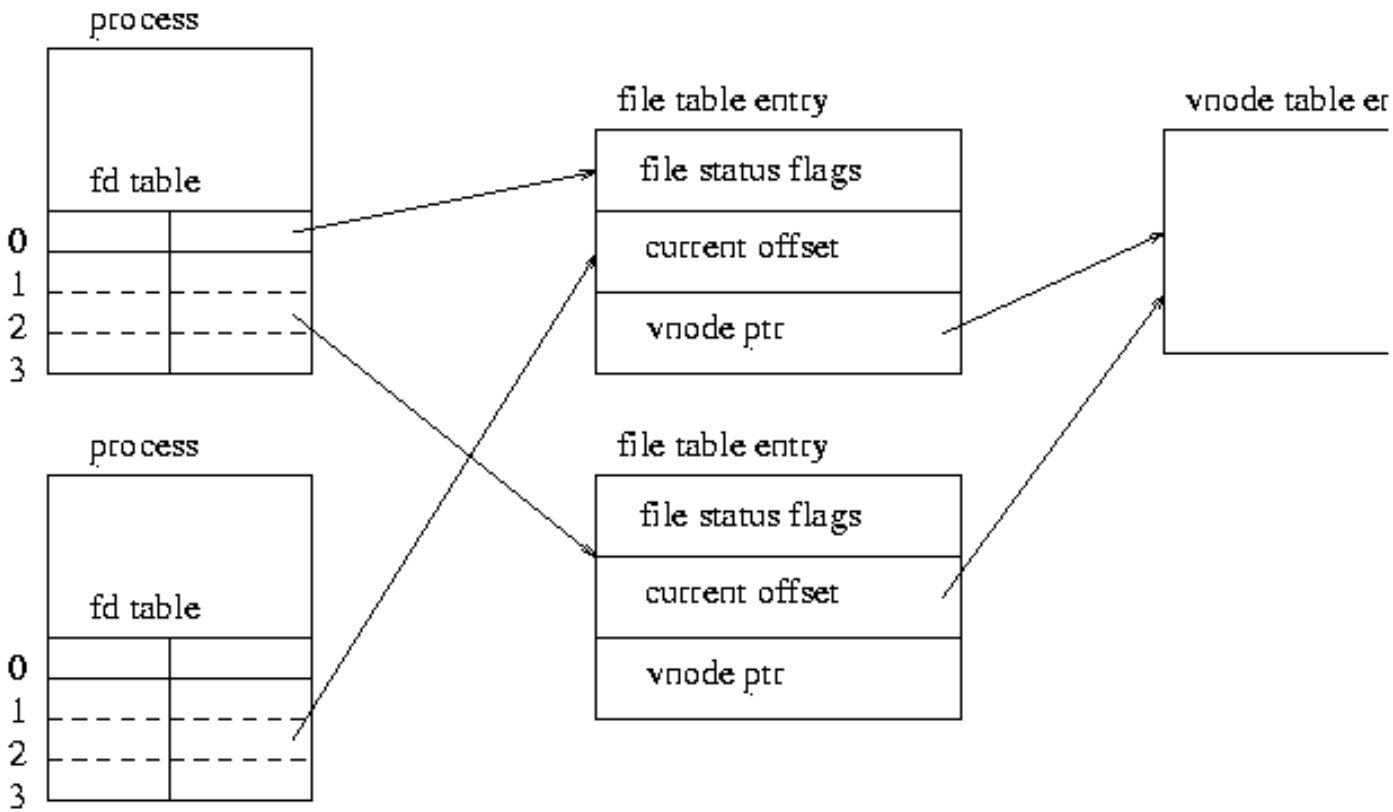

Figure 2: from `tinf2.vub.ac.be/~dvermeir/manuals/uintro/uintro.html`

Each process has its own *file descriptor (FD) table*. File descriptor `N` in *process A* can refer to a different file than file descriptor `N` in *process B*.

Low-Level I/O: File Descriptors vs. Descriptions

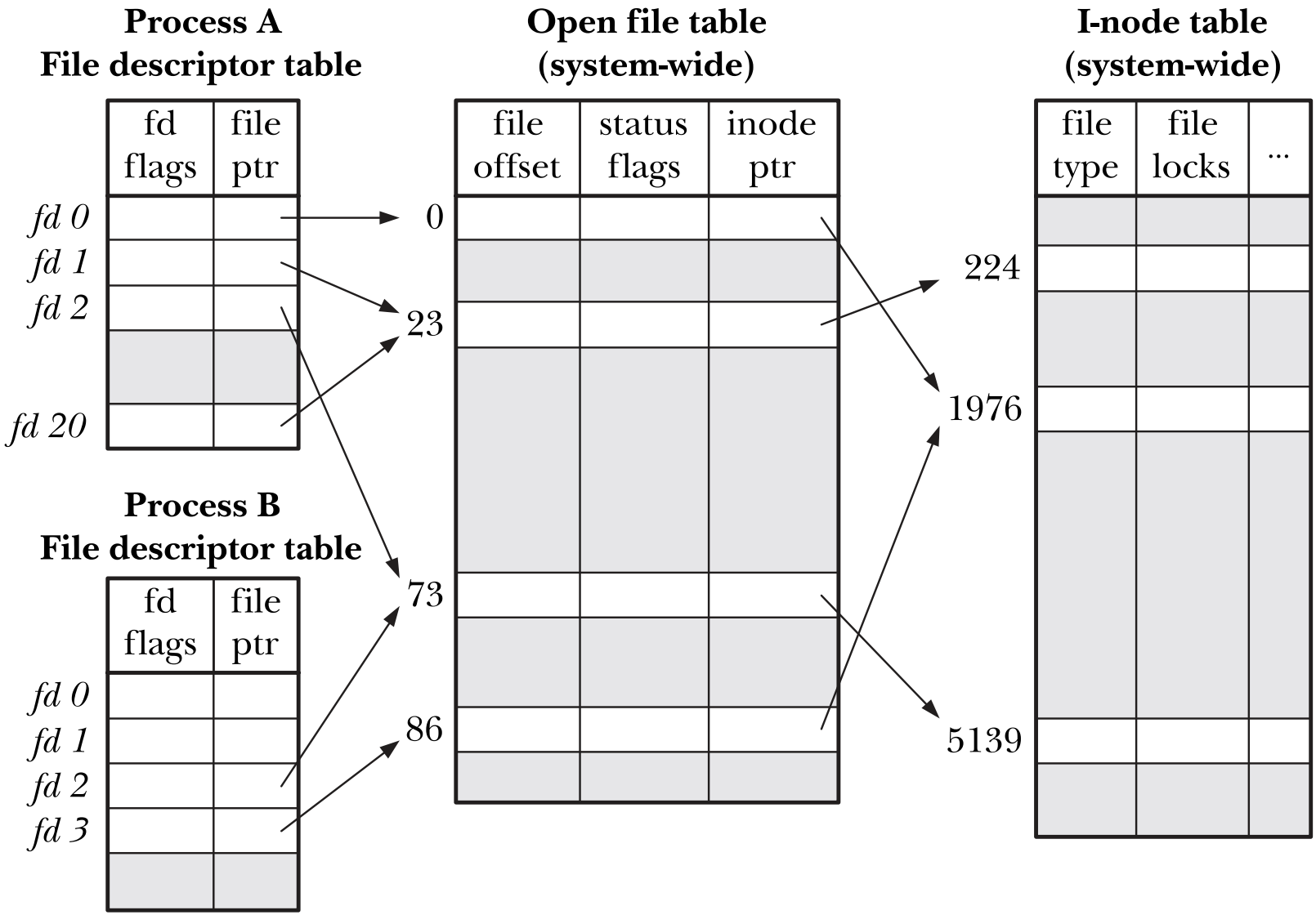

Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Figure 3: Mapping between file descriptors and file descriptions

Low-Level I/O: exec and File Descriptors

When a program calls `exec`, its process is replaced by a new program that still retains the FDs of the original process.

But upon replacing the process image with the new program, the `FILE *` variables are gone (memory leaks?).

So the new program must either:

1. Perform low-level I/O using the `read()` or `write()` system calls; or
2. Use `fdopen()` to associate a new buffered file stream with an existing open file descriptor.

```
FILE *fdopen(int fd, const char *mode);
```

More on `exec`/`execl` when we discuss processes in-depth.

Low-Level I/O: dup/dup2

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- dup returns a new FD that refers to the same file as `oldfd`.
- dup2 does the same, but lets you specify the value of new FD.
  - dup2 first closes `newfd` if already in use.

Output Redirection with dup2

```
int main (void) {
    int fd = open("lsout", O_WRONLY | O_CREAT, 0600);
    if (fd == -1) {
        perror("open");
        exit(1); }
    dup2(fd, STDOUT_FILENO);
    execl("/bin/ls", "ls", "-l", (char *)NULL);
    perror("execl");
    return 1; // why are we always returning 1?
}
```

Low-Level I/O: Arnold's Examples

dup: [www]/lectures/arnold/w07/lowLevelFileIO/lowLevelFileIO6.c

dup2: [www]/lectures/arnold/w07/lowLevelFileIO/lowLevelFileIO7.c

where [www] = mcs.utm.utoronto.ca/~209/23s

Section 3

Error Number (`errno`)

Error Number (errno)

The macro `errno` is implemented (system-dependently) as variable `int`.

```
#define errno /*implementation-defined*/
```

- At the start of a program, `errno` has the value `0`.
- Library functions can write strictly positive `int` to `errno` to indicate the last error that occurred.
  - `perror` is provided by `stdio.h` and used to print the associated description.

    ```
    FILE *f = fopen("non_existent", "r");
    if (f == NULL)
        perror("fopen() failed");
    else
        fclose(f);
    ```

    Possible output:

    ```
    fopen() failed: No such file or directory
    ```
  - `strerror` likewise produces the description string.

Error Number (errno)

The macro errno is implemented (system-dependently) as variable int.

```
#define errno /*implementation-defined*/
```

- errno doesn't force your program to fail, but you can use it as an excuse to.
- In addition to gdb, valgrind, it can be used as validation (more useful than assert.h) to aid debugging.

Issues pre-C11

Previously errno was supposedly a macro but also unspecified whether it was a macro or declared identifier. As of C11, errno is a macro.

More reading

See descriptions and demo in en.cppreference.com/w/c/error/errno.

Section 4

Signals

Signals: An Introduction

- Signals are unexpected, asynchronous events that can happen at any time.
- Unless you make special arrangements, most signals terminate your process.
- Signals are a basic form of **inter-process communication**.
- You have already been sending signals through the shell, e.g., via the `ctrl+c` and `ctrl+z` key combinations.
  - What does `ctrl+c` do?
    - The terminal sends the `SIGINT` signal to the process.
    - By default, `SIGINT` ("Interrupt from keyboard") terminates the process.
  - What about `ctrl+z`?
    - Similarly, `ctrl+z` triggers a `SIGTSTP` signal ("Stop typed at terminal")

The full list

On Linux, you can find a list in `/usr/include/bits/signum.h`.

Signals: Punishment for Misbehaviour

The kernel sends several other signals to terminate processes when a program misbehaves:

- `SIGSEGV` Invalid memory reference
- `SIGFPE` Floating-point exception
- `SIGILL` Illegal instruction

There are also the user-defined signals `SIGUSR1` and `SIGUSR2` that we can use for our own purposes. By default they terminate the process.

Signals: Sending from the Shell

To send a signal `SIGNAME` to one or more processes given their process ID(s) `pid`'s, use `kill`. Why is called `kill`?

```
$ kill -SIGNAME <pids>
```

For example:

```
$ kill -SIGINT 11248
$ kill -SIGKILL 11248
```

Don't let the name `kill` fool you! It's generally for sending signals, but often used to send `SIGKILL`, hence the name.

Unstoppable signals

`SIGKILL` and `SIGSTOP` cannot be caught, blocked, or ignored (see `man 7 signal`).

Sending signals between processes

A process can send a signal to another process using the `kill()` system call.

Signals: Handling

There are three options for handling signals:

1. Use the default action `SIG_DFL`.
2. Ignore the signal (i.e., the signal does nothing to your process).
3. Write a signal handler function, which will be called automatically upon receipt of a signal.

Changing the Default Action

Two options for changing the default signal action:

2. The `signal()` C standard library function.
3. The `sigaction()` system call.

`signal()` is cross-platform but more limited in scope, whereas `sigaction()` is more flexible but found only on POSIX.1-compliant systems.

More in the GNU C library manual

`www.gnu.org/software/libc/manual/html_node/Signal-and-Sigaction.html`

Signals: Handlers

Related code is provided `signals.h`.

`signal` simply points each signal to a handler function.

```c
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void handler() { printf("in signal handler\n"); }
int main() {
    for (int i = 0; i < 256; i++)
        signal(i, handler); // run handler() for every signal
    printf("starting\n");
    sleep(10);
    return 0;
}
```

Arnold's Examples

`mcs.utm.utoronto.ca/~209/23s/lectures/arnold/w07/signals/4handler.c`

Signals: Actions

Related code is provided `signals.h`.

`sigaction` is the name of a system call function.

```
int sigaction(int sig,
    const struct sigaction *act,
    struct sigaction *oldact);
```

And it's also the name of the struct type that it takes as the 2nd and 3rd arguments:

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

- `sa_handler` can be set to `SIG_IGN` (ignore), `SIG_DFL` (default action), or the address of a handler function (see `man 2 sigaction`).

Signals: Actions (Example)

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
int i = 0;
void handler(int signo) {
    fprintf(stderr, "Sig %d; total %d.\n", signo, ++i);
}
int main(void) {
    struct sigaction newact;
    sigemptyset(&newact.sa_mask);
    newact.sa_flags = 0;
    newact.sa_handler = handler;
    if (sigaction(SIGINT, &newact, NULL) == -1) exit(1);
    if (sigaction(SIGTSTP, &newact, NULL) == -1) exit(1);
    for(;;); //Infinite loop
}
```

Signals: Limitations

Signals are useful, but there are some drawbacks.

Information

- Signals convey no information, aside from what type of signal (e.g., `SIGINT`, `SIGUSR1`) it is.
- Generally only used to indicate abnormal conditions: Not for data exchange.

Queuing

- Multiple instances of the same signal do not queue.
- If signal `Y` is sent while a previously-sent signal `X` is pending, then the second `Y` is lost.
- Example, if your process receives a `SIGCHLD` (Child stopped or terminated), it may be that only one child process has terminated, or that *multiple* child processes have terminated.

Aside: Types and Portability

What is a `sigset_t` anyway? Good opportunity to demonstrate the primary need for `typedef`: allowing us to write *portable code*.

From `x86_64-linux-gnu/bits/types/sigset_t.h`:

```
typedef __sigset_t sigset_t;
```

From `x86_64-linux-gnu/bits/types/__sigset_t.h`:

```
#define _SIGSET_NWORDS (1024 / (8 * sizeof (unsigned long int)))
typedef struct {
    unsigned long int __val[_SIGSET_NWORDS];
} __sigset_t;
```

The type `__sigset_t` might be declared differently depending on your system, but code you write will be portable thanks to `typedef`. Try searching for the `#include` directive in `signal.h`.

Signal: Sets as Bit Vectors

A signal set (`sigset_t`) is a *bit vector* that specifies the set of signals to block; operate on them using the following standard library functions:

1. `int sigemptyset(sigset_t *set);`
2. `int sigfillset(sigset_t *set);`
3. `int sigaddset(sigset_t *set, int signo);`
4. `int sigdelset(sigset_t *set, int signo);`
5. `int sigismember(const sigset_t *set, int signo);`

See `man 3 sigsetops` for usage.

Note: recall the earlier discussion on bitwise operators.

Signals: Masking During Program Execution

As our program may receive signals spontaneously at any time, we may need to block some signals from being delivered at an inopportune moment (e.g., writing to disk).

- This *temporary* block is different from ignoring a signal entirely.
- Use the `sigprocmask()` system call to examine or change the set of blocked signals, via a *mask* (i.e., a bit vector representing a set of signals).

Here's an example:

```
$ sigset_t set, oldset;
$ sigemptyset(&set);
$ sigaddset(&set, SIGINT);
$ sigprocmask(SIG_BLOCK, &set, &oldset);
/*... Critical operation ...*/
$ sigprocmask(SIG_SETMASK, &oldset, NULL);
```

Signals: Masking During Handler Execution

From `man 2 sigaction`:

> *sa_mask specifies a mask of signals which should be blocked (i.e., added to the signal mask of the thread in which the signal handler is invoked) during execution of the signal handler. In addition, the signal which triggered the handler will be blocked, unless the SA_NODEFER flag is used.*

Happy Birthday Example (Adapted from Karen's)

`github.com/rhubarbwu/csc209/blob/master/lectures/lec07/birthday.c`

- Try uncommenting both `sigaction` calls.
  - You might notice you're stuck!
  - Open another terminal and use `kill`.
- Then, for `SIGINT`, try using `sigaddset` instead of `sigaction`.

Signals: Async Safety

From `man 7 signal-safety`:

> *Suppose a program is in the middle of a call to a `stdio` function such as `printf` where the buffer and associated variables have been partially updated. If, at that moment, the program is interrupted by a signal handler that also calls `printf`, the second call to `printf` will operate on inconsistent data, with unpredictable results.*

Happy Birthday Example (Adapted from Karen's)

`github.com/rhubarbwu/csc209/blob/master/lectures/lec07/birthday.c`

- Uncomment the `printf` and `sleep` calls in the infinite loop.

Signals: Writing to a Broken Pipe

Remember this?

```
$ ssh cslab
rupert@apps0:~$ client_loop: send disconnect: Broken pipe
```

- This happens when an `ssh` connection is left in-active for a long time.

`SIGPIPE` is sent to a process that tries to write to a pipe or to a *socket* that does not have any readers.

More about processes and pipes later...

Signals: Other Ways of Sending

- Recall `grep` searches for matching patterns.
  - `pgrep` search PID's for matching program names.
- For `kill` sends signals by PID(s).
  - `pkill` sends signals by matching program names.

```
$ pkill -SIGINT hbd
$ pkill hbd
```

- Some other applications like `htop` also allow sending signals.

Section 5

Midterm Test

Midterm Test

- Time: 7-9pm
- Location: Instructional Centre (IB)
  - IB110: LEC0102/0103
  - IB120: LEC0101/0104
  - Go to your assigned room.
- Use the washroom and have a snack beforehand.
- Sit at least two seats apart from each other.
- No aids of any kind; put your devices away.