UMBC
TRAINING CENTERS

Operating Systems

TCPRG4004-2021-04-08

Table of Contents

Chapter 1. Fundamental Concepts

- Learn the fundamental building blocks of POSIX applications.
- Examine practical programming techniques.
- Explore security concerns from a variety of viewpoints.

Introduction

All operating systems have essentially one task: **Manage resources**.

Those resources take the form of memory, cpu time, disk I/O bandwidth, network bandwidth, and so on. But in order for an operating system to be useful, it has do its job in an efficient manner that is also easy to use. These are some of the added facilities to make systems more usable:

- Graphical user interfaces,
- Programming languages,
- Networking capabilities,
- Automation tools, and
- many others.

The goal in this course is to look at that basic purpose in detail, and to explore how those pieces fit together.

This course will look at four high level topics:

1. Process execution environment (memory map, cpu scheduling, input/output)
2. System calls vs. library functions
3. Error handling
4. System data types

Process Execution Environment

When a program is loaded and begins execution, a number of resources must be initialized for it. The most obvious ones are:

- Virtual memory, to hold the program code and all of its data.
- Cpu time, so that the program code can be scheduled for execution.
- Structures to provide the ability to read and write data.

Each of those items requires a large amount of support infrastructure within the operating system so that applications don't have to know or deal with specifics that might change from platform to platform.

Virtual Memory

- Virtual memory requires management at three levels:
  - Process address space must be grouped by usage (called `struct vm_area`).
  - The **vmarea**s may be shared between processes, and may shrink or grow on demand.
  - Physical RAM must be allocated and deallocated, including heuristics for making the best use of RAM under low memory conditions.

Cpu Time

- Cpu time must be apportioned to individual tasks:
  - There's only so much physical cpu time available, so algorithms must have as little overhead as possible and be scalable.
  - In order to make programming portable between POSIX systems, applications work in *virtual cpu*s that can be assigned to physical cores as those cores are available.

Persistent data storage

- Persistent data storage requires management of long term data structures:
  - Disk space is separated into partitions to give the sysadmin flexibility.
  - Each partition must maintain records of which blocks are inuse or free, and must do so in a way that allows for quick allocation.
  - Metadata representing files and directories must be organized and quick to access.
  - Applications must be forced to *open* these resources so that the operating system can control

access.

- Applications must be able to read and write files both sequentially and randomly (jumping around within a file prior to a read or write operation).

System Calls vs. Library Functions

System calls are hooks directly into the operating system, while library functions are implemented entirely in user space.

- System calls tend to be slower when they are invoked repeatedly to process small amounts of data, but provide core functionality that cannot be achieved via a library function.
- System calls are part of the POSIX definition, while library functions are typically defined by the programming language being used.
- System calls execute within the operating system's address space and are thus not limited by privilege levels as user space libraries are. (Essentially, all system calls execute with "superuser privileges".)

This course will make significant use of system calls.

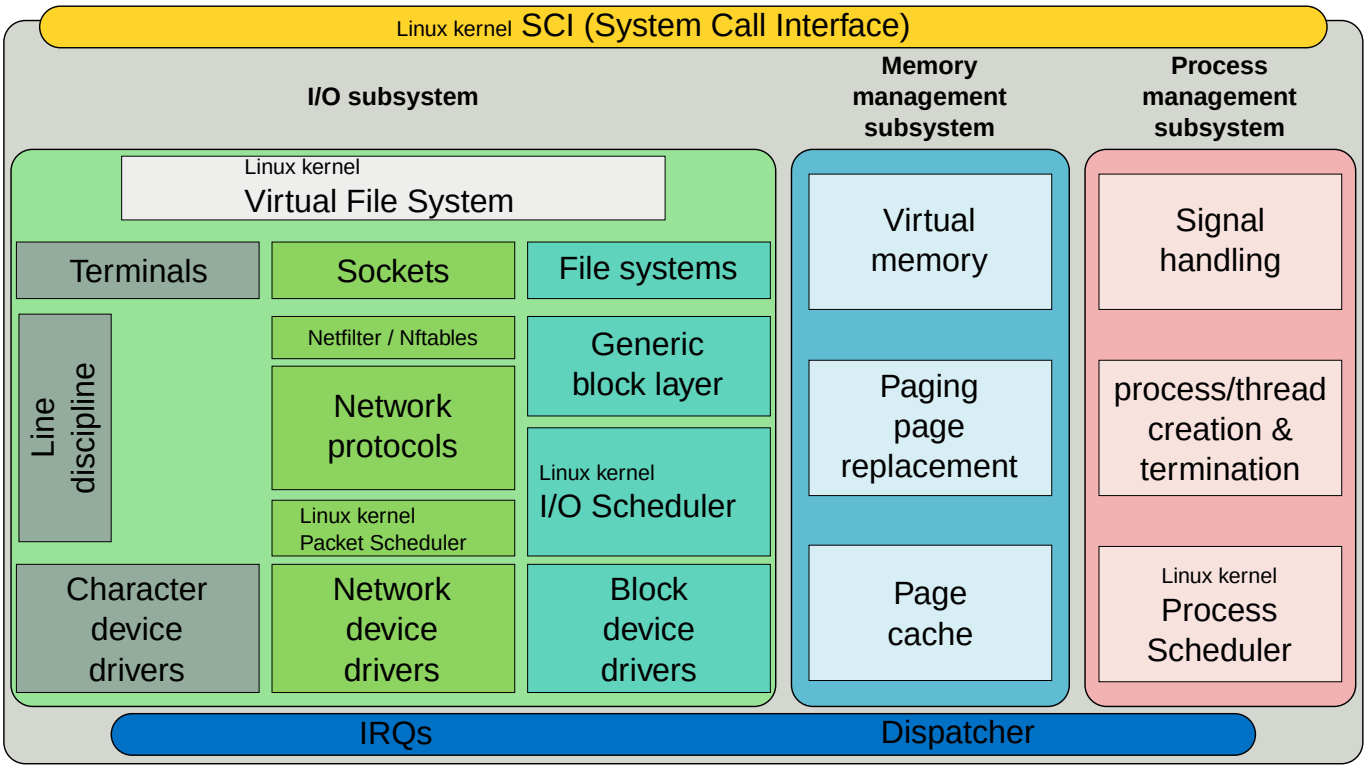


Figure 1. By ScotXW - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=47075153

When reading online documentation, there may be references to man pages in the format of `function(number)` where `function` is the name of the item being documented and `number` is the section of the man page that the item appears in. On Linux systems, executing `man man` will display the documentation for the `man` command and therein is documented the list of available sections.

For this course, section 2 (system calls) and section 3 (library functions) are the most important.

Error Handling

Error handling on POSIX systems comes in many forms, but there are two basic approaches that dominate common usage:

1. Return values from system calls.

   As one might expect, when a system call fails, it returns an error code to the caller, which is expected to check the return value and take appropriate action on an error.

   This tends to be a less than ideal approach, since it's simple for a programmer to fail to check the return value!

2. Signals can be sent for events which are not system call-related.

   For example, accessing an invalid memory location can cause a signal to be delivered if the memory address is invalid or the access itself is not allowed. Attempting to write to memory that is read-only will trigger a ***segmentation fault*** signal (`SIGSEGV`). Access of any time to a memory address that is not part of a process's address space may also trigger a segmentation fault.

This course will be discussing when and how each approach should be used, but in general, system calls will typically return `-1` on failure, while other errors will be reported via signals.

System Data Types

Data types within the operating system are carefully chosen to represent the needed values without using more memory than is actually required. Sometimes these data types correspond directly with data types provided by a programming language, and sometimes they don't.

In order to be consistent across all languages and all hardware platforms where a POSIX operating system may be implemented, standard data types are defined and must be used when making system calls.

Application Data Types

This course is going to focus on application programming in the C language, so it's important to be familiar with the C language core data types and how those data types map to the ones used inside the operating system.

Table 1. Core C Data Types

Type	`printf` Format	Number of Bits Per Data Model				
		C Standard	LP32	ILP32	LLP64	LP64
`char`	???	>= 8	8	8	8	8
`unsigned char`	`%hhu`					
`signed char`	`%hhd`					
`short int`	`%hd`	>= 16	16	16	16	16
`unsigned short int`	`%hu`					
`int`	`%d`	>= 16	16	32	32	32
`unsigned int`	`%u`					
`long int`	`%ld`	>= 32	32	32	32	64
`unsigned long int`	`%lu`					
`long long int`	`%lld`	>= 64	64	64	64	64
`unsigned long long int`	`%llu`					

The data models referenced above are choices made by particular implementations of the C language for a given platform.

- For 32-bit systems:
  - LP32 (a.k.a. 2/4/4) have 16-bit integers, and 32-bit longs and pointers. Example: Win16 API.
  - ILP32 (a.k.a. 4/4/4) have 32-bit integers, longs, and pointers. Examples: Win32 API, Unix and Unix-like systems (Linux, macOS).

- For 64-bit systems:
  - LLP64 (a.k.a. 4/4/8) have 32-bit integers and longs, and 64-bit pointers. Example: Win64 API.
  - LP64 (a.k.a. 4/8/8) have 32-bit integers, and 64-bit longs and pointers. Examples: API, Unix and Unix-like systems (Linux, macOS).
- Whether a bare `char` is `signed` or `unsigned` is platform- and compiler-dependent.
- Other models exist but are extremely rare.
- Note that exact-width integer types are available in `stdint.h` as of C99.

Operating System Data Types

The previous chart is useful, but it doesn't help when communicating with the operating system. The operating system defines its own data types for things like *process id* and *file size field*. In an ideal world, programmers could use the operating system's `typedef`s and not have to worry about the actual data type. But in the real world, how would they print a process id if they don't know its size? Do they use `%d` or `%ld` or something else?

Here is a list of the most common data types defined in POSIX and what the real type is behind it. The student may want to fold the corner down on this page—it will be referred to a lot throughout the course!

These data types are defined in terms of the LP64 memory model, i.e., a 64-bit Linux system.

Table 2. POSIX Data Types

Type	Definition (on LP64)	Size	Use
`blksize_t`	`int`	32	Block I/O size
`caddr_t`	`char *`	64	Address in memory (almost equivalent to `void *`)
`daddr_t`	`long`	64	Disk address (or "block number")
`id_t, gid_t, pid_t, uid_t`	`int`	32	Process id, group id, user id
`ino_t`	`unsigned long`	64	Filesystem inode number
`mode_t`	`unsigned int`	32	File modes (file type, permissions)
`off_t`	`long`	64	File offset
`offset_t`	`long`	64	Same as `off_t`
`pthread_*_t`	(discussed separately)		
`size_t`	`unsigned long`	64	Size of something (file size, `sizeof`, etc)
`ssize_t`	`long`	64	Signed size

Type	Definition (on LP64)	Size	Use
`time_t`	`long`	64	Time since the epoch (typically 1970-Jan-01)

Summary

This chapter has been an overview of the fundamentals necessary to understand upcoming topics.

- The process execution environment
- System calls vs. library functions
- Error handling techniques
- Mapping system data types to C language data types

Chapter 2. Files

Students will learn how files are managed on POSIX systems, both concepts (such as inodes and associated metadata) and application access (via system calls).

- Inodes, and how information is stored on the disk
- Retrieving file information via `stat()`
- Allowable access modes (in other words, "permissions")
- Changing file attributes such as ownership and timestamps
- Various file descriptor-related system calls

Introduction

Every operating system must support some form of ***persistent storage*** if it's going to process any significant data set or produce reports of any significant size. This course will be referring to such storage as "disk storage" or "disk space", but it really encompasses a wide variety of actual storage techniques, like solid state disks, network attached storage, and so on.

Essentially, that means some way to load and store data—users are NOT going to want to type in their data every time they run the application!

This means a generic interface for manipulating disk storage is necessary. On POSIX systems, the primary data structure for this is called an ***inode***. (It appears that the word is originally derived from ***index node***, as it was a simple index number into an array of data structures, but it may be helpful to think of it instead as ***information node***, since that's really what it is—the inode contains all of the metadata for a single item of storage on the disk.)

There is one inode allocated for:

1. The root directory of a partition
2. All subdirectories and files underneath the root directory
3. Many filesystems reserve a few inodes for administrative purposes

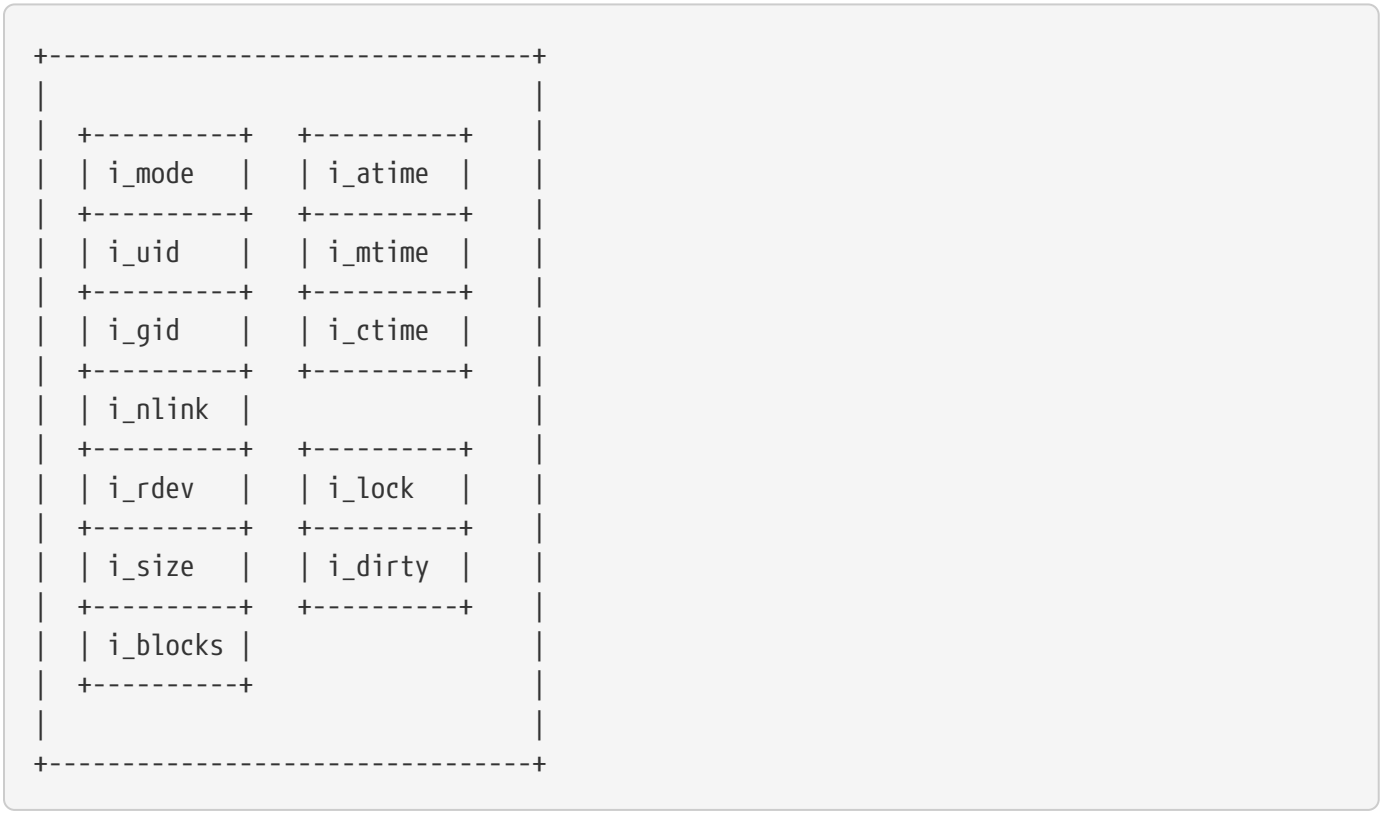
Inode number zero always indicates an unallocated inode. For example, in a directory, each entry will have an associated inode number, but empty slots in the directory will be denoted by an inode number of zero.

Contents of an Inode

Each *filesystem* defines what the structure of an on-disk inode looks like. The term *filesystem* is used to represent a particular storage format on the disk for directories and files. On Linux, common filesystems include **ext4** amnd **xfs**, but there are dozens available. Each one was created to solve a particular set of problems so each has advantages and disadvantages.

This image is a representation of the generic inode used by the operating system to represent all inodes. Each individual filesystem will have its own internal structure as well, which is implementation-specific.

Generic in-memory inode



The last two fields are pointers and never appear in the on-disk structure. They are used to keep track of which kernel threads are waiting on the inode lock to be released, and to track a linked list of all inodes whose buffers are dirty and need to be written to disk.

These fields are the same ones that are reported by `ls` when using the `-l` option. Applications can access this information using the `stat()` system call.

Using the `stat()` System Call

Applications can access the metadata of the inode using the `stat()` system call.

```
#include <sys/stat.h>

int stat(const char *pathname, struct stat *buf);
```

The `pathname` may be a relative pathname (meaning the directory search process starts with the application's current directory) or an absolute pathname (the application's current directory is not used and the search begins from the top-level directory).

TIP

In fact, **all** system calls that accept pathnames work like this. It's why Linux utilities all process pathnames the same way—they're simply be handed off to the operating system to do the heavy lifting.

The `struct stat` referenced in the function signature is defined in `sys/stat.h` and it contains the user space-visible portions of the inode:

```
struct stat {
    dev_t st_dev;             // ID of device containing the inode
    ino_t st_ino;             // Inode number
    nlink_t st_nlink;         // Number of hard links
    mode_t st_mode;           // File type and access modes
    uid_t st_uid;             // User ID of owner
    gid_t st_gid;             // Group ID of owner
    // ...
    dev_t st_rdev;            // Device ID (special files only)
    off_t st_size;            // Total size, in bytes (non-special files)
    blksize_t st_blksize;     // Block size for filesystem I/O
    blkcnt_t st_blocks;       // Number of 512B blocks allocated
    struct timespec st_atim;  // Time of last access
    struct timespec st_mtim;  // Time of last modification
    struct timespec st_ctim;  // Time of last inode change
    // ...
};
```

Many of the same fields are present, although this course hasn't discussed some of the opaque data types used inside this structure (looking up their definition is left as an exercise for the student).

Interpreting the Data

Many command line utilities, such as `cp` and `ls`, make use of the information provided by `stat()`.

1. When copying a file, permissions are preserved as much as possible
2. When overwriting a file while copying, permissions on the destination are used to determine whether to prompt the user for confirmation

   When `cp` is given a read-only destination, it will prompt for confirmation. If the user responds affirmatively, it will try to remove the destination that exists and create a new file. The ability to remove the destination is controlled by write permission on the directory (and whether the filesystem is mounted read-only), so the user may still receive an error message reporting failure.

3. When listing the contents of a directory, there are options to sort by file modification time, file access time, and so on
4. Applications can make use of `st_blksize` to perform efficient I/O
5. The `st_blocks` field is the number of 512-byte blocks actually allocated to the file

   This might be less than `st_size`/512 when the file has “holes” in it, such as when an application has performed a `seek(2)` beyond the end of the file and then written data. This causes the operating system to "extend" the file with physically unallocated data blocks.

For backward compatibility, there are three macros defined to access the seconds field of the `struct timespec` structure:

```
#define st_atime st_atim.tv_sec
#define st_mtime st_mtim.tv_sec
#define st_ctime st_ctim.tv_sec
```

The `struct timespec` consists of two fields, `tv_sec` to hold the whole number of seconds, and `tv_nsec` to hold the fractional portion, measured in nanoseconds.

Note that the interpretation of `st_mode` is complicated because the field contains two pieces of information: file type, and allowable access modes. (Details on these values can be found in the `inode(7)` man page.

1. Access modes are mapped to POSIX permissions, meaning that the bottom 9 bits of this field indicate read/write/execute permissions for each of the user, the group, and others POSIX refers to these as the *file permission bits* and sets a mask value of `0777` to isolate them.
2. Three additional bits indicate the set-uid, set-gid, and sticky bits POSIX refers to the *file permission*

bits, plus these 3, as *file mode bits*.

3. Four additional bits are interpreted as the file type

In order for an application to determine the file type, code similar to the following may be used:

```
struct stat sb;
stat(pathname, &sb);

if ((sb.st_mode & S_IFMT) == S_IFREG) {
    /* Handle regular file */
} else if ((sb.st_mode & S_IFMT) == S_IFDIR) {
    /* Handle directory */
}
```

Because code like the above is so common, macros are defined to reduce the repetitiveness:

```
struct stat sb;
stat(pathname, &sb);

if (S_ISREG(sb.st_mode)) {
    /* Handle regular file */
} else if (S_ISDIR(sb.st_mode)) {
    /* Handle directory */
}
```

The available macros are listed here (the parameter `m` is the `st_mode` value):

Macro	Notes
`S_ISREG(m)`	regular file
`S_ISDIR(m)`	directory
`S_ISCHR(m)`	character device
`S_ISBLK(m)`	block device
`S_ISFIFO(m)`	FIFO (named pipe)
`S_ISLNK(m)`	symbolic link (not in POSIX pre-200112)
`S_ISSOCK(m)`	socket (not in POSIX pre-200112)

Various mask macros are also provided to isolate individual permission bits.

Macro	Mask	Notes
S_IRWXU	00700	owner has read, write, and execute permission
S_IRUSR	00400	owner has read permission
S_IWUSR	00200	owner has write permission
S_IXUSR	00100	owner has execute permission
S_IRWXG	00070	group (not in user) has read, write, and execute permission
S_IRGRP	00040	group has read permission
S_IWGRP	00020	group has write permission
S_IXGRP	00010	group has execute permission
S_IRWXO	00007	others (not in user or group) have read, write, and execute permission
S_IROTH	00004	others have read permission
S_IWOTH	00002	others have write permission
S_IXOTH	00001	others have execute permission

This group of permissions are treated specially, as defined below the table.

Macro	Mask	Notes
S_ISUID	04000	set-user-ID bit
S_ISGID	02000	set-group-ID bit
S_ISVTX	01000	sticky bit

The S_ISUID bit:

1. When applied to a binary executable, causes the owner of the executable to be stored into the owner credentials of the process that results when the executable is invoked via execve(2).
2. Has no meaning in other contexts.

The S_ISGID bit:

1. When applied to a binary executable, causes the group of the executable to be stored into the group credentials of the process that results hen the executable is invoked via execve(2).
2. When applied to a directory, causes all future inodes allocated within the directory to inherit the group ID. Existing inodes are not affected. Subdirectories will also inherit the S_ISGID bit as well.
3. When used on an ordinary file and the S_IXGRP bit is not set, application access to the file will include mandatory file/record locking.

Mandatory record locking is discouraged. It can lead to deadlocks among applications that share a common file when the applications are not specifically written to accomodate mandatory locking.

The `S_ISVTX` bit (a.k.a., the "sticky bit"):

1. When applied to a binary executable, causes the operating system to not flush the read-only executable portions of the binary when the application terminates. This is a historical anachronism and is no longer supported on Linux (or any modern POSIX operating system).
2. When applied to a directory, causes the operating system to reject any changes to the contents of the directory by anyone other than the owner. For example, the `/tmp` directory typically has this bit turned on so that temporary files cannot be deleted or renamed by anyone other than the owner of said file. (This prevents data manipulation vulnerabilities that could otherwise occur.)

Changing Inode Properties

The previous section defined what the inode properties are; This section will describe how they can be changed.

All of these operations require *appropriate privileges*.

This is the POSIX way of saying “root authority”. They don’t want to use that phrase because it limits them to systems that actually have a "root" username. Also, Linux includes *capabilities*, documented under `capabilities(7)` in the man pages, that can provide *appropriate privileges* in some situations without requiring root authority.

Here are the system calls available and the inode field(s) they manipulate:

Table 3. System Calls that Manipulate Inode Fields

Field	System Call Notes
`st_mode`	`chmod(2)`
`st_nlink`	`link(2)`, `unlink(2)`
`st_uid`, `st_gid`	`chown(2)` (there is no `chgrp(2)` system call)
`st_atim`, `st_mtim`	`utime(2)` and `utimes(2)`
`st_rdev`	`mknod(2)` (can only be set at inode creation)

Changing Inode Properties on Open Files

Occasionally, it may be useful to modify the properties of a file when the only information available is the file descriptor (the small integer value provided by the operating system when a file is successfully opened).

For example, suppose that an application is passed an open file as stdin by the shell. But the application wants to copy the permissions of that file to a new file that it's creating—how can it obtain `st_mode`?

For that reason, the system calls in the previous table (and others, as shown below) have corresponding variants that can be provided a file descriptor instead of a pathname.

Table 4. System Calls that Manipulate Inode Fields of Open Files

Field	System Call Notes
`st_mode`	`fchmod(2)`
`st_uid`, `st_gid`	`fchown(2)` (there is no `chgrp(2)` system call)
`st_atim`, `st_mtim`	`utimensat(2)` and `futimes(3)`

NOTE

The `futimes(3)` library function invokes `utimesnsat(2)` to do the work. See the man page for the system call for notes on its non-standard API.

Table 5. Other System Calls that Operate on Open Files

System Call	Notes
`fchdir`	Change directory to the directory given by `fd`
`fstat`	Retrieve `struct stat` for given `fd`
`fstatfs`	Retrieve information for filesystem that contains `fd`
`fsync`	Wait until all modified buffers have been written to `fd`
`ftruncate`	Truncate existing file `fd` down to zero bytes

Examples

The first example demonstrates how to retrieve information about a particular file and then print out the data.

print-inode.c

```c
#include <stdio.h>
#include <stdlib.h>

#include <sys/stat.h>

int main(int argc, char **argv)
{
    (void)argc;
    while (*++argv) {
        struct stat sb;

        if (stat(*argv, &sb) < 0) {
            perror(*argv);
            return 1;
        }

        // There is no portable way to get formats for these types
        // (unlike the PRIXXX macros for standard integers).
        // The closest thing to portable is casting all values to
        // `long` before formatting.
        printf("%s:\n"
                "  st_dev     = 0x%lx,\n"  // hex
                "  st_ino     = %lu,\n"
                "  st_nlink   = %ld,\n"
                "  st_mode    = 0%lo,\n"  // octal
                "  st_uid     = %ld,\n"
                "  st_gid     = %ld,\n"
                "  st_rdev    = 0x%lx,\n"  // hex
                "  st_size    = %ld,\n"
                "  st_blksize = %lu,\n"
                "  st_blocks  = %ld,\n"
                "  st_atim    = %ld.%09ld,\n"
                "  st_mtim    = %ld.%09ld,\n"
                "  st_ctim    = %ld.%09ld\n"
                , *argv
                , (long) sb.st_dev
                , (unsigned long) sb.st_ino
                , (long) sb.st_nlink
                , (unsigned long) sb.st_mode
```

```
                , (long) sb.st_uid
                , (long) sb.st_gid
                , (long) sb.st_rdev
                , (long) sb.st_size
                , (unsigned long) sb.st_blksize
                , (long) sb.st_blocks
                , sb.st_atimespec.tv_sec,
                sb.st_atimespec.tv_nsec,
                sb.st_mtimespec.tv_sec,
                sb.st_mtimespec.tv_nsec,
                sb.st_ctimespec.tv_sec,
                sb.st_ctimespec.tv_nsec);
    }
}
```

And this example shows how to read the inode metadata from one file and apply it to other files.

copy-stat.c

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

#include <sys/stat.h>
#include <sys/time.h>

int main(int argc, char **argv)
{
    if (argc < 3) {
        fprintf(stderr, "Usage: %s <src> <dst>...\n", argv[0]);
        return 1;
    }

    struct stat sb;
    if (stat(*++argv, &sb) < 0) {
        perror(*argv);
        return 2;
    }


    while (*++argv) {
        // Apply as many of the fields as possible.
        // If an error occurs, skip it and keep going.
        chmod(*argv, sb.st_mode & 0777);
        chown(*argv, sb.st_uid, sb.st_gid);
        // truncate(*argv, buf.st_size);  // Damaging, but possible

        struct timeval both[2];
        both[0].tv_sec  = sb.st_atimespec.tv_sec;
        both[0].tv_usec = sb.st_atimespec.tv_nsec / 1000;
        both[1].tv_sec  = sb.st_mtimespec.tv_sec;
        both[1].tv_usec = sb.st_mtimespec.tv_nsec / 1000;
        utimes(*argv, both);
    }
}
```

Summary

This chapter has been an overview of how files are managed on a POSIX operating system.

- Inodes, and concepts regarding application access
- File attributes
  - How to access them
  - How to change them

Exercises

Exercise 1 – `set_x.c`

Write a program that will ensure the its command-line argument, a filename, is executable to all roles. Non-execute permissions should not be modified.

Exercise 2

Using the `ls -li` command to show a file's inode number, determine when your text editor changes the inode of the filename. Do `cp`, `dd`, or `mv` change the inode? Why or why not?

Chapter 3. File I/O

Students will learn POSIX access to data associated with inodes. Such data could consist of ordinary file data, directory contents, or even devices (depending on inode type).

- An overview of file I/O (input/output)
- The POSIX operations of `open`, `read`/`write`, and `close`
- How the ***file offset*** is stored and used
- Atomic operations — what does that mean?
- The relationship between file descriptors and open files
- Operations on file descriptors (duplication, status flags, locking)

Introduction

Applications must be able to read and write data to be useful. So, the operating system must provide some method of managing persistent storage, what most programmers would call ***disk storage***.

There have been operating systems in the past in which each type of device was accessed using a unique API. Most operations would include opening the storage device (which is where the operating system verifies the authority to access the resource), reading and writing to the device, and then closing the device (where any buffers are flushed).

- Access to a printer might be done using functions such as `openprt()`, then `readprt()` and `writeprt()`, followed by `closeprt()` when all access is complete.
- The same approach could be used for serial ports, such as modems or terminals.
- Access to the low-level disk drive itself could be through functions such as `openblk()` and `closeblk()` (disk drives are referred to as ***block devices*** because they are commonly accessed a block at a time, not a character at a time).

This is clearly not going to work well in the long-term! When a tape backup system is added, a new application has to be written to support that device!

Instead, the API must be generic across all types of storage. A single `open()` function, for example. When invoked, the operating system would somehow need to determine the device to access. On POSIX systems, this is done using the ***path name***.

Path Names

In early operating systems, storage was so limited that the OS didn't implement directories! There was a top-level location and all files were stored there. This is clearly very simple to implement, but is extremely limiting.

Modern operating systems implement *hierarchical file systems*.

There are two terms that this course will be using, *filesystem* and *file system*. The first represents the *physical* layout of how inodes and files are stored on the disk. The second is a *logical* organization of files, in which files and directories may be spread over multiple physical locations.

In these systems, there is a single top-level location, but information can be organized into any number and depth of subdirectories.

On Linux systems, it's common for the top-level location to have up to two dozen subdirectories, and each of those directories can contain more subdirectories. There is no limit to how deep these directories can be nested. (Although there is a practical limit—eventually, the system will run out of disk space to store the inodes!)

How is the user going to identify where they want their data to be stored? The system is going to need some way to specify a route from the top-level location down into the depths of that hierarchy. POSIX systems do this with *path names*.

There are two types of pathnames: *absolute pathnames* and *relative pathnames*. Did the reader notice that the space is missing between those two words in this paragraph? This is very similar to *filesystems* vs. *file systems*. On POSIX systems, leaving out the space refers to a specific instance of a path name, while putting the space in refers to the *concept*.

- Absolute pathnames **always** start with a / (slash) and **always** uniquely identify a location in the file system.

  Absolute pathnames tell the operating system to start at the very top level of the file system and follow a particular route to identify a specific location.

  ```
  /etc/passwd
  /usr/local/bin/python2.7
  /tmp/my_file
  /home/student/labfiles
  ```

- Relative pathnames **never** start with a / (slash) and **never** uniquely identify a location in the file system—that's why they're called *relative*, after all!

Relative pathnames tell the operating system to start the search with whatever directory the application happens to be in.

```
labfiles/oslabs/examples/files/copy-stat.c
../student
./my_file
```

Note that the result from calling `open()` doesn't change, regardless of which type of pathname is used.

```
$ cd /tmp               # Absolute pathname
$ ls -l my_file         # Relative pathname
$ ls -l /tmp/my_file    # Absolute pathname
$ cp /etc/* /tmp        # A hundred (or more) absolute pathnames!
$ cp /etc/* .           # A hundred absolute pathnames and one relative pathname
```

The POSIX I/O Procedure

Every application follows the same basic approach when performing I/O of any kind, as shown in the image, below.

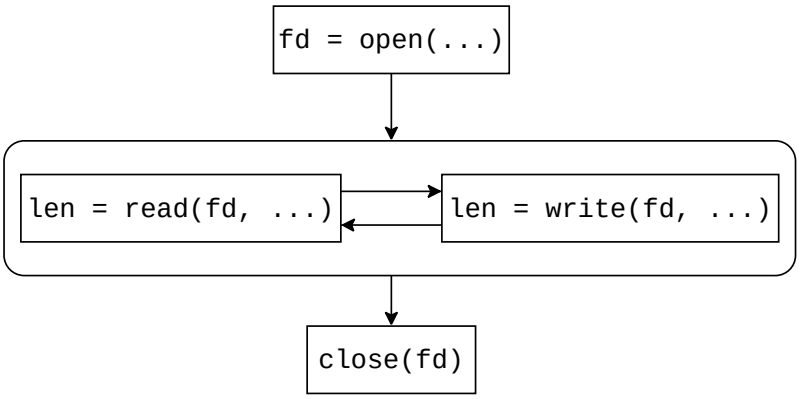


Figure 2. Application File I/O Flowchart

Here's a breakdown of each piece of the picture:

1. The `open()` function is where the operating system is given a pathname and it verifies allowable access and returns a ***file descriptor*** if access is allowed.

   For example, opening the file for read permission has one set of access checks, while opening the file for write permission has different checks.

2. The file descriptor is then used to reference the opened file.

   The `read()` and `write()` functions are given the file descriptor which is used by the operating system to maintain any necessary state information.

   On Linux, the file descriptor is an index into an array of `struct file` pointers, and the `struct file` contains all of the details regarding the opened file. Things such as how the file was opened (read, write, execute, or some combination of the three), the current offset into the file (i.e., where the next I/O operation will begin).

3. When the application is done with the file, it calls `close()` and passes the file descriptor.

   This is when resource deallocation occurs; buffers are flushed and freed, locks are released, and so on.

Using the `open()` System Call

Because all access to persistent storage is via the `open()` system call, this chapter will start there.

First, the function declaration as defined by the Linux man page:

```
#include <fcntl.h>

#include <sys/stat.h>
#include <sys/types.h>

int open(const char *pathname, int flags, mode_t mode);
```

Each call to `open()` produces a file descriptor that refers to the inode. It is therefore possible to have multiple file descriptors pointing to the same inode. However, each file descriptor will have its own *open mode* (such as read-only or read-write) and its own seek offset within the file (the address within the file where the next I/O operation will be performed).

The first parameter is the pathname to the inode content that the application wants to access. (The inode could be an ordinary file—the most likely case—but it could also represent a directory or a network socket or some other special file type.)

The `flags` parameter is a list of options. Each option is a sequence of bits (at least 1, but possibly more) and they are bitwise OR'd into the second parameter (but see the exception in the chart, below). The following flags are available (listed alphabetically):

Flag	Purpose
`O_APPEND`	Seek to end-of-file on every `write()`
`O_CREAT`	Create a new file if doesn't exist
`O_CLOEXEC`	Close on `execve()`
`O_EXCL`	Only used with `O_CREAT`; the file *must not* exist yet

`O_RDONLY`
`O_WRONLY`
`O_RDWR`
	Only one of these may be specified.
	Open the pathname read-only, write-only, or read-write

`O_TRUNC`	Truncate existing content to zero bytes

The `mode` parameter identifies the permissions that the application wants to set on a newly created file (such as when the `O_CREAT` flag is specified). It is ignored otherwise.

The process *umask* is still used to mask the permissions when `mode` is specified and the file is being

created. A common mistake is specifying the `mode` in decimal instead of octal! If using a constant instead of the macros from `sys/stat.h`, be sure to add a leading `0`.

Most flags are self-explanatory, but there are a few combinations to be aware of. Each requires appropriate permissions on the target file, such as write permission for `O_APPEND`.

1. When using `O_APPEND`, it is *impossible* to overwrite any portion of the file.
2. When using `O_CREAT`, a new file will be created if one doesn't already exist and the permissions stored in `mode` will be used (as described above).

   Note that this option cannot create device special files as there is no way to specify major and minor device numbers; use `mknod()` instead.

3. When using `O_CREAT` and the file already exists, this flag is ignored and the `mode` parameter is not used.
4. When using `O_CREAT` and `O_EXCL`, the `open()` only succeeds if the file does not already exist and the `open()` was able to successfully create a new one.
5. When using `O_CLOEXEC`, executing another application by calling `execve()` will cause the opened file to be automatically closed. (The `execve()` system call is covered in a separate topic.)
6. Specify `O_RDONLY` for read-only access, `O_WRONLY` for write-only access, and `O_RDWR` for both read and write access.

   Note that using `O_CREAT` *does not* imply write access, so some version of write access must still be requested if the programmer expects to be able to write to the newly created file!

7. When using `O_TRUNC`, the content stored under the pathname is truncated to zero bytes, i.e., all of the content is removed. (The use of `O_TRUNC` with `O_RDONLY` is undefined by POSIX. Some implementations truncate the file.)

If the `open()` is successful, the return value is the file descriptor assigned to this data stream. The seek offset associated with the file descriptor is initialized to zero (even when files are opened with the `O_APPEND` option).

WARNING	This system call returns `-1` on failure, but return values greater than or equal to zero are valid. Be careful when checking the return value for an error.

Reading and Writing

Once the file has been opened, the file descriptor can be used to read or write the contents of the file, as appropriate to the permissions requested when the file was opened.

For example, if the application requests `O_RDONLY` permission, it may read from the file but not write to it.

The vast majority of I/O to a file descriptor is via the `read()` and `write()` system calls. There is a scatter-gather implementation of these calls in `readv()` and `writev()`, and there is glibc library support for asynchronous I/O via `aiocbp()` (see `man 7 aio` for details). But those are beyond the scope of this course.

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

In both cases, the address of a buffer is provided along with the number of bytes to read or write. The return value is always the number of bytes *actually* transferred, along with `-1` to indicate an error and `0` to indicate EOF. For file descriptors that represent non-blocking I/O, it's possible the return value will be `0` but there may be data available in the future. An example is checking a network socket to see if data is available, but if there isn't any, the program would want to continue executing.

NOTE

The number of bytes actually transferred may not be the same as the number requested. A `read` operation on a terminal, for example, may return less data than requested because the user pressed **Return**. Similarly, a `write` operation may write less than the number specified if the device runs out of space.

Both functions have the same parameters and the same return values, except that the buffer for `read()` cannot be `const` (the destination buffer must be writable).

Reads and writes always begin at the current seek offset associated with the file descriptor. If the application wants to change the seek offset, a call to the `lseek()` system call accomplishes that:

```
#include <unistd.h>

#include <sys/types.h>

off_t lseek(int fd, off_t offset, int whence);
```

This system call uses `off_t` to represent a seek offset that is 64-bits in size on platforms that support 64-bit file sizes. (Some 32-bit systems may support 64-bit file sizes, so this `typedef` is defined appropriately.)

The `whence` parameter identifies whether the `offset` is relative to the beginning of the file (`SEEK_SET`), the end of the file (`SEEK_EOF`), or the current position (`SEEK_CUR`).

NOTE

It is not an error to specify a seek offset beyond the end of the file. Such operations allow an application to seek to an arbitrary location in the file and then read or write at that location.

Reads will always return zero bytes (essentially, treated as EOF) and writes will extend the file size to the seek offset plus the number of bytes written. (All intervening data blocks between the prior EOF and the newly written data will not be physically allocated, but will appear to exist when an application attempts to read that region; unallocated blocks appear to contain null bytes. Such files are referred to as "holey" files because there is a "hole" in the disk allocation.)

The return value from `lseek()` is always the new (absolute) position within the file.

Closing the File Descriptor

When the program is done using the file, close the file descriptor. While this may not be technically required (most POSIX systems allow thousands of files to be opened simultaneously), it's good programming practice in general to release any resources that are no longer need as soon as possible.

```
#include <unistd.h>

int close(int fd);
```

Technically, this system call can return an error. One such error is `EINTR` (interrupted system call). However, the man page for `close()` explicitly documents why `close()` should never be retried, and further clarifies that the return value should be used only for diagnostic purposes.

Atomic Operations

POSIX systems define a maximum size for file reads and writes that are guaranteed to be ***atomic***.

Atomicity is the granularity of I/O operations that cannot be interrupted by other I/O operations taking place on the same inode.

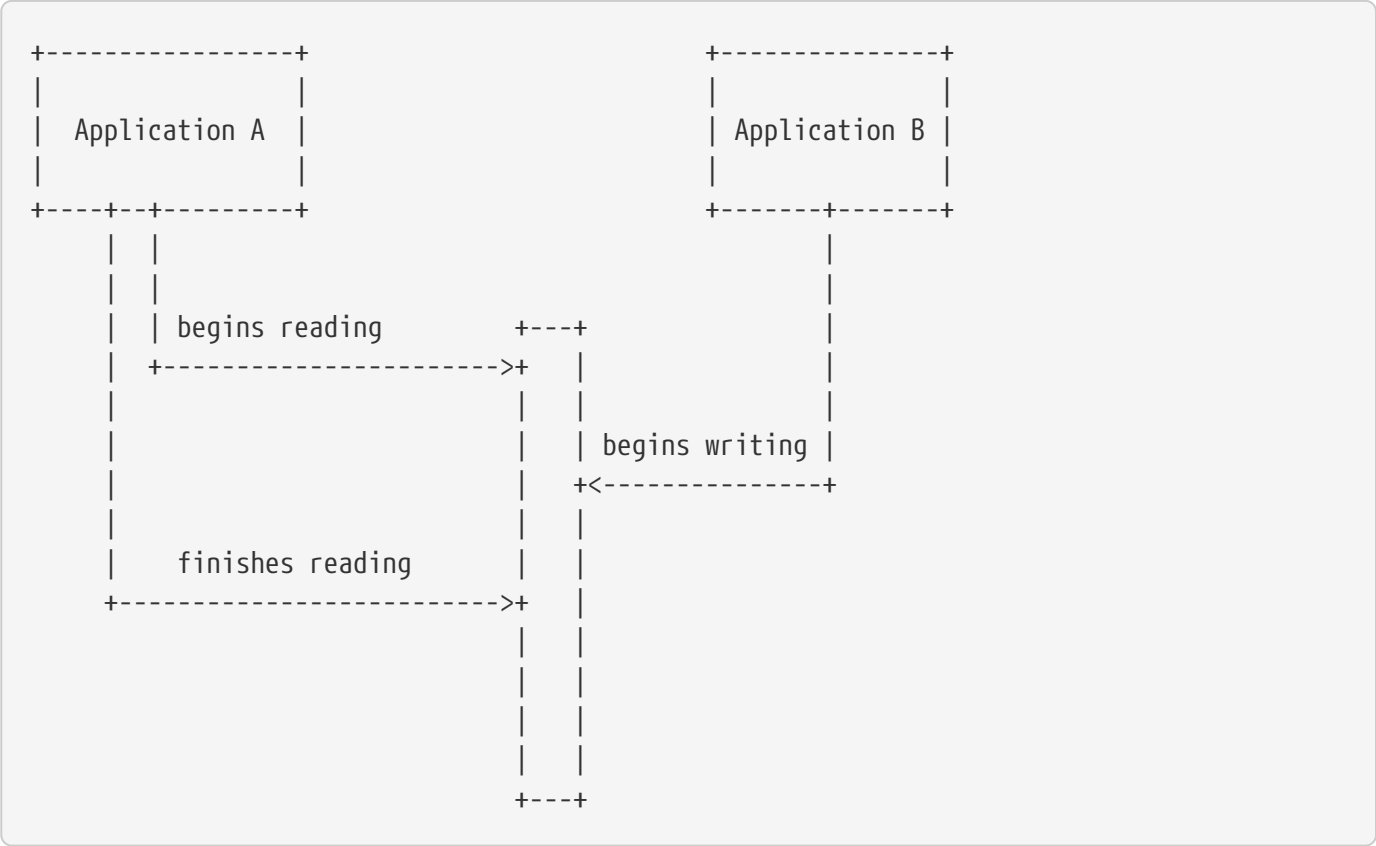
Without atomicity, it would be possible for an application to be reading a file while another application is writing to the same file such that data is corrupted. If both applications were performing I/O in overlapping regions of the file, the following scenario could result:

1. Application **A** opens the file and begins reading 100 bytes.
2. Meanwhile, Application **B** opens the file and begins writing 100 bytes.
3. Application **A** could read a partial buffer (perhaps 40 bytes) and then Application **B** could preempt **A** and write the entire 100 bytes.

What happens when Application **A** finishes reading?

The first 40 bytes of the buffer will have the file's original data, then **B** will have overwritten the first 100 bytes, so when **A** runs again, the remaining 60 bytes of the buffer will have the information just written by **B**!

When Atomicity Doesn't Exist



POSIX defines `read()` and `write()` to be atomic to `PIPE_BUF` bytes on ordinary files. This means that as long as an application performs I/O in sizes of `PIPE_BUF` bytes or less, the above overlap cannot occur. This is sufficient for many cases, such as log files.

If an application must perform a larger amount of I/O, the operating system supports the creation of *advisory locks* on regions of a file.

This allows one application to *lock* a region of a file as a way of notifying other applications that it's going to be using that region. Other applications can ask the operating system first to see if a region is locked before attempting to perform I/O.

However, these locks are *advisory*, not *mandatory*. This means an application that doesn't check for locks could simply ignore them and perform the I/O. All applications must coordinate how and when locks will be used in order for them to be beneficial.

File Locking

This section doesn't deal with locking *an entire file*, but rather a range of bytes within a file. That range of bytes doesn't need to actually be allocated yet for the lock to be applied.

The following structures are used to manage locks.

```
#include <fcntl.h>
#include <unistd.h>

int fcntl(int fd, int cmd, ... /* arg */ );

struct flock {
    // ...
    short l_type;    // Type: F_RDLCK, F_WRLCK, F_UNLCK
    short l_whence;  // l_start is SEEK_SET, SEEK_CUR, SEEK_END
    off_t l_start;   // Starting file offset for lock
    off_t l_len;     // Number of bytes to lock
    // ...
};
```

In general, an application will add lock management around its I/O operations, obtaining a lock prior to the I/O and releasing the lock upon finishing. If an application tries to obtain a lock on a region that is already locked by another application, it will *block* waiting for the lock to be released. The term *block* means to wait for an operation to complete. Processes can block waiting for a timer, waiting for I/O to complete, waiting for a socket buffer to drain, and so on.

The Examples section shows a simple program that performs lock management when doing I/O to a data file.

Miscellaneous Operations

There are numerous miscellaneous operations that can be performed on open files beyond just simple I/O. Most applications don't need to make use of these, but they are mentioned here so the student knows they exist.

1. File descriptors can be duplicated via `dup()` and `dup2()`. This is useful when multiple file descriptors should refer to the same inode, such as when a shell user is redirecting both stdout and stderr to the same destination.
2. Flags can be manipulated that cause open file descriptors to be closed when an program calls `execve()`. This is equivalent to setting the `O_CLOEXEC` flag when opening the file (not discussed in this course).
3. An application can retrieve *file status flags* to determine how the file descriptor was originally opened (the flags include most `open()` options, such as `O_RDONLY` and `O_TRUNC`).

Examples

The first example creates a new file with 1,000 blocks and writes the letter `A` into the first byte of every block. Because it is calling `lseek()` to position the seek offset at the beginning of each block, it is constantly seeking beyond the end of the file (as described earlier). (Note the options passed to `open ()`, and the use of `fstat()` instead of `stat()` to avoid a potential race condition.)

one_byte_block.c

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

#include <sys/stat.h>

/*
 * Given a single filename on the command line, this program writes the letter
 * 'A' into the first byte of every disk block up to block number 1,000.  It
 * is destructive, so it ensures that the file being written doesn't already
 * exist before beginning...
 */
int main(int argc, char **argv)
{
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>.\n", argv[0]);
        return 1;
    }

    // Only create it if it doesn't already exist.
    int fd = open(argv[1], O_CREAT|O_EXCL|O_WRONLY, 0666);
    if (fd < 0) {
        perror(argv[1]);
        return 2;
    }

    struct stat buf;
    // Retrieve the disk block size for this file descriptor.
    if (fstat(fd, &buf) < 0) {
        perror(argv[1]);
        return 3;
    }

    printf("Blocksize is %d.\n", buf.st_blksize);
    size_t location = 0;
    for (int n = 0; n < 1000; ++n, location += buf.st_blksize) {
```

```
        if (lseek(fd, location, SEEK_SET) < 0) {
            perror("Unable to seek");
            return 4;
        }

        if (write(fd, "A", 1) < 0) {
            perror("Unable to write");
            return 5;
        }
    }

    close(fd);
}
```

The second example simply copies all of the data from one file to another. It does not attempt much error checking at this point, but it will as it evolves.

NOTE

It is possible to include debugging output inside the `get_lock()` and `release_lock()` functions that indicates when particular locks are being obtained or released. It would be possible to post-process that output to ensure that every lock is released properly. Exploration of this possibility is left to the student.

copy-file.c

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

#include <sys/stat.h>

void get_lock(int fd, int type, off_t offset, size_t len);
void release_lock(int fd, off_t offset, size_t len);

/*
 * Given two pathnames on the command line, this program reads the first one
 * and writes to the second one.  The file being written is truncated if it
 * exists (and permission is allowed).
 */
int main(int argc, char **argv)
{
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <inputfile> <outputfile>\n");
        return 1;
    }
```

```
    // Open the input file first before opening the output.
    int input = open(argv[1], O_RDONLY);
    if (input == -1) {
        perror(argv[1]);
        return 2;
    }
    // Create it if it doesn't already exist, truncate otherwise.
    int output = open(argv[2], O_CREAT|O_TRUNC|O_WRONLY, 0666);
    if (output == -1) {
        perror(argv[2]);
        return 3;
    }
    // Errors at this point should probably cause the partial
    // output file to be deleted.  We're not doing that as the
    // partial contents may be helpful in debugging...

    // We want the I/O to be as efficient as possible, so
    // retrieve the disk block size for this file descriptor.
    struct stat buf;
    if (fstat(input, &buf) < 0) {
        perror(argv[1]);
        return 4;
    }
    printf("Blocksize is %d.\n", buf.st_blksize);

    size_t buffer_size = buf.st_blksize;
    char *buffer = malloc(buffer_size);
    if (!buffer) {
        fprintf(stderr, "Out of memory\n");
        return 5;
    }

    off_t offset = 0;
    for (;;) {
        get_lock(input, F_RDLCK, offset, buffer_size);
        ssize_t len = read(input, buffer, buffer_size);
        // Was there an error while reading the file?
        // (Partial reads are okay.)
        if (len < 0) {
            perror("Reading from input error");
            break;
        }
        if (len == 0) {
            break;
        }

        get_lock(output, F_WRLCK, offset, len);
```

```
        ssize_t written = write(output, buffer, len);
        // What if written < len?
        if (written < 0) {
            perror(argv[2]);
            return 6;
        }
        release_lock(input, offset, buffer_size);
        release_lock(output, offset, len);
        offset += len;
    }

    free(buffer);
    release_lock(input, offset, buffer_size);
    close(input);
    close(output);
}

void get_lock(int fd, int type, off_t offset, size_t len)
{
    struct flock lock = {
        .l_type = type,
        .l_whence = SEEK_SET,
        .l_start = offset,
        .l_len = len,       // zero means to EOF
    };
    if (fcntl(fd, F_SETLKW, &lock) < 0) {
        perror("Setting lock");
        exit(7);
    }
}

void release_lock(int fd, off_t offset, size_t len)
{
    struct flock lock = {
        .l_type = F_UNLCK,
        .l_whence = SEEK_SET,
        .l_start = offset,
        .l_len = len,
    };
    if (fcntl(fd, F_SETLKW, &lock) < 0) {
        perror("Releasing lock");
        exit(8);
    }
}
```

Summary

This chapter has been an overview of how file input/output is performed on a POSIX operating system.

- The need for generic input/output operations
- A review of POSIX pathnames
  - Absolute — always start with a slash
  - Relative — never start with a slash
- An overview of typical POSIX I/O
  - The `open()` system call
  - The `read()` and `write()` system calls
  - The `lseek()` system call
  - The `close()` system call
- An overview of record locking
- Miscellaneous filedescriptor operations
- Examples demonstrating basic techniques

Exercises

Exercise 1 – `cp_plain.c`

Write a program that will copy a file. The new copy of the file should have permissions `666`.

Exercise 2 – `cp_match.c`

Modify the previous exercise so that the permissions of the new copy match that of the original.

Exercise 3 – `block_search.c`

Write a program that takes a string to search for and a filename. The program should print two numbers: the number of blocks of the file containing that string, and the total number of blocks used by the file.

Chapter 4. File Buffering

Students will compare and contrast ANSI C file I/O with POSIX file I/O, including a discussion of buffering techniques in both the application address space and the kernel.

- Why is buffering necessary?
- Buffering inside the kernel vs. buffering in user space
- How to decide on the best buffering sizes

Introduction

Data is stored on disk as bytes (typically 8-bit values). Such data consists of ASCII characters, whole numbers stored in binary, floating point stored in IEEE-754 format (also binary), and so on. But does the programmer want to perform I/O in such small sizes? No, because efficiency would suffer. There is overhead in setting up an I/O transfer, so increasing the amount of data being written reduces the cost of the overhead.

Disk drives are optimized to perform I/O in larger chunks than individual bytes or even small numbers of bytes at a time. Legacy disk drives might have used 128-byte ***sectors*** or 512-byte sectors, but it's common for modern devices to use 4096-byte sectors.

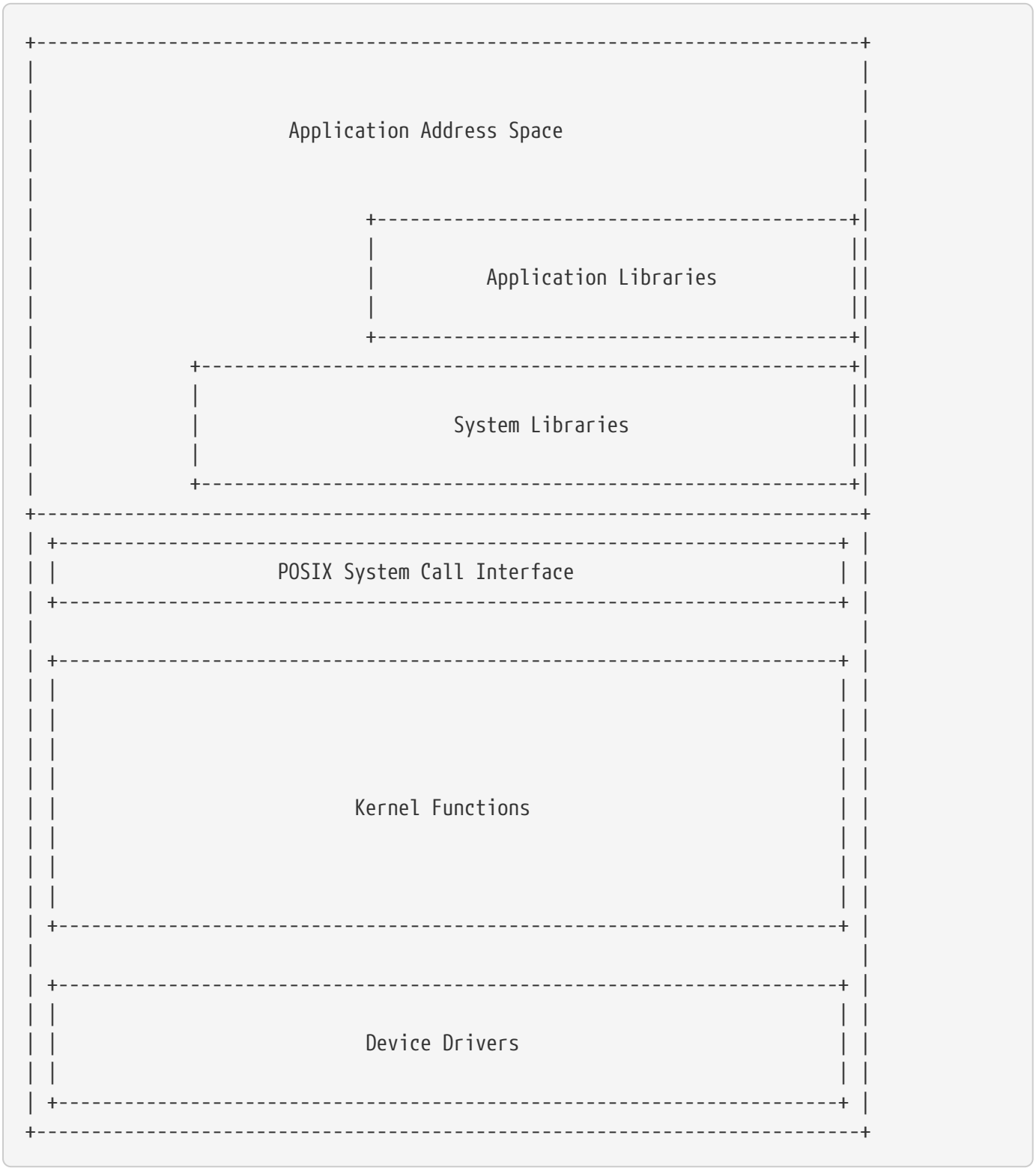
(A ***sector*** is the minimum physical I/O operation that the disk is capable of performing. If an application wants to write a single byte, the operating system must read the entire sector from the disk into RAM, modify the appropriate byte, then write the sector back out.)

Knowledge of the sector size allows an application to create its own buffers that are aligned with the requirements of the disk drive, but the issue goes beyond just the physical sector size of the disk. Frequently there are other size concerns as well. Filesystems often implement read-ahead and write-behind with some specified size for each. Should the application try to factor those into account as well?

In general, the filesystem's job is to hide low-level details of disk storage from the application programmer and it works very well. It's only when the programmer is concerned about efficiency, that they might wish to understand more about how the various buffering layers actually work.

The goal of this chapter is to explore the buffering layer provided by the ANSI C language and contrast it with the buffering layer provided by a POSIX operating systems. Proper use of each produces the most efficient code!

Buffering Layers



Buffering Provided by ANSI C

The C language itself doesn't provide any I/O operations. All I/O operations are performed by functions that are part of a library (typically referred to as *the standard C library*).

Because C is designed to be portable across operating systems, it defines structures and functions that provide an abstract file access API. Part of that API is file buffering.

Operations provided by the language use the `FILE *` data type extensively. Functions that operate on `FILE *` objects usually—but not always—start with the letter `f`, as in `fopen()`, `fread()`, `fwrite()`, and `fclose()`. (There are functions that do NOT start with an `f`, such as `setvbuf()`, `clearerr()`, and `rewind()`. And just because a function name starts with an `f`, does not mean that it takes a `FILE *` object as a parameter, such as `fabs()`.)

The structure behind `FILE` is *opaque*, but in practice there are certain fields it must contain, such as the operating system's file descriptor, flags describing how the file was opened, and fields to manage the file buffer. (The term *opaque* is used to mean that the specifics of the structure are not relevant to the application programmer. The phrase *black box* comes to mind—as long as it does what it's supposed to do, it doesn't matter (to the programmer) *how* it does it.) It is the file buffer that this chapter is interested in.

The standard C library implements a platform-specific buffer size. This means that all I/O performed through the standard I/O streams (`stdin`, `stdout`, and `stderr`) are automatically buffered. Calls to functions like `printf()` or `getchar()` use those buffers.

The ANSI C buffers provide a huge performance benefit. Consider the performance of these two equivalent programs.

using-ansi.c

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


int main(int argc, char *argv[])
{
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <infile> <outfile>\n", argv[0]);
        return 1;
    }

    FILE *input = fopen(argv[1], "r");
    if (!input) {
        perror(argv[1]);
```

```
        return 2;
    }

    FILE *output = fopen(argv[2], "w");
    if (!output) {
        perror(argv[2]);
        fclose(input);
        return 3;
    }

    char buffer;
    int return_code = 0;

    for (;;) {
        size_t rlen = fread(&buffer, 1, sizeof(buffer), input);
        if (ferror(input)) {
            perror("Unable to read");
            return_code = 4;
            goto cleanup;
        }
        if (feof(input)) {
            break;
        }

        size_t wlen = fwrite(&buffer, 1, rlen, output);
        if (wlen != rlen || ferror(output)) {
            perror("Unable to write");
            return_code = 5;
            goto cleanup;
        }
    }

cleanup:
    fclose(input);
    fclose(output);

    return return_code;
}
```

```
$ time ./using-ansi x y

real    0m0.223s
user    0m0.218s
sys     0m0.004s
```

not-using-ansi.c

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <infile> <outfile>\n", argv[0]);
        return 1;
    }

    int input = open(argv[1], O_RDONLY);
    if (input < 0) {
        perror(argv[1]);
        return 2;
    }

    int output = open(argv[2], O_CREAT|O_TRUNC|O_WRONLY, 0666);
    if (output < 0) {
        perror(argv[2]);
        close(input);
        return 3;
    }

    char buffer;
    int return_code = 0;

    for (;;) {
        ssize_t rlen = read(input, &buffer, sizeof(buffer));
        if (rlen < 0) {
            perror("Unable to read");
            return_code = 4;
            goto cleanup;
        }
        if (rlen == 0) {
            break;
        }

        ssize_t wlen = write(output, &buffer, sizeof(buffer));
        if (wlen != rlen) {
            perror("Unable to write");
            return_code = 5;
            goto cleanup;
        }
```

```
    }

cleanup:
    close(input);
    close(output);

    return return_code;
}
```

```
$ time ./not-using-ansi x y

real    0m3.217s
user    0m0.477s
sys     0m2.610s
```

Since both applications are reading and writing one character at a time, why is the performance so different?

The code on the left uses an implicit buffer built into the ANSI C library. This means that whenever the application calls `fread()` to get a single byte, it's really asking the library for the next byte *from the buffer*. On Linux, the default buffer size is 8192 bytes, but the value can change from one system to the next, and by the programmer at runtime. The default buffer size is defined as the `BUFSIZ` macro in the `stdio.h` header file. Applications can call `setvbuf()` to change buffer settings.

This means that the standard library will request 8192 bytes on the first request, and then use the contents of the buffer for the next 8191 requests. The code on the right must make 8192 system calls.

System calls are *slow* and application libraries should always cache or buffer information when possible. Some libraries do this automatically. For example, the `getpid()` function exists in the `glibc` standard library on Linux. The first time it's invoked, it uses the `getpid()` system call and then caches the information. it knows that the process ID of the application can't change, so there's no need to ever invoke that system call again.

Buffering Provided by the Operating System

If the ANSI C library is so good at buffering data, all applications should use it at all times, right?

No, it doesn't work like that. The reason the ANSI version of the previous code examples was so fast is because the code was performing I/O operations ***one byte at a time***! Applications that perform their I/O in larger chunks, in particular, chunks larger than `BUFSIZ`, may be even faster than the standard library!

This is particularly important in certain environments.

Imagine an application needed to read satellite image data from a file. Let's further say that the data is compressed.

There are different compression formats available. JPEG and PNG are two very common techniques. Satellite data isn't likely to use a lossy compression algorithm like JPEG, so for the sake of argument, assume it is PNG. Image data in a PNG file can be compressed multiple ways, but the big takeaway is... ***there's no way to calculate where in the file a particular row of image data begins***. This means an application must read the entire file into memory in order to process the image data.

The application could read this compressed file into memory in chunks of 8K, but the whole thing will need to be read for the pixel data to be available, so why not read the whole thing at once? If the file size is 256K, the program could allocate a buffer that large and use a single `read()` system call to read the whole thing. Is this going to be faster than the ANSI approach?

Yes, it is. Reading a 256K file using the ANSI library would require filling the 8K buffer 32 times. That means 32 system calls. The solution in the previous problem would take ***ONE*** system call.

Which is Better?

So, is it better to use ANSI layer buffering or the buffering provided by the operating system? Unfortunately, there's not a single answer to that question.

In situations where the data to be read or written is going to be manipulated in fairly small amounts, the ANSI buffering might be better. Multiple small reads or writes will be aggregated and reduce the number of system calls. But if the application will be doing *random access* I/O, then every time the program seeks to a new position, the contents of the ANSI buffer will be invalidated—the application loses the benefits of the buffer. In which case, it simply becomes an extra layer of overhead on top of the operating system.

The operating system approach is definitely best when the amount of data to be transferred exceeds the size of the ANSI buffer. Note that the ANSI buffer—location and size—can be configured via `setvbuf()` so this may be less of an advantage. At a minimum, it is best when using system calls for I/O to work with data transfer sizes that are a multiple of the physical block size of the filesystem (see `struct stat` and the `st_blksize` field from a previous chapter).

The end result is that the programmer must be familiar with **both** and be ready to use either one as the requirements dictate.

There is one other area of concern that will be covered in a later chapter. When an application calls `fork()`, the process' entire virtual addressing space is duplicated, including the `FILE` structure and the buffer that the structure refers to. So what happens if an application `fwrite()`'s data into the buffer, then calls `fork()`? There are now two processes, both of which have data in their respective buffers, and both of which will want to flush those buffers to the file when they fill up! The data in the buffer will be written twice! The effect of this can be summed up in one word: **Oops**.

WARNING

Do not mix the use of the ANSI functions and system calls on the same file descriptor! The ANSI functions must perform I/O on the file descriptor *eventually* and if the programmer is also making system calls on the same file descriptor, the result is undefined. The ANSI functions might be filling the buffer and leaving the seek offset in a location that the application doesn't expect, for example.

Keep in mind that if the code *does* need to access the data stream of a `FILE*` directly by using a file descriptor, the safest thing to do may be to `fflush()` the `FILE*` before making any system calls. Also note that `clearerr()` might be needed when signal handlers are mixed with file I/O (*signals* are an upcoming topic).

In choosing which technique to use, the programmer must draw on what they know of the execution environment and the use pattern that they expect. In some cases, the choice might be out of the programmer's hands. For example, stdin, stdout, and stderr are all `FILE*` objects. Using system calls with them would be ill-advised (and keep the above **WARNING** in mind).

Sometimes it can be helpful to rely on platform-specific defaults. For this reason, the `getconf` command line utility or `sysconf()` library function may be useful. The `getconf` utility is a cmmand line front-end to the `sysconf()` library function. They allow the user/programmer to query the execution environment and determine static values that can be used as defaults. For example, `getconf PAGESIZE` returns `4096` on Linux systems by invoking `sysconf(_SC_PAGESIZE)` and printing the return value.

Summary

- What is buffering and why use it?
- Different types of buffering
  - Application layer buffering done via functions and libraries
  - Operating system buffering done inside the kernel
- The markers that help determine when to use each one
  - Small data transfers should probably use the ANSI functions
  - Data transfers larger than the ANSI buffer size should probably use the operating system buffering
  - However, there are exceptions to both
- Avoid mixing the use of ANSI functions and system call I/O on the same file descriptor

Exercises

Exercise 1

Using profiling information, find the optimal “block size” to copy a file in terms of time. Test with files that smaller than the block size as well as many times larger.

Exercise 2

The `dd` command is a low-level copying command with a lot of flexibility. Why does it allow the user to set the block size for the copy?

Exercise 3

Determine the size of the buffer used by the C libary function `fread`.

Exercise 4 – `cp_block.c`

Implement a more efficient `cp` command using optimized buffering (choose between ANSI or operating system buffering). The copied file should have the same permissions as the source file.

- Allow the user to specify the size of the copy buffer.
  - Use the `-b` option to specify the buffer size. Perform appropriate error checking on the parameter value. For ANSI C buffering, the `setvbuf()` function (or similar) should be used.

Chapter 5. Directories and Links

Students will learn how to interact with POSIX directories, including reading the contents of directories and creating new directory entries without allocating new inodes.

- Directories and (hard) links
- Symbolic links
- Current working directory
- Operating relative to a directory (`openat()`, et al)
- Scanning directories
- Advanced APIs

Introduction

Directories are like the table of contents for this book: the table of contents serves as a way to find the information one wants by looking for an appropriate top-level topic, then looking inside that topic for the most appropriate subtopic, and so on. For example, the reader could scan the topics and decide that **Files** was the most appropriate. Then, they would look inside that topic for the most appropriate subtopic releated to their search. And so on, and so on. Eventually, they reach the specific topic they are searching for and begin reading.

While the concept of a directory is fairly universal at this point, ***how*** directories are managed is far from universal!

Each filesystem is responsible for storing data on the device, which means each filesystem implements directories however it wishes.

Requirements for directories include:

1. Support for a large number of entries (modern filesystems allow ***billions*** of entries).
2. Efficient sequential access (such as producing a directory listing using `ls`).
3. Fast lookup of an individual filename when then name is known (such as finding the inode number for `from_file` in the command, `cp from_file to_file`).

There have been many different implementations over the years, some better at some requirements and others better at other requirements. As the cost of storage space dropped and as processors became faster, the data structures used to store directories became more sophisticated. Commonly used implementations include:

- A simple sequence of bytes in a fixed record format.

  The old `s5` filesystem on AT&T Unix had directories that were always 16-byte records: two bytes for the inode number, and 14 bytes for the filename entry within the directory. Fixed record lengths made it easy to calculate buffering requirements, and records fit neatly into sectors allowing for efficient I/O.

- A stream of bytes in which each entry was variable length.

  The `ffs` filesystem on BSD Unix has variable length records, where one of the first fields inside the record was the total record size, allowing for quickly jumping to the beginning of the next record. The two most important fields were the two bytes for the inode number, and up to 255 bytes for the filename entry. The predominant feature of this implementation was the long filename support.

- A "virtual" file that uses hash tables to perform searches quickly.

  Hashes are not helpful for a sequential scan of a directory (such as the `ls` command would use), but

can be extremely fast when a filename is known and the directory must be searched. Otherwise, hashing directories were frequently derivatives of the `ffs` implementation. More disk space is consumed as a trade-off for faster file access.

All of the above techniques are in regards to the logical structure of the directory, not the physical storage on the media. A filesystem is likely to try very hard to ensure directories are allocated contiguous space as a means to make directory access as fast as possible. Directories that become very large may even change their logical structure!

The `ext4` filesystem on Linux does this. It starts out using the `ffs`-style implementation with variable length records, but when the number of entries grow large enough to consume more than a single 4K disk block, it switches to a unusual hash/B-tree structure that provides significantly better search performance. The Linux kernel documentation has more information.

This chapter will explore the ***use*** of directories more than the ***implementation***; however, some of the above descriptive information is exposed by the POSIX API for directory access.

Interestingly enough, the ANSI C language has no official standard for describing how to access a directory, even though access to files ***is*** defined (via subroutines like `fopen()`). This is not the only strange anomaly of the C language, but it **IS** an interesting trivia question for beginners! The reader might be wondering why it doesn't… The language cannot know what environment it's running in, and that environment may not include directories.

Before this chapter gets into the details, there is one other important aspect: hard links.

- A ***hard link*** is the term given to a filename which references an inode number of another file that already exists.

  For example, if the file `foo` has inode number `970` and the user creates `bar` with the same inode number, the two files are referred to as hard links. (The command to create a hard link is `ln`, and the system call is `link()`.)

- POSIX systems manage hard links by keeping a reference count in the inode. The inode (and data blocks) are identical when accessed via either of the filenames.

  If `foo` is removed, the reference count will be decremented, but because `bar` still refers to the inode, it will not reach zero, thus the inode and its data will not be freed. When `bar` is ***also*** removed, the reference count drops to zero and the inode and associated data blocks are released back into the pool of available disk space.

Symbolic Links

Hard links have one significant limitation—they cannot span filesystems.

Because a hard link is a filename with the same inode number as an existing file, it must be stored in the same filesystem—there's no way to refer to a different one with a single inode number.

A different kind of link, called a ***symbolic link***, can bypass that limitation. Instead of creating a filename that refers to a specific inode, a symbolic link ***contains the replacement path name*** as its data. As this is implemented inside the operating system, any and all applications can take advantage of it.

Some applications have special handling for symbolic links. The `bash` shell handles directory links specially when used with `cd`; when executing `cd symlink`, the shell keeps the resulting pathname as the value of `$PWD` so that `cd ..` can work properly. If it didn't do this, executing `cd ..` would go up one directory level from the ***physical*** directory location, which might be nowhere near the logical pathname of the symbolic link. Other command line utilities with special handling include `find`, `chmod`, `chown`, `tar`, and many other applications where the automatic forwarding may not be appropriate.

WARNING

If a symbolic link contains a relative pathname, it is relative to ***the directory that contains the symbolic link***.

```
/ (root)
|-- bin
    |-- bash
|-- usr
    |-- bin
        |-- sh -> ../../bin/bash
```

- If the `/usr/bin` directory contains the symbolic link `sh`, and
- the symbolic link refers to `../../bin/bash`, then
- any reference to `/usr/bin/sh` will be transparently redirected to `/bin/bash`.

Absolute pathnames are unaffected.

Symbolic links can also refer to directories, which hard links cannot do. This can trap applications that are naive about symbolic links.

- For example, suppose the filename `foo` is a symbolic link to a directory `bar`.
- Inside the `bar` directory is another symbolic link back to `foo`.
- What happens when a backup program tries to save the `bar` directory?

If it's not cognizant of symbolic links, it will save the contents of the `bar` directory by traversing the link over to `foo`, but then `foo` comes right back to `bar` again. (That backup is going to fill up the backup media and take a ***really long time*** to finish.)

Symbolic links are created using the `symlink()` system call and the contents of the link itself can be read using the `readlink()` system call. The most important operation on a symbolic link is the `lstat()` system call, because it provides a way to detect symbolic links.

NOTE

All applications that try to `open()` a symbolic link will be transparently pointed to the inode for the file that the link refers to. The ***only*** exception to this are applications specifically written to recognize symbolic links.

The most important component of writing such an application is querying the file type before opening a file. There are two ways of doing this: call `lstat()` and look at the `st_mode` field (as previously done with `stat()`), or attempt an operation that's only valid on symbolic links and look for an error return value (such as `lchown()`). The former is the preferred approach.

```
#include <unistd.h>

ssize_t readlink(const char *path, char *buf, size_t bufsize);
int symlink(const char *path1, const char *path2);
```

NOTE

Because symbolic links with short content strings are so common, most filesystems support storing the content of the link inside the inode itself, since other parts of the inode won't be needed. This means a symbolic link will consume an available inode but no data blocks. Some filesystems can store up to 60 characters of content in the inode!

Current Working Directory

When the operating system is asked to resolve an absolute pathname, it always starts at the top-level directory (a.k.a. the *root* directory). Relative pathnames start from the current directory.

Every POSIX user has likely typed in a pathname that uses `.` or `..` to reference a file or directory. Such references are *relative pathnames*. But any pathname that does not start with `/` is also relative. In the following table, the user **student** has just logged in and currently occupies their home directory. Each row shows equivalent references to the same location.

Absolute Pathname	Relative Pathname
`/home/student`	`.`
`/home/student/labfiles/oslabs/examples/fileIO/Makefile`	`labfiles/oslabs/examples/fileIO/Makefile`

That last example doesn't save nearly as much typing as the first one. But because the user can change their *current directory* any time they wish, making multiple references to files in the `fileIO` directory could become significantly shorter. This table shows what the pathname references look like after executing this command:

```
$ cd ~/labfiles/oslabs/examples/fileIO
```

Absolute Pathname	Relative Pathname
`/home/student/labfiles/oslabs/examples/fileIO/Makefile`	`./Makefile`
`/home/student/labfiles/oslabs/examples/fileIO/Makefile`	`Makefile`
`/home/student/labfiles/oslabs/examples/fileIO/Makefile`	`../fileIO/Makefile`

The operating system resolves relative pathnames by beginning the path traversal in the current directory. Because this is done by the operating system *for every system call that access pathnames*, it works across all applications.

Applications can change the current directory by using the `chdir()` system call. When the application terminates, the parent shell's current directory is not affected — each process has its own reference to its current directory. And what happens in the process, stays in the process.

If the `chdir()` system call is given a pathname that contains a symbolic link, the application follows the symbolic link (just as it always would) and the process is placed into the physical directory that the link refers to. This means that later using `chdir("..")` might not produce the expected result!

WARNING

In the following example, `/usr/bin` is a symbolic link to `../bin`. When the application called `chdir("/usr/bin")`, it ends up in the `/bin` directory. Later, invoking `chdir("..")` will make `/` the current directory, not `/usr`!

```
/ (root)
|-- bin
    |-- bash
|-- usr
    |-- bin -> ../bin
```

Operating on Files Relative to a Directory

In some situations, the application may be given a file descriptor that refers to a directory (instead of the actual pathname). In this case, the program can open files relative to the given file descriptor by using the `openat()` system call.

This is a pretty rare occurrence, although it *can* happen. One example would a parent-child coprocess/threaded situation in which the parent opens the directory and passes it along to the child as a file descriptor. Another one is where the ability to pass a file descriptor through a Unix domain socket is used and the file descriptor is a reference to an open directory. Neither of these approaches is discussed further in this module.]

```
#include <fcntl.h>

int openat(int dirfd, const char *pathname, int flags);
int openat(int dirfd, const char *pathname, int flags, mode_t mode);
```

As can be seen from the above function prototypes, the use of `openat()` is identical to the `open()` system call, with an extra parameter at the beginning that identifies the directory that relative pathnames will be based on. This means application code that must reference multiple files relative to a given directory can gain a performance edge by opening the directory first, then using relative pathnames for the files.

In the example code snippet, below, there are four relative pathnames that are under one directory, then an absolute pathname that will be resolved like an ordinary `open()` system call.

```
const char *list_of_files[] = {
    "Makefile",
    "file-io/copy-file.c",
    "buffering/using-ansi.c",
    "buffering/not-using-ansi.c",
    "/home/student/my_other_file",
};
int directory = open("/home/student/labfiles/oslabs/examples", O_RDONLY);
// ... proper error handling ...
for (int i=0; i < 5; i++) {
    int fd = openat(directory, list_of_files[i], O_RDONLY);
    // ... more proper error handling ...
}
```

Because the search for relative pathnames is relative to an open directory, the result should be better performance.

The biggest benefit of using `openat()` comes when an application is reading and processing multiple files from a directory, as described in the next section.

Reading Directories

It is fairly common for an application to want to process more than a single file. Maybe the application wants to copy multiple files from one directory to another, for example. In general, the simplest approach is to let the user specify the list of files on the command line, where the shell will allow the use the wildcards. The application will merely need to process `argv` one element at a time in a loop.

But there may be times when it is more convenient to give the application a directory name and expect the application to read the directory on its own. The most obvious example of such a situation is when the shell is not being used to execute the application. For example, a Python script might want to run the application, but because using a shell to do so opens up performance and security concerns, the Python script is going to invoke the application directly.

In situations where an application wants to open a directory and read the file names from it (such as the `ls` command does), the function to use is `opendir()`. Note that this is NOT a system call, but a library function. After the directory has been opened, there are other functions to read entries from the directory (`readdir()`) and close the directory when finished (`closedir()`).

The reasons behind this are ancient and arcane. The short version: since different filesystems store directories differently, there needed to be an abstract directory concept that the operating system provides to user space, but that concept had to easily account for changes over time. It was decided that doing so with a system call was a bad choice, as updating a library function is much easier than updating a system call.

Many modern Unix system no longer support the `read()` operation on opened directories Trying to read from a directory typically returns EOF immediately, or returns the EISDIR error code (as Linux does).

Because these are library functions and not system calls, they use the `DIR *` data type to represent the opened directory (similar to `FILE *` for opened files).

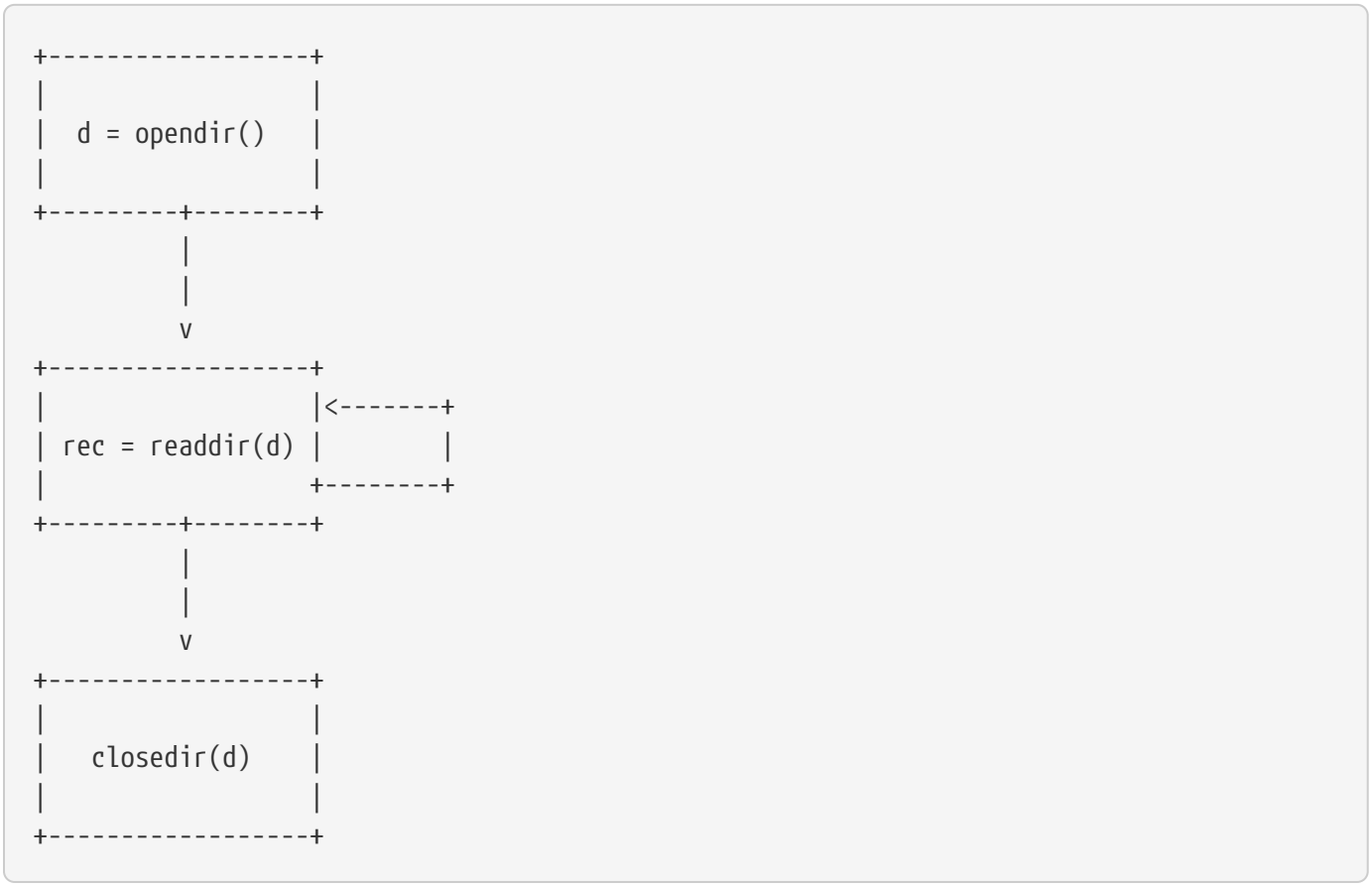
Function prototypes for directory handling functions

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);

DIR *fdopendir(int fd);
int dirfd(DIR *dirp);
```

Common Application Approach



The `readdir()` function returns a pointer to a static `struct dirent` as allocated by the library. This means subsequent calls may overwrite the contents. Applications that may need information from prior calls should copy the information to an appropriate buffer.

The commonly needed field is, of course, the `d_name` field, which identifies the name of the directory entry. Because the field is null-terminated, the simplest way to save the contents is to call the `strdup()` library function. Just be sure to `free()` the return value at some point!

Return value from `readdir()`

```
struct dirent {
    ino_t          d_ino;           // (POSIX) Inode number
    off_t          d_off;           // Not an offset; treat as opaque type
    unsigned short d_reclen;        // Length of this record
    unsigned char  d_type;          // Type of file; may be DT_UNKNOWN
    char           d_name[NAME_MAX]; // (POSIX) Null-terminated filename
};
```

An simplistic example implementation of the `ls` command is shown next. Note the use of the library functions to access the directory, and how the `struct dirent` is processed after reading each entry.

ls-inode.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <dirent.h>


int main(void)
{
    DIR *d = opendir(".");
    if (!d) {
        perror("Unable to open .");
        return 1;
    }

    while ((struct dirent *rec = readdir(d))) {
        // It's 64-bit, so could be up to 20 digits!
        printf("%10ld %s\n", rec->d_ino, rec->d_name);
    }
    if (errno != 0) {
        perror("Unable to read from .");
        closedir(d);
        return 2;
    }

    closedir(d);
}
```

Advanced APIs

There are other topics related to the handling of directories and links that are beyond the scope of this module. However, this section does mention them to give the student some hints to what else is possible. Be aware that some elements on this list are Linux-specific and may require a particular version of the Linux kernel.

A comparison of these three can be found here: https://lwn.net/Articles/604686/ Note that the linked article is circa 2014, but the content consists of discussion around the *how* and *why* of these three approaches and is still relevant today.

- The `fanotify` filesystem event notification system.

  The `fanotify` man page describes the newest API for monitoring filesystem events, useful for virus scanning and hierarchical storage management. Support is provided for *notification events* and *permission events*. The former are simply notifications and no response is required from the receiver. The latter expect a response indicating whether the given access should be allowed or disallowed.

  The event list is currently short, but includes *access*, *open*, *open for execution*, *attrib* (metadata change), *create*, *delete*, *moved*, *file modified*, and *close*. (There may be others.)

- The `inotify` filesystem event notification system.

  The `inotify` API is older than `fanotify`, which in the case of Linux, means it supports more events and has been in use (and thus tested) over a longer duration. However, it is being supplanted by `fanotify` for many use cases.

- The `dnotify` filesystem event notification system.

  Yes, there is a third—and older still—notification system. This was the first attempt on Linux and suffers from multiple design flaws; it has been deprecated and its use is highly discouraged.

- Efficient transport of file data.

  There are system calls that bypass user space and transfer data directly from one file descriptor to another *entirely within kernel space*. This is a huge performance win because the mode switches back and forth between kernel space and user space are eliminated.

  The system calls are `sendfile()`, `splice()`, `vmsplice()`, `copy_file_range()`, and `tee()` (the system call, not the command line utility). The last two have man pages with code samples.

- Treating an arbitrary directory as a process' root directory.

  The `chroot()` system call causes an arbitrary directory to be treated as `/` for that process, from the

point of the successful system call until the process terminates. As POSIX would say, appropriate privileges are required for the call to succeed. There are some obvious—and not so obvious—security issues surrounding `chroot()`. The command line `chroot` utility is a front-end to this system call.

- Managing allocation of data blocks to files.

  The `fallocate()` system call can preallocate disk space, thereby guaranteeing that access in the future will not be delayed by allocation requirements. The `ftruncate()` is similar, but changes the file size only and does not cause blocks to be allocated when a file grows.

Other Linux-isms will be mentioned in the appropriate chapter.

Summary

- Directories were defined
- Links are described and both types discussed
  - Hard links (directory entries that reference existing inodes)
  - Symbolic links (files that contain pathnames to other filesystem locations)
- The current working directory and how to change it
- How to access a file relative relative to a directory (`openat()`)
- Opening, reading, and closing directories
- Advanced APIs (filesystem notifications, in-memory file copies, etc)

Exercises

Exercise 1 — `cp_link.c`

Write a program that will copy a file. This program should work correctly on symbolic links. The default should be to copy the link itself. Use the `-L` option to indicate the symbolic link should be followed (meaning to copy the content, not the link itself).

Exercise 2 — `cp_recurse.c`

Modify the previous exercise so that if the source is a directory, it is copied recursively. Use the `-r` option to turn on directory copying (i.e., a "recursive" copy).

Start by just copying the top-level directory, then add recursion.

Chapter 6. Processes

Students will learn how POSIX systems manage work to be performed. Coverage will include CPU load, memory allocation, command line use, and associated metadata.

- Process IDs
- Process virtual memory map
- Command-line arguments
- The environment list
- Nonlocal gotos

Introduction

Every computer system exists to run programs. So far, this module has referred to these programs as "applications", but POSIX has formal definitions of some terms related to the execution of code.

An **Image** is the memory footprint of a program while it is executing.

Historically, it was used to refer to the application code while it was stored on disk, awaiting activation, because the compiler+linker would generate an "executable image" and save it as the application. Systems are more sophisticated now; the term is used generically to refer to an application and its memory footprint while it's executing.

A **Process** is the execution of the image, meaning that it represents the environment in which the application runs.

A **Thread** is a single execution unit within a process. Where the process represents the environment that the application executes within, the thread represents a virtual CPU running within that environment. It's possible for a single process to have dozens or hundreds of threads (or more), each executing code in different parts of memory (or even the same part).

The process environment includes the following items:

- All allocated virtual memory. This includes application code and data, but also shared library code and data, memory mapped files, the CPU stack, and environment variables and command line parameters. Such memory isn't necessarily ***physically*** allocated. (Systems that support hardware memory management typically interpret the attempt to access unallocated space as a program error and will generate a SIGSEGV.)
- The user and group credentials being used. Systems may define multiple categories of credentials, such as filesystem credentials, effective credentials, and real credentials.
- Filesystem reference information. This consists of essentially three things:
    1. the current directory,
    2. the root directory, and
    3. a ***namespace*** reference.

On Linux, a namespace is a collection of user-visible resources that the system administrator may want to isolate on a per-user basis. For example, a cloud services organization may want to allow multiple simultaneous users on their cloud, but not allow any of them to see the activities of the others via commands such as `ps` or `netstat`. The namespace makes it possible for each login to have its own list of processes, network sockets, mount points, and so on, separate from other logins.

- Metadata related to CPU management. This includes support for multiple threads of execution at

once, thread priorities, scheduler characteristics, and timeslice sizes.

- Interprocess communication controls. Shared memory, message queues, and semaphores are examples of ways that processes can communciate with each other dynamically at runtime. The operating system needs to keep track of which processes are using which resources so it knows when those resources can be reclaimed.
- Miscellaneous. This catch-all category includes timers (both coarse and high resolution), error handling controls (such as signals), device controls (which processes can read input from the keyboard and when), and management of resource limits (maximum CPU time, maximum amount of memory, maximum number of simultaneous open files).

Process IDs

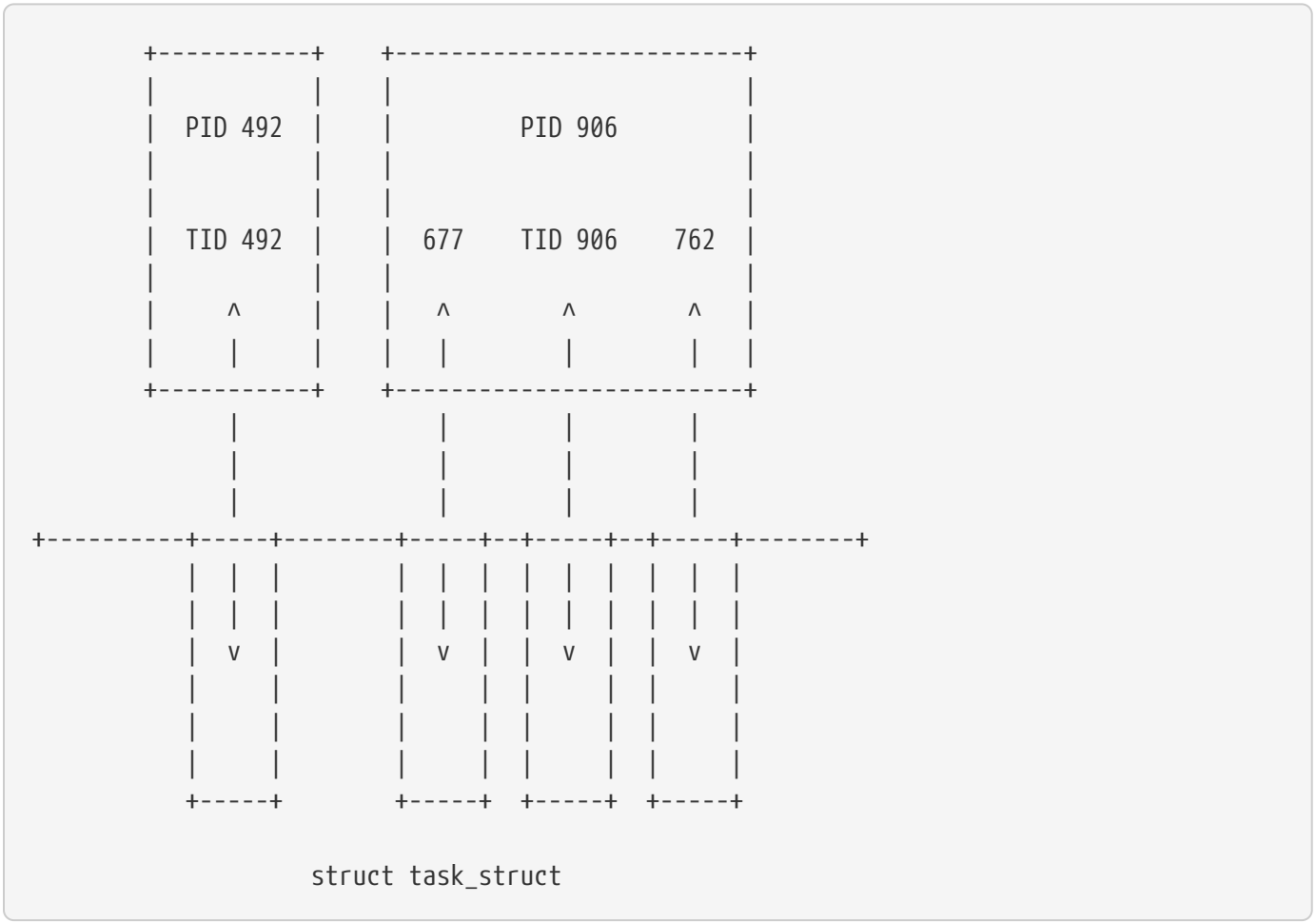
Users would have a difficult time managing their system if everything was done by name. Imagine having an application running four times (four ***processes***), each one a browser window connected to a different web server. How would the user know which one to stop by looking at the name?

POSIX operating systems therefore identify each process with a unique number that is assigned when the process is created. It is called the ***process ID***. Process IDs (or "PIDs") are guaranteed to be unique at the time they are allocated. They may be reused later, but will not overlap temporally. They are generally small integer numbers, but POSIX does not restrict the size; there are some vendors whose PIDs are 10 digits long (AIX)!

PIDs are important because they are reported by all of the various performance and security monitoring tools.

They are also frequently used by applications to create temporary files with unique names (if every PID is guaranteed unique at the time the process is created, temporary files that contain the PID should also be unique). In general, this works fine, but there are other situations that can make this unreliable. It is recommended programmers use the `mkstemp()` library function, or something like it, to create temporary files.

Here's how the Linux kernel views process IDs and thread IDs.



NOTE

The Linux kernel keeps track of processes internally using the `struct task_struct` data structure. Many of the environment attributes are pointers to other data structures, such as the credentials and the memory management fields.

In fact, every *thread* in a process has one of those `task_struct` structures created for it, then the attributes that are shared by all threads are simply pointers that point to a single component. Every `task_struct` is assigned a number. The first thread in a process (the one that executes `main()`), will have the PID assigned as its *thread ID* (or "TID"). All other threads in a process will be assigned their own TIDs, but they are allocated out of the same pool of numbers that PIDs come from.

What this means is if a Linux system is configured to allow **N** number of processes, that same pool is also used for threads, so the number **N** is really a limit on the total number of threads in the system as well.

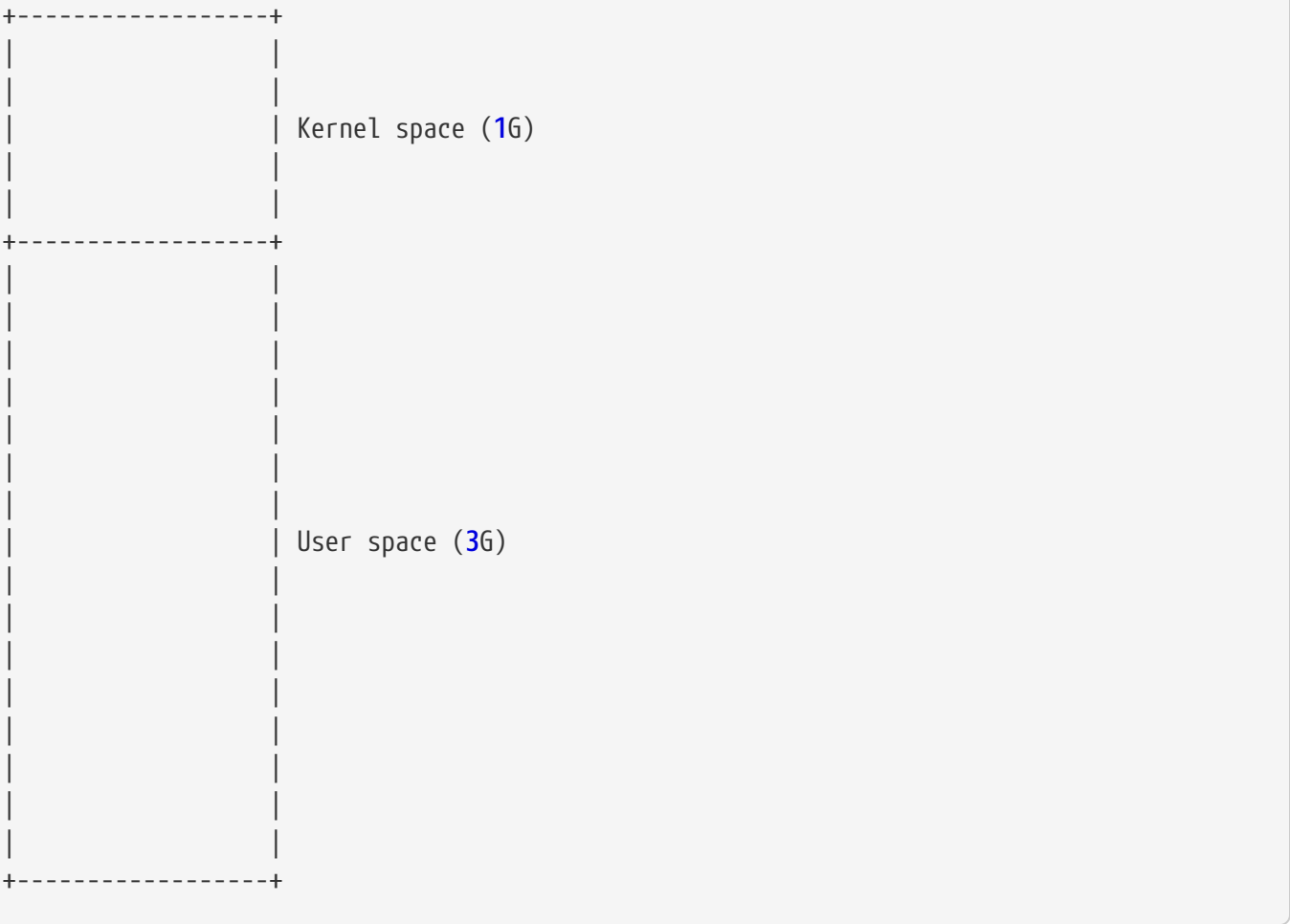
Longtime Unix users will recognize process IDs from commands such as `ps` (process status) and `top` (top CPU consumers). Those process IDs are also used in `kill` (to stop a running process) and `lsof` (to list the open files belonging to a process).

The Process Virtual Memory Map

Each process runs inside a private virtual addressing space. No other process can access the memory of another, unless their cooperating to a given purpose (for example, using shared memory to pass data back and forth).

How big is this virtual addressing space? First, the 3G/1G model:

The Linux 3G/1G Memory Model



Most system administrators will choose this model when a 32-bit system has less than 2.5G to 3G of physical memory. At those memory sizes, this is a fairly efficient model. The kernel reserves memory above the 1GB physical memory mark to user space applications, and memory below that point it can access directly. This works well for device drivers that must run inside the kernel, because the kernel generally doesn't need to access the user space of processes that are not currently running.

However, with 3G of physical memory or more, the overhead of using overlays for the kernel to access higher physical memory means performance starts to suffer. At this point, the 4G/4G model may be better:

The Linux 4G/4G Memory Model

In this model, both the kernel and the user space application have full access to the entire 32-bit range of addresses. Because the kernel can access the entire 4G without overlays, the overhead of memory access is reduced. However, every interrupt that occurs will require the addressing space to be changed and, unfortunately, there is a performance penalty for this. Whether this model actually ended up being any faster depended on the overall load of the machine, because once the model is selected, it's used for *every* process!

There is one other model, the `4G/64G` model. It is only mentioned as a historical artifact, since it was an interim solution between Intel's 32-bit architecture and its not-yet-built 64-bit technology. As might be expected from the name, the user space had 4G of memory addressing available, but the kernel had access to 64G of physical memory by manipulating special virtual memory capabilities of the CPU chip that added another 4 address lines to the physical memory bus.

And now, on to the memory model of a modern 64-bit system.

The Linux 64-bit Memory Model

This memory model, originally created by AMD and named **AMD64**, has been adopted by the rest of the industry (including Intel) and is now named **x86_64**. For a time, Intel had pushed their own 64-bit memory model (Itanium) that broke the addressing space up into eight regions, then assigned each region a particular use. Given that each region had 2EB of addressing space (an exabyte is a million terabytes), the size of each region wasn't an issue, but the segmentation of the addressing space brought back the horrors of the original 8088 segmented architecture and it was widely panned.

The **x86_64** memory model has the following benefits:

1. It is huge!

   If a programmer can't fit their code and data within a 16EB addressing space, they need to go back to the drawing board.

2. It is easily extendable.

   Because the kernel space starts at the top and user space starts at the bottom, they can each grow towards each other as systems require more virtual address space. Many times, hardware only supports **N** number of bits in the address. When new hardware comes out that supports more, the top and bottom regions can grow by just adding those bits to the address. On Linux, the contents of

`/proc/cpuinfo` shows the number of bits for virtual and physical addresses.

3. There are multiple memory management options.

   For a system that has terabytes of memory, it is no longer efficient to manage memory in 4K chunks. This scheme has no preconceived limitations on page size or physical memory addressing. The memory model allows up to 256TB of memory in the system. However, there aren't very many systems actually capable of that. So any application that tries to actually ***use*** 128TB of space is going to find much of its memory swapped out to disk. Because the kernel has to have memory reserved for recording which 4K page is swapped to which 4K disk block, there's associated overhead in the kernel. All of which means that an application that wants to allocate thousands of times more virtual memory than physically exists in the machine is going to perform terrible no matter what.

This course will focus on the 64-bit memory model.

This memory model provides an expansive range of virtual address for loading application code and data as well as shared library code and data. Here's an example of the address map for the Bash shell:

Example Output of `cat /proc/$$/maps`

```
561b86420000-561b8644d000 r--p 00000000 103:02 5113494                   /bin/bash
561b8644d000-561b864fe000 r-xp 0002d000 103:02 5113494                   /bin/bash
561b864fe000-561b86535000 r--p 000de000 103:02 5113494                   /bin/bash
561b86535000-561b86539000 r--p 00114000 103:02 5113494                   /bin/bash
561b86539000-561b86542000 rw-p 00118000 103:02 5113494                   /bin/bash
561b86542000-561b8654c000 rw-p 00000000 00:00 0
561b86598000-561b8675a000 rw-p 00000000 00:00 0                          [heap]
7fd33207f000-7fd332082000 r--p 00000000 103:02 11272672                  /lib/x86_64-
linux-gnu/libnss_files-2.32.so
7fd332082000-7fd33208a000 r-xp 00003000 103:02 11272672                  /lib/x86_64-
linux-gnu/libnss_files-2.32.so
7fd33208a000-7fd33208c000 r--p 0000b000 103:02 11272672                  /lib/x86_64-
linux-gnu/libnss_files-2.32.so
7fd33208c000-7fd33208d000 r--p 0000c000 103:02 11272672                  /lib/x86_64-
linux-gnu/libnss_files-2.32.so
7fd33208d000-7fd33208e000 rw-p 0000d000 103:02 11272672                  /lib/x86_64-
linux-gnu/libnss_files-2.32.so
7fd33208e000-7fd332094000 rw-p 00000000 00:00 0
7fd3320aa000-7fd33261b000 r--p 00000000 103:02 3408710
/usr/lib/locale/locale-archive
7fd33261b000-7fd33261e000 rw-p 00000000 00:00 0
7fd33261e000-7fd332644000 r--p 00000000 103:02 11272646                  /lib/x86_64-
linux-gnu/libc-2.32.so
7fd332644000-7fd3327b1000 r-xp 00026000 103:02 11272646                  /lib/x86_64-
linux-gnu/libc-2.32.so
```

```
7fd3327b1000-7fd3327fd000 r--p 00193000 103:02 11272646                  /lib/x86_64-
linux-gnu/libc-2.32.so
7fd3327fd000-7fd3327fe000 ---p 001df000 103:02 11272646                  /lib/x86_64-
linux-gnu/libc-2.32.so
7fd3327fe000-7fd332801000 r--p 001df000 103:02 11272646                  /lib/x86_64-
linux-gnu/libc-2.32.so
7fd332801000-7fd332804000 rw-p 001e2000 103:02 11272646                  /lib/x86_64-
linux-gnu/libc-2.32.so
7fd332804000-7fd332808000 rw-p 00000000 00:00 0
7fd332808000-7fd332809000 r--p 00000000 103:02 11272647                  /lib/x86_64-
linux-gnu/libdl-2.32.so
7fd332809000-7fd33280b000 r-xp 00001000 103:02 11272647                  /lib/x86_64-
linux-gnu/libdl-2.32.so
7fd33280b000-7fd33280c000 r--p 00003000 103:02 11272647                  /lib/x86_64-
linux-gnu/libdl-2.32.so
7fd33280c000-7fd33280d000 r--p 00003000 103:02 11272647                  /lib/x86_64-
linux-gnu/libdl-2.32.so
7fd33280d000-7fd33280e000 rw-p 00004000 103:02 11272647                  /lib/x86_64-
linux-gnu/libdl-2.32.so
7fd33280e000-7fd33281c000 r--p 00000000 103:02 11272474                  /lib/x86_64-
linux-gnu/libtinfo.so.6.2
7fd33281c000-7fd33282b000 r-xp 0000e000 103:02 11272474                  /lib/x86_64-
linux-gnu/libtinfo.so.6.2
7fd33282b000-7fd332839000 r--p 0001d000 103:02 11272474                  /lib/x86_64-
linux-gnu/libtinfo.so.6.2
7fd332839000-7fd33283d000 r--p 0002a000 103:02 11272474                  /lib/x86_64-
linux-gnu/libtinfo.so.6.2
7fd33283d000-7fd33283e000 rw-p 0002e000 103:02 11272474                  /lib/x86_64-
linux-gnu/libtinfo.so.6.2
7fd33283e000-7fd332840000 rw-p 00000000 00:00 0
7fd33284f000-7fd332856000 r--s 00000000 103:02 3673875                   /usr/lib/x86_64-
linux-gnu/gconv/gconv-modules.cache
7fd332856000-7fd332857000 r--p 00000000 103:02 11272532                  /lib/x86_64-
linux-gnu/ld-2.32.so
7fd332857000-7fd33287b000 r-xp 00001000 103:02 11272532                  /lib/x86_64-
linux-gnu/ld-2.32.so
7fd33287b000-7fd332884000 r--p 00025000 103:02 11272532                  /lib/x86_64-
linux-gnu/ld-2.32.so
7fd332884000-7fd332885000 r--p 0002d000 103:02 11272532                  /lib/x86_64-
linux-gnu/ld-2.32.so
7fd332885000-7fd332887000 rw-p 0002e000 103:02 11272532                  /lib/x86_64-
linux-gnu/ld-2.32.so
7ffce44dc000-7ffce44fd000 rw-p 00000000 00:00 0                          [stack]
7ffce4509000-7ffce450d000 r--p 00000000 00:00 0                          [vvar]
7ffce450d000-7ffce450f000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

The important columns are the first two, which define the range a virtual addresses, and the last column which identifies the object that occupies that memory range.

1. The first field is **start:end** memory addresses (inclusive and exclusive).
2. The permissions that apply to the address range.
3. The offset within the object used to initialize the memory.

   Regions can overlap. Suppose a region needs the first 7K of a file; it'll be represented as 8K in the `maps` listing because pages are multiples of 4K. But that means the next region (that must start at 8K because of 4K pages) will have an offset of 4K into the file and the first 3K will be unused within the region. This happens because the compiler and linker will pack regions that must be `r-x`, `rw-`, and `r--` next to each in the disk file, but memory protection can only happen on page boundaries.

   It's easier to understand by picking an application and running through the numbers by hand. For example, `/usr/bin/size` has `26,469` bytes of executable code, `1,464` bytes of initialized static data, and `1,336` bytes of uninitialized static data. That means the `text` portion takes 7 pages of 4K each (28K) and becomes the first region. The `data` region begins at the 28K mark, and starts at offset 24K (because the first `data` byte is at `text+1`, or `26,470` which is `1,894` bytes *past* the 28K mark) so the first `data` byte is at (28K + 1894 bytes) and it extends to the end of the page (because `1,464` will fit into the same page). Other sections are similarly allocated: each region that is protected differently must begin on a page boundary both in memory and in the disk file.

4. The **major:minor** device number that holds the object. Correlate major device numbers against `/proc/devices` to find which driver it is, or `grep` through the output of `ls -l /dev` to find a specific node.
5. An identifier for the object (an inode number for filesystem objects).
6. A human readable representation of the object.

   Object names with brackets around them are dynamic:

   - `[heap]` is the application and library heap (used by `malloc()`)
   - `[stack]` is the area reserved for stack usage (for the default thread)
   - `[vsyscall]` was an optimization approach for system calls; replaced by `vdso`
   - `[vdso]` is the *virtual dynamic shared object* page; used for faster system calls
   - `[vvar]` is the data area used by the `vdso` page

Passing Data from Parent to Child

Each process has its own virtual addressing space. This means it's impossible for one application to modify the memory that belongs to another process, thus it's impossible for one process to cause another to crash or execute illegal instructions (barring a bug or logic flaw within the kernel).

However, having applications pass data back and forth at runtime is one of the key tenets for writing reusable, modular code. Programmers would like to write small applications that each do a tiny piece of the job, then le the user chain them together to produce a final result. This approach allows the user to put the smaller pieces together in whatever shape they want without writing any new code.

More sophisticated approaches for this will be discussed later in the module. For now, this section will look at passing data from the parent process to the newly created child process.

Creating the Child Process

First, process creation is performed by calling the `fork()` system call. A better name for this system call nowadays would be `clone`, because people know what a clone is—it's an exact duplicate of the original, all the way down to the DNA. And that's what `fork()` produces: a duplicate process, all the way down to individual bytes in memory. Here is how `fork()` treats the various pieces of the process environment:

- Cloned:
  - Current directory / current root directory
  - Credentials
  - Virtual memory
  - Open files
  - Resource limits
- Initialized:
  - Process ID
  - Accounting fields (CPU time, amount of I/O, etc)
  - No memory locks or `fcntl()` file locks are cloned (`flock()` locks *are*)
  - Pending signal mask is cleared
  - No timers are cloned
  - No asynchronous I/O operations are cloned

Looking at the above list, how can the parent and child communicate? There are multiple ways:

1. Open a pipe. The parent creates the pipe and writes to one end, while the child inherits the pipe and

reads from the other end.

2. Use shared memory. A region of memory can be shared (see the `mmap()` system call) in a way that allows both processes to access it.
3. Variations on the above:
4. * Opening a file and allowing the child to inherit the file descriptor
5. * Mapping a file into the virtual addressing space and allowing the child to access it

Running an Application Inside the Child

Once the child process has been created, it can replace its own virtual memory with the image of a different application. This is done using the `execve()` system call.

```
#include <unistd.h>

int execve(const char *pathname, char *const argv[], char *const envp[]);
```

This system call is given the name of the application to execute and a list of command line arguments and environment variables:

1. It causes the operating system to replace the virtual memory of the current process with the image specified as the first parameter, `pathname`.
2. The operating system clears the virtual address space of all existing mappings and loads new ones from the `pathname` executable.
3. It creates a dynamic stack region and copies the contents of `argv` into that memory, along with the contents of `envp`. (Note that both arrays must be terminated with a sentinel null pointer.)
4. The actual stack is set just below that data and a startup stub is invoked, passing it the locations of `argv` and `envp`.
5. The startup routine identifies which shared libraries are required and begins mapping them into memory.
6. After all libraries are mapped into memory, the `main()` function is invoked and is passed the number of elements in `argv`, along with the locations of where `argv` and `envp` were copied.
7. When `main()` returns, some ANSI C housecleaning is done and the application terminates. If `main()` falls off the end of a multithreaded application, the entire process terminates and any executing threads are summarily terminated.

   Note that the libraries are not explicitly unmapped, although the ANSI C cleanup functions do take care of language-level resources (like closing `stdin`, `stdout`, and `stderr`).

Nonlocal GOTO Statements

This is an unusual topic which isn't used frequently, but is very convenient in certain circumstances.

A regular `goto` statement merely jumps from one location in the current function to another location (identified via what the C language calls a *label*, so these are sometimes referred to as *labeled gotos*.)

A *nonlocal goto* can jump out of one function and into another!

This is clearly going to require some setup work in advance. There's no way that one function can jump into another without messing up the stack frames and having unpredictable contents in the CPU registers. So nonlocal gotos require that the function that will act as the destination for the jump execute a specific function, `setjmp()`, which can record some of the important state information for later use. When another function wants to jump back to that point, it calls `longjmp()`.

```
#include <setjmp.h>

int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);

// _POSIX_C_SOURCE
int sigsetjmp(sigjmp_buf env, int savesigs);
void siglongjmp(sigjmp_buf env, int val);
```

Nonlocal gotos are primarily useful in two situations:

1. When a deeply nested function call detects an error and the proper handling requires returning to a "known good" location in the application.
2. When a signal handler wishes to bypass normal signal handling (such as having application code periodically check whether a signal has occurred) by jumping straight to the "known good" location.

WARNING	These operations can significantly decrease readability of the code and should be avoided when possible!

While they're called "nonlocal gotos", they can only jump to a location that has previously executed the `setjmp()` call. Also, they cannot jump from one *process* to another (the memory space of a process belongs to that process and cannot be seen or interacted with by any other application).

There are a variety of restrictions on what nonlocal gotos can accomplish, due primarily to compiler optimizations. For example, the value of automatic variables are unspecified after a `longjmp()` if all of the following are true:

1. they are local to the function that made the `setjmp()` call;
2. their values are changed between the calls to `setjmp()` and `longjmp()`; and
3. they are not declared `volatile`.

Example of `setjmp()` and `longjmp()`

```c
#include <stdio.h>
#include <setjmp.h>
#include <unistd.h>

jmp_buf original_postition;

void level5()
{
    puts("Calling 'longjmp()'...");
    longjmp(original_postition, 1);
}

void level4()
{
    puts("About to call 'level5()'...");
    level5();
}

void level3()
{
    puts("About to call 'level4()'...");
    level4();
}

void level2()
{
    puts("About to call 'level3()'...");
    level3();
}

int main(void)
{
    puts("Calling 'setjmp()' to get everything set up...");
    if (setjmp(original_postition)) {
        puts("setjmp() returned nonzero");
        puts("Returned via 'longjmp()'!");
    } else {
        puts("setjmp() returned 0, executing normally");
        puts("About to call 'level2()'...");
        level2();
    }

    puts("Exiting program");
}
```

Summary

This chapter has discussed how processes are managed on a POSIX operating system.

- Process IDs and thread IDs
- Process virtual memory map
- How command line parameters and environment are passed from parent to child
- How and why nonlocal gotos are used

Exercises

Exercise 1 – `mip.c`

Write a program that prints an address from each memory section of itself:

- Stack
- Heap
- Data
- Code

Exercise 2 – `env.c`

Implement the equivalent of the Unix `env` command, but with changes:

1. Allow environment variables to be set using `VAR=value` syntax (just like the existing `env` command);
2. Any parameters which are not `name=value` pairs are collected into a command line for execution; and
3. All variables recognized in step 1, above, are passed to the new command.
4. Environment variables created should only exist for the command executed and should not appear in the environment when the application terminates.
5. Wait for the provided command to finish execution and report its exit code, then terminate with the same exit code.

The following command lines should work properly, including failure messages when appropriate (copy any program to the name `./some` for testing purposes):

```
./env FOO=1 BAR=2 ./some silly command  # Sets FOO and BAR, then executes ./some
./env ./some FOO=1 silly BAR=2 command  # (same)
./env some silly command PATH="$PATH:." # Notice it's not "./some" anymore!
```

Chapter 7. Process Credentials

Students will learn the components of POSIX C2 security, which includes user ids, process credentials, and file permissions.

- Users and groups
- Process credentials
- Retrieving process credentials

Introduction

Security is always an important aspect of any operating system and the applications that run on that operating system. This chapter will explore the security options available in a POSIX operating system.

A modern POSIX operating system has two primary permission mechanisms. The first is ***standard file permissions*** and the second is ***access control lists***.

Standard File Permissions

Standard file permissions provide two categories of users who may require access to a filesystem resource. The first category is the owner of the file (of which there can be only one), and the second category is the group assigned to the file. These categories are typically abbreviated with **u** (for ***user***, a.k.a. owner) and **g** (the ***group***).

Within each category the same types of permissions apply. There are three permissions: **read**, **write**, and **execute**. These three permissions are typically abbreviated with **r**, **w**, and **x**.

Each username is assigned a numeric value (called the ***user ID***) by the system administrator when the account is created. Similarly, groups are created and users are allocated to groups by the system administrator.

When a process attempts to access a file system object, the operating system verifies the request against the file permissions. This verification is always performed as described below:

1. If the user ID of the file matches the user ID of the process, the ***user*** category permissions are checked and the verification is complete.
2. If the group ID of the file matches one of the group IDs of the process (a process can belong to multiple groups at once, called ***secondary groups***), the ***group*** category permissions are checked and the verification is complete.
3. The ***other*** category of permissions are checked and the verification is complete.

(The list of secondary groups is initialized when a user logs in. Most systems have an upper limit on the number of concurrent secondary groups that a user may belong to, typically in the range of 8 to 32 on older systems and typically 1024 or more on modern systems.)

Access Control Lists

Access control lists (*ACLs*) are an extension of the above mechanism. The same categories apply (*user* and *group*), but file system objects can list multiple user IDs and multiple group IDs and the permissions that apply for them. This makes ACLs much more flexible than standard file permissions, but increases the management complexity and makes it more difficult to isolate which permissions are causing a request to fail. Backup standards like `tar` and `cpio` don't consider ACLs, meaning that those commands cannot be used to backup a filesystem that contains ACLs without losing information during restoration.

While standard file permissions can be stored in the inode for the file system object, ACLs can be of unlimited specificity, which makes them arbitrarily sized and unable to be stored inside the inode. Instead, most POSIX systems allocate a second inode and additional data blocks to hold the contents of the ACLs where they are frequently stored as ordinary text. This increases the disk space requirements for a file with ACLs as well as the time required to access the file.

The ACL data blocks must be read, and then their text scanned to verify the permissions That means both extra seeks and extra reads, followed by tokenizing and parsing of the content. Linux limits the size of the ACL text to 64KB (see the man page for `xattr(7)`).

Exploration of the details of the ACL implementation are beyond the scope of this course. For details on how Linux supports ACLs, see the `setfacl` and `getfacl` commands for more information. (The actual commands may be different on other POSIX systems.)

Manipulation of Process Credentials

There are only two situations where an application may need to manage its credentials:

- When it is executed by a user with root authority and wants to voluntarily relinquish that authority to limit exposure to security vulnerabilities.
- When the executable has a *set-id* permission turned on and it only needs those permissions during a limited code path. Executables with the *setuid* (*setgid*) bit turned on will start with their effective and saved user (group) ID set to the user (group) specified in the inode.

For example:

- The `sshd` daemon that needs to open a privileged socket port requires root authority for that operation, but once the operation is complete that authority can be relinquished.

  It is invoked by a process with root authority, so it inherits that authority. But it gives up those privileges once it has opened port 22. Later, it needs to regain that authority to accomplish login tasks.

- The `passwd` command must modify secure files under `/etc`, but it only needs that authority for a short code path.

  It is invoked by ordinary users but gains root authority because the executable is *setuid* and owned by `root`. It gives up those privileges until it needs to modify the files under `/etc`. At that time, it regains root authority to make the changes, then gives it up again.

There are two sets of system calls that POSIX applications use to manage these values:

1. The `getuid()` and `setuid()` system calls access the *real* user ID.
2. The `geteuid()` and `seteuid()` system calls access the *effective* user ID.

Here are the rules for how these values can be changed:

- An unprivileged process can set the real and effective user (group) ID to any of the values stored in the real, effective, or saved fields.
- A privileged process (one with user ID `0` stored in its real or saved user ID) can change its real and effective user (group) ID to any arbitrary value.

As covered in the previous chapter, when a process calls `fork()`, the user and group credentials are copied into the new process. If the child invokes `execve()` and the executable has the *setuid* or *setgid* bits turned on, the credentials can change if the `execve()` is successful.

A process can have more authority than the user and group ids might imply through a technique called *capabilities*.

- Capabilities take all of the various authorities that `root` normally has on a POSIX system and breaks them down into separate categories.
- These categories can then be assigned to users (or applications), and when processes are created by the user (or an application is executed via `execve()`) those authorities are granted.
- There is a later chapter on applications with elevated privileges where capabilities will be discussed in more detail.

Summary

Students have learned the components of POSIX C2 security, which includes user ids, process credentials, and file permissions. These are the underpinnings of all security mechanisms on POSIX systems.

- Users and groups
- Overview of access control lists
- Manipulating process credentials

Chapter 8. Signals: Introduction

Students will learn how POSIX systems handle synchronous and asynchronous events, including the ability to ignore, block, or suspend such handling.

- Signal dispositions
- Signal handlers
- Useful signal-related functions
- Signal sets, the signal mask, and pending signals

Introduction

Signals are an asynchronous notification system implemented on POSIX operating systems.

Signals allow one process to notify another of some event. There are a limited number of signals available, so applications must choose between signals with predefined meanings and signals that are left by the operating system as user-defined.

- Predefined signals will generally terminate the target process, and in some cases will cause a core dump to be generated (if enabled).
  - The operating system may send these types of signals as the result of something the process did (such as a floating point exception, or an attempt to execute an illegal instruction), or
  - these signals may be the result of an action taken by the user (such as `Ctrl+C` to terminate a process, or killing processes associated with a terminal by logging out of that terminal session).
- User-defined signals will also terminate the target process, but because they are user-defined, they are never sent by the operating system itself.

Non-POSIX systems typically also include some type of asynchronous notification system and the C library on those platforms will often emulate signals using whatever support the operating system provides. This allows signal handling code to be somewhat portable, although specific signals may or may not be generated on a given platform. For example, Windows will convert `Ctrl+C` into `SIGINT` just as POSIX systems do, but Windows does not have an equivalent to the `SIGSYS` signal.

Default Signal Handling

The following chart shows the defined POSIX signals, their default disposition/action, and a brief description. Some of the signals are required by the C standard.

Name	Action	Description
SIGABRT	create core image	Abort process (formerly SIGIOT). C Standard.
SIGALRM	terminate	Real-time timer expired
SIGBUS	create core image	Bus error
SIGCHLD	ignore	Child process stopped or terminated
SIGCONT	continue	Continue (if stopped; otherwise, ignored)
SIGFPE	create core image	Floating-point exception. C Standard.
SIGHUP	terminate	Terminal line hangup
SIGILL	create core image	Illegal instruction. C Standard.
SIGINT	terminate	Interrupt process (Ctrl+C). C Standard.
SIGKILL	terminate	Kill process (cannot be caught)
SIGPIPE	terminate	Write on a pipe with no reader
SIGPOLL	terminate	Pollable event
SIGPROF	terminate	Profiling timer expired
SIGQUIT	create core image	Quit process (Ctrl+\\)
SIGSEGV	create core image	Segmentation violation. C Standard.
SIGSTOP	stop	Stop process
SIGSYS	terminate	Illegal system call (a.k.a. SIGUNUSED)
SIGTERM	terminate	Software termination signal. C Standard.
SIGTSTP	stop	Stop process (terminal initiated; usually Ctrl+Z)
SIGTTIN	stop	Stop on terminal input for bkgd process
SIGTTOU	stop	Stop on terminal output for bkgd process
SIGTRAP	create core image	Trace trap
SIGURG	ignore	Urgent condition on socket
SIGUSR1	create core image	User-defined signal
SIGUSR2	create core image	User-defined signal
SIGVTALRM	terminate	Virtual alarm clock

Name	Action	Description
SIGXCPU	create core image	Exceeded CPU time limit
SIGXFSZ	create core image	Exceeded file size limit

Even within this required group, there are some may not be implemented on specific platforms.

The numeric value of the signal will vary from platform to platform. SIGBUS is signal 7 on x86 and ARM, but is signal 10 on others.

WARNING

Software must never assume that a particular signal name maps to a particular signal number. Source code should always use the names so that code compiled on another platform will still build properly (even though such signals may not work because the signal itself is unimplemented).

Other signals beyond the required set also exist. Shown in this chart are those that are implemented on Linux, but this is not an exhaustive list for other platforms. Always check the documentation for the particular operating system and version being used to determine the specifics.

Name	Action	Description
SIGEMT	create core image	Emulate instruction executed
SIGIO	terminate	I/O now possible (a.k.a. SIGPOLL)
SIGPWR	terminate	Power failure
SIGSTKFLT	terminate	Stack fault on coprocessor (unused)
SIGWINCH	ignore	Window resize

There are additional signals on Linux that go all the way up to 63. However, those are *real-time signals* and are covered later.

User-defined Signal handling

This chapter has covered what the system does by default when a signal arrives. But how can an application modify those actions?

While the default action for `SIGINT` (`Ctrl+C`) is to terminate the process, an application may have other actions it wants to take instead of (or in addition to) the default action. This is referred to as *handling the signal* or *catching the signal*.

The following system call controls how signals are delivered.

```
#include <signal.h>

// Do NOT assign to both sa_handler and sa_sigaction, as some platforms use a
// union to store both.
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    // ...
};

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

The `sigaction()` system call takes three parameters.

1. The first is the signal number to be affected, specified using the signal name.

WARNING	Note that `SIGKILL` and `SIGSTOP` cannot be caught or ignored. Specifying either one in a call to `sigaction()` to change their handling will be ignored.

2. The second parameter, if non-`NULL`, is a pointer to a `sigaction` structure that identifies how to handle the signal:

   a. `sa_handler` (and `sa_sigaction` when `sa_flags` contains `SA_SIGINFO`)

   - **Take the default action**: set to `SIG_DFL`
   - **Ignore it**: set to `SIG_IGN`

     The signal will never be added to the queue of signals to deliver for the process.

   - **Catch it**: set to the address of a function

The `sigaction` structure contains the address of an application function that will be invoked when the signal is delivered.

b. `sa_mask`

- This mask specifies signals that will be automatically blocked before execution of the signal handler begins.
- The original signal mask will be restored when the signal handler returns.
- By default, the signal being caught is automatically added to the list.

c. `sa_flags`

- In most situations, this field is likely zero.
- However, if the programmer wants the operating system to automatically restart system calls made on long duration devices, the `SA_RESTART` value should be added here. (More on this in the next chapter.)
- Also, `SA_SIGINFO` can be used to switch to a different signal handler function signature that includes the `siginfo_t` parameter.

3. The third parameter, if non-NULL, is a pointer to a structure that will be filled in with the current configuration for the given signal. (This allows an application to restore those settings at a future point, if it wishes.)

Here is some example code that shows examples of all four `sa_handler` possibilities.

sighandling.c

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

void sigint_handler(int sig);
void establish_handler(int sig, void (*func)(int));

int main(void)
{
    establish_handler(SIGINT, sigint_handler);

    establish_handler(SIGQUIT, SIG_IGN);

    establish_handler(SIGTERM, SIG_DFL);

    puts("Send signals SIGINT, SIGQUIT, or SIGTERM");
    puts("to this program. Suspend with Ctrl+Z, then");
    puts("`kill -(INT|QUIT|TERM) %1` to send the signal");

    int count = 0;
    for (;;) {
        sleep(1);
        printf("\r%d ", ++count);
        fflush(stdout);
    }
}

void establish_handler(int sig, void (*func)(int))
{
    struct sigaction sa = { .sa_handler = func };
    if (sigaction(sig, &sa, NULL) == -1) {
        perror("sigaction: set");
        exit(1);
    }
}

void sigint_handler(int sig)
{
    // unused argument
    (void)sig;
    const char msg[] = "SIGINT occurred!\n";
    write(1, msg, sizeof(msg));
}
```

The following chapter has more information on how to design and write signal handlers.

This chapter has described how to handle a signal such that a particular function is invoked, but how are signals sent? There is a system call that handles sending signals: `kill()`. For legacy reasons, it is called `kill` and not `sigsend`—early implementations did not allow signals to be caught, they simply killed the target process!

```
#include <signal.h>

int kill(pid_t pid, int sig);
```

There are some options for specifying the process `pid`. In all cases, the sending process must have appropriate privileges to deliver a signal to the target(s) or an error will be returned. When multiple targets are possible, an error for one target does not stop other targets from being signaled. In other words, no error is returned if *any* target was successful signaled when there are multiple targets.

- If `pid > 0`, the signal is sent to the target process id.
- If `pid == 0`, the signal is sent to all processes with the same process group id as the sender.
- If `pid == -1`, the signal is sent to all processes with the same user id as the sender, excluding the sender. If the sender has superuser privileges, the signal is sent to all processes excluding system processes and the sender.
- If `pid < -1`, the signal is sent to all processes with a process group id equal to the absolute value of `pid`.

The `sig` parameter is any signal valid on the system, plus signal zero. A signal number of zero is a special placeholder that can be used to check whether a given process exists without actually delivering a signal. This check also isn't dependent on some of the security requirements imposed by the various options for `pid`, above.

Legacy API

While the legacy API may appear in older source code, it should not be used in new applications:

```
#include <signal.h>

void (*signal(int sig, void (*function)(int)))(int);
```

The confusing signature is noticably simplified by use of a common `typedef`.

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int sig, sighandler_t function);
```

In other words, a function that takes a handler for a signal, and returns the old handler.

WARNING	This API has a number of problems, including cross-system portability (there are different semantics on different systems) and race conditions regarding how the given signal is dispatched while the signal function is executing.

Managing Signal Sets

There will frequently be situations where an application may want to respond to more than one signal. For example, perhaps `Ctrl+C` should cause an application to return to its starting state, while `Ctrl+\\` should be the termination signal.

When multiple signals are being used, or when the delivery of a single signal needs to be delayed until an appropriate time, it may be necessary to *block* signals from being delivered. This is done using system calls that can modify a list of signals to be blocked. The kernel maintains a bitmap consisting of one bit for each possible signal. If the bit is set, the signal is blocked and won't be delivered. If the bit is clear, the signal will be delivered when it arrives, as in its normal fashion. This bitmap is separate from *how* a signal is being handled. Thus, a signal can be set to `SIG_DFL` or have a handler function assigned to it and still be blocked.

This bitmap is the `sigset_t` type, an opaque type from `signal.h`. Because the size of the bitmap could increase over time, the API is defined to use a `sigset_t` pointer. This allows it to be a 32-bit value on older systems and 64-bits on a more modern system.

There are multiple functions for manipulating the bitmap itself.

```
#include <signal.h>

// Functions for manipulating the bitmap (frequently these are macros)
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

The first two functions set all bits in the mask to zero or one, respectively, and the next two functions add or remove a single bit from the mask. The last function allows for a simple boolean test to determine whether a given bit is set or clear.

NOTE Because there is no "signal 0", the bottom bit represents signal 1, meaning that a 32-bit mask can represent signals 1 through 32 and a 64-bit mask can represent signals 1 through 64.

Useful Signal Functions

The bitmap discussed in the previous section is used to control the *process signal mask*. This mask defines which signals can be delivered immediately (bit clear) or which signals are blocked and are merely added to a queue (bit set).

The system calls in this section manage the contents of the process signal mask.

```
#include <signal.h>

// System call for managing the bitmap inside the kernel
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigpending(sigset_t *set);
int sigsuspend(const sigset_t *set);
```

NOTE

In actual use, the `sigprocmask()` function is merely a front-end to the actual `rt_sigprocmask()` system call. This is done so that the glibc library can hide implementation details that have changed over time.

The following example code shows a simple program that blocks multiple signals and later reports which signals occurred.

sigset.c

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

int main(void)
{
    sigset_t mask;

    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGQUIT);
    sigaddset(&mask, SIGTERM);

    // This blocks all three, wiping out any existing mask in the kernel
    sigprocmask(SIG_SETMASK, &mask, NULL);
    puts("Signals blocked while sleeping for 5 seconds...");
    sleep(5);
    sigpending(&mask);

    printf("Pending signal mask is: 0x%016llX\n", *(unsigned long long*)&mask);
}
```

In the above application, three signals are blocked while a five second delay passes. When that five second interval is up, the application terminates.

However, the program could call `sigprocmask()` again and change the signal mask such that a signal that had been blocked was now unblocked. If the signal had occurred during the five second interval, calling `sigprocmask()` to unblock it would immediately trigger the delivery of the signal. Signal management like this is critical in some applications where an operation that is underway cannot be interrupted, yet the operation could periodically check for a termination request on its own.

For example, an application may be communicating with a web camera, reading frames of video from the camera. While reading the contents of a single frame, the operation should not be interrupted as the device requires the entire frame to be read. But between frames, the application could check to see if a signal had occurred and respond to it. Two solutions are possible in this case: call `sigprocmask()` and unblock signal(s) between frames so that signals are delivered, or leave them blocked but call `sigpending()` to poll for any queued signals. Which solution to choose is dependent on the overall application design; it may be simpler to choose one approach over the other.

It is possible for multiple signals to be blocked and pending. If multiple of those signals are unblocked, it is unspecified in the POSIX standard what order those signals are delivered. For example, if signals 1, 3, and 6 are all blocked and pending, and the application unblocks 1 and 3, POSIX does not say whether

signal 1 or signal 3 is delivered first.

The current Linux kernel (v5.4 as of this writing) actually checks for synchronous signals first and delivers those prior to any asynchronous signals. In addition, it sends the lowest numbered signal first. When all pending synchronous signals have been delivered, then asynchronous signals are delivered. The next chapter has a list of which signals are synchronous.

The last system call to cover is `sigsuspend()`. As its name implies, this system call puts the current process to sleep and waits for a signal to occur. The mask provided to `sigsuspend()` temporarily becomes the process signal mask (as though `sigprocmask()` had been called), to be restored the `sigsuspend()` is complete. (This is performed atomically.) Typical usage is for the mask to be empty, allowing any signal to trigger completion of this system call and the application to continue execution.

Summary

This chapter provides an overview of signal handling on POSIX systems. Non-POSIX systems are typically very similar (for compatibility purposes) but frequently have a much smaller set of available signals, or have signals that are not applicable in a POSIX environment. Always be sure to compare the semantics of signal delivery described here with those on the particular system being used, as the POSIX standard allows some portions of signal delivery to be "implementation defined".

- Signal are asynchronous software events
  - Sometimes the operating system will convert a hardware event into a signal, such as `SIGSEGV` or `SIGBUS`
- Signal handlers are functions within the application that are invoked when a given signal is delivered
- Alternatively, signals can be *ignored* (via `SIG_IGN`) or take their default action (via `SIG_DFL`)
- There are functions to manipulate a bitmap of signals (setting and clearing individual bits)
- The bitmap can define which signals may be delivered and which are *blocked*
- Such bitmaps are only used when installed for the current process
- The operating system can be queried to determine if any blocked signals are pending

Exercises

{exercise!}

Build a multiprocess application that uses signals to communicate.

Exercise 3: `pingpong.c`

Implement a `pingpong` program in which a parent process prints `ping!\t` and a child process prints `pong!\n`.

- As stated, create a program which `fork()`s and the two processes signal each other to synchronize.
- One process should print the first string and the other process should print the second string.
- Each should include a ½-second delay between the message being displayed and the other process being notified.

Sample output

```
$ ./pingpong
ping!   pong!
ping!   pong!
ping!   ^C
$
```

Chapter 9. Signals: Signal Handlers

Students will learn the specifics of writing signal handlers, including how such signal handlers can be invoked in the middle of high-level library operations and the ramifications of such.

- Designing signal handlers
- Async-signal-safe functions
- Interrupted system calls

Introduction

The previous chapter discussed the high level aspects of what signals are and how they are used, but did not delve into the details of signal handlers themselves.

This chapter will discuss:

- How and when signal handlers are invoked
- Which signals are delivered to the ***process*** and which are delivered to the ***thread***
- The environment in which the handler is invoked
- Whether other signals can interrupt a signal handler
- Which system calls are considered "safe" in terms of concurrency and rentrancy

How Do Signal Handlers *Actually* Work?

This diagram describes the block organization of the Linux kernel. What it *does not* include is what happens in user space, i.e. the part of virtual memory that holds application code and data.

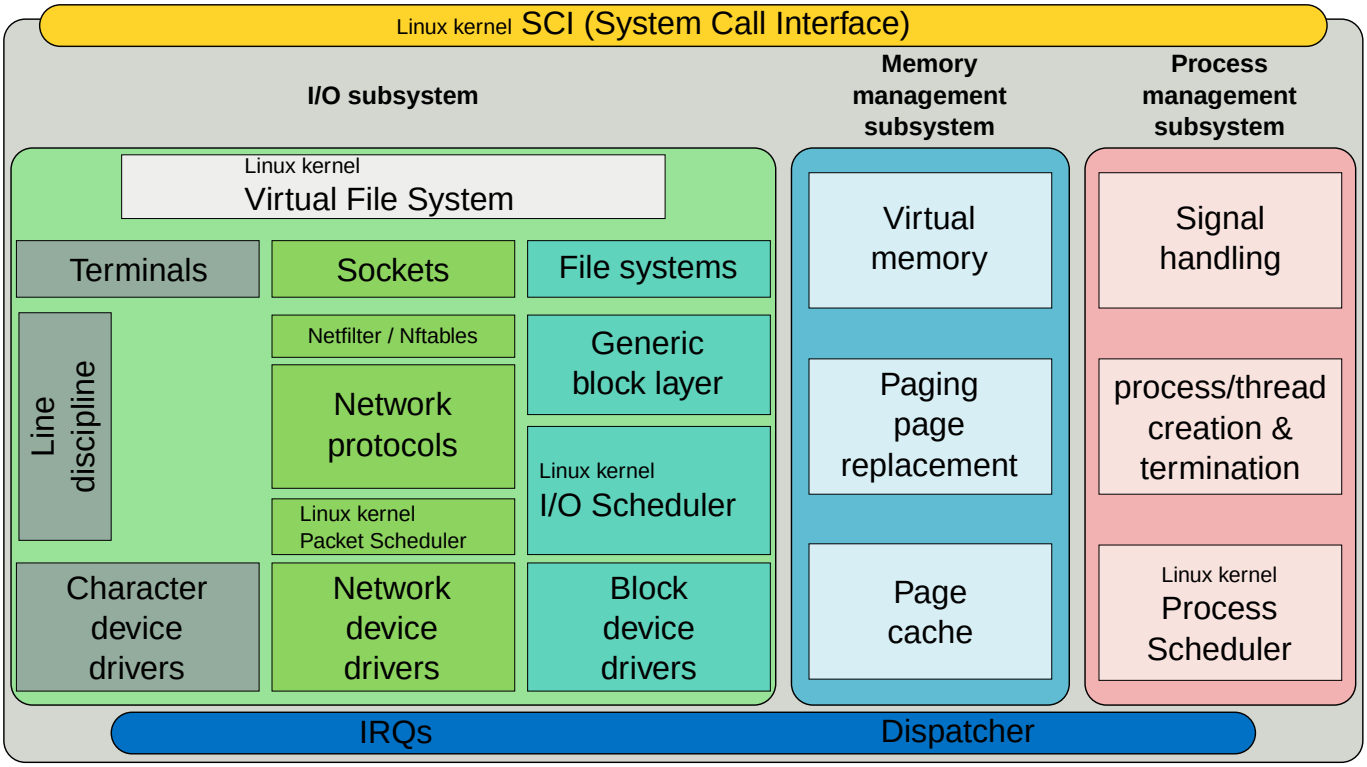


Figure 3. By ScotXW - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=47075153

At the top of the figure is the **SCI** (System Call Interface). Whenever an application wants to perform an operation that requires operating system approval, it invokes a function in that area. The function then delegates to the appropriate block to resolve the request.

So what does this have to do with signals? The signal could occur any any time, while application code is executing or while operating system code is executing. If application code is executing and a signal occurs, it is fairly simple for the kernel to dispatch the signal handler. The general approach is that the operating system checks for unblocked, pending signals for a given process every time it is about to dispatch that process.

Here's a sample scenario involving two processes running under the same user id on a single-core system (for simplicity):

1. Process **A** is running (perhaps calculating π to the last digit; compute-intensive with no use of the SCI)
2. Process **A**'s time slice expires, so the kernel dispatches process **B**

3. Process **B** sends `SIGTERM` to process **A** via `kill(pid_A, SIGTERM)`
    - The kernel checks the signal delivery setting of process **A**:
        - If the signal is being ignored, do nothing.
        - If the signal is not being ignored, set the bit in the list of pending signals.
4. Process **B** continues to run until its time slice expires
5. The kernel schedules process **A** to run again **but** before resuming process **A**, it checks the pending signal list
    - If there are no pending signals, resume **A**
    - If the pending signals are being blocked, resume **A**
    - Otherwise, pick an unblocked signal and deliver it to process **A**
        - If the signal is set to its default handler, handle it internally (termination? core dump?)
        - Otherwise, invoke the application's signal handler
            - This is done by creating a *trampoline* inside the memory space of the application
            - The address of the signal handler is stored in the trampoline
            - The kernel then resumes process **A** inside the trampoline
            - The trampoline calls the handler address that was stored into it
    - Process **A** is now running inside the signal handler
        - When the application's signal handler returns, it comes back to the trampoline. Functions that never return, such as `longjmp()`, leave it forever executing in "signal handler mode" and can only invoke async-safe functions if it wants to avoid undefined behavior.
        - The trampoline invokes a special system call that returns back into the kernel
        - If more unblocked signals are pending, the kernel does the next one using the same technique
        - Otherwise, resume process **A** from where it left off
6. Process **A** is running inside application code (calculating the value of π)

In order to make signals efficient, the operating system will only invoke signal handlers when execution is returning to an application with pending signals from inside the kernel. This can happen under a variety of circumstances:

1. A hardware interrupt queued a signal for the currently running process.
2. A process with a pending signal made a system call and is about to return from that call.
3. A process with a pending signal is about to be dispatched (it is about to receive some cpu time).

All of this means there is an indeterminate amount of time from when a signal is queued for a process to the corresponding delivery of that signal.

A ***trampoline*** is a small piece of code designed as a "jumping off point" between two layers of code. In this case, the kernel allocates a single frame of physical memory and places it at a random location in the application's virtual addressing space. The page contains two small functions. The first function makes a C language call to the application's signal handler, which is read from a pointer allocated from within the same page. This allows the kernel to store the handler's address into the pointer, then jump to the trampoline. The kernel cannot call the signal handler directly because the handler must return into the kernel and not into the application. The kernel must regain control to check for other signals, as well as perform other overhead operations. (This is not ***exactly*** how it works, but it is very close. Additional details are beyond the scope of this course.)

Linux systems implement POSIX real-time signals. These types of signals have extended capabilities beyond the basic signals. They were defined in the POSIX 1003.4 working group on threads, but when the draft was voted on and ratified, it was renamed to 1003.1d to more clearly represent that it was part of the C language function binds. POSIX requires that the operating system provide a queue length of at least 32 for each real-time signal. Real-time signal handlers are passed a `siginfo_t` structure pointer that contains detailed information about the signal event, such as the process id that sent it, a timestamp of when the signal was posted to the queue, and more. Normal signals must also be queued up (required by POSIX) but the standard does not state what the queue must be, so many operating systems implement a queue length of one, allowing them to simply set or clear a bit in a bitmap...

Process Signals vs. Thread Signals

This module has not discussed threads yet, but there are a few aspects of signals that are relevant, so they will be covered here.

At this point, think of threads as "virtual CPUs with a Stack" that execute the application code. This means that there can be multiple functions in the application running at a single point in time. This relates to signals because some signals are triggered as a result of something the application does (like an illegal memory access causing `SIGSEGV`), so those signals should be delivered to the thread that triggered them, right?

Linux defines two categories of signals: synchronous and asynchronous. Asynchronous signals are those that the application cannot predict when they might occur, such as `SIGINT` (the user pressing `Ctrl + C`) or `SIGALRM` (a real-time timer). These types of signals are queued for the ***process***, not individual threads. The next time any code within the process is about to be scheduled for cpu time, the sequence of steps in the previous section occurs. There's no way to predict which thread might be about to run when the signal is delivered.

Synchronous signals are a direct result of something the application did. Let's say that a thread within the application tried to access an invalid memory address. The operating system woudl detect the illegal access (via the page fault mechanism) and generate a `SIGSEGV` as a result. This signal would be queued to ***the specific thread that triggered the signal***. Only that thread can respond to the signal, so when that thread is about to receive cpu time, the operating system uses the sequence in the previous section.

The current Linux kernel (v5.4 at this writing) defines six signals as synchronous. See `kernel/signal.c` in the kernel source code: `SIGSEGV`, `SIGBUS`, `SIGILL`, `SIGTRAP`, `SIGFPE`, `SIGSYS`.

Here are some observations based on the above. To begin with, each and every thread has a `sigset_t` defining which signals are blocked and which signals are pending ***for that specific thread***. In addition, all threads that belong to a given process point to the same "shared" `sigset_t` of pending signals for the process.. When an asycnhronous signal occurs, it is added to the current thread's pending list, but also the shared list. When a synchronous signal occurs, it is only added to the pending list of the thread that triggered the signal.

Signal Management in Processes and Threads

More discussion about how threads respond to signals will be covered in the threads chapters, later in the module.

Signal-Safe Functions and System Calls

Signals can be delivered at any time. This means the application could be in the middle of a `printf()` call when a signal handler is invoked. So what happens if the signal handler invokes `printf()`? If `printf()` is not thread-safe and re-entrant, it's likely that `printf()` will not function properly.

There are ways to work around this. The first is to disable signals any time a function is going to be invoked that isn't thread-safe. This is clearly not practical!

Another option is to wrap the existing C library functions with code that disables signals. This is essentially the same solution, but with the work taken out of the hands of the application programmer.

The ideal option would be for C language library functions to be thread-safe! Unfortunately, the C standard defines how these functions must work and some of those definitions directly contradict the ability to be thread-safe. An example of that is the C function, `strtok()`, but there are quite a few others.

The only portable thing for programmers to do is to avoid the use of functions that could be problematic. To that end, POSIX has specifically defined system calls that must be thread-safe. And in the latest version of the 1003.1 standard, the same thing has been done for language functions. They are not listed here (there are over a hundred), but they can be found in `man signal-safety`.

Interrupted System Calls

The last section discussed what happens when a signal invokes a function or system call that fails to be re-entrant.

But what if the signal handler does everything it's supposed to do and plays nicely with everyone else. Does it still have an effect on the main application code? The answer is: maybe.

Suppose that an application is in the middle of reading input from the keyboard. The code has been waiting a long time for the user to type something, but while the application is waiting, a signal occurs and a handler is invoked and completes successfully. Should that trigger an error from the `read()` system call that is reading from the console? Keep in mind that I/O operations that are long duration are unlikely to ever finish if signal handlers are causing them to return with an error!

The POSIX approach is to allow the application to set the `SA_RESTART` flag inside the `struct sigaction` when registering a handler for the signal. The operating system can then restart the system call without reporting an error back to the application. This makes perfect sense, since most applications would look at the error code, `EINTR` (which means "Interrupted system call"), and simply re-invoke the system call again anyway. It's much cleaner to let the operating system handle it internally.

But what about I/O operations that are normally pretty fast (relatively speaking)? In those cases, the `EINTR` is returned.

Finally, when a slow I/O operation is interrupted by a signal handler, it's possible that it could return a partial buffer prematurely. Using the above example of an application waiting for user input, suppose the user types a few letters and then a signal occurs. The signal handler will be invoked and when it returns, the operating system will cause the `read()` operation to terminate with a partial buffer. The application will need to retrieve that partial buffer and perhaps call `read()` again to wait for the rest of the user input.

This type of input handling is only needed by applications that communicate with slow devices, have handlers setup for some signals, and don't block those signals during the I/O operations. If a programmer's application falls into that group, they will often create a low-level function that does the I/O and handles the partial buffer so that the rest of the application can simply call that function.

Summary

This chapter has described in detail how the signal handlers themselves are coded. Keep in mind that signals can be delivered at essentially any time. (Signal delivery only occurs when the kernel is returning to a user space application. From the application's point of view, there's no way to know when that is.)

It might seem that the application programmer can tell which functions enter the kernel and which don't by simply reading the documentation. But some functions in the C library cache the results of system calls, so the second and subsequent calls don't really cross the user-space/kernel boundary. An example is `getpid()`—the C library knows that the process id can't change over the lifetime of the process, so only the first invocation is really a system call. Similarly, any hardware interrupt can invoke a device driver that then posts a signal to any process id, and when the interrupt completes, the signal *may* be delivered.

Exercises

{exercise!}

Add signal handling to the `cp` program.

Exercise 3: `cp_signal.c`

1. Modify the `cp` command from a previous exercise to allow `-` as an input filename to represent stdin.
2. Add handling for `Ctrl + C` so that the application terminates cleanly, removing the current destination file if it hasn't been completely copied yet.
3. Modify the `cp` command from a previous exercise so that `Ctrl + \\` prints a report to stderr of how much data has been written to the current destination file so far (include the destination file name and a byte count).

To interactively test the `cp` command's signal handling, follow this procedure:

1. Execute `cp_signal - output`. The `cp` command should now be reading from stdin.
2. Type a few lines of text, each ending with `Enter`.
3. Type `Ctrl + \\` and the filename and number of bytes should be displayed.
4. Type a few more lines and `Ctrl + \\` again. Does the output look correct?
5. Type `Ctrl + C` and ensure the `output` file is removed.

Advanced: what does your solution do if `-` is used on the command line multiple times? For example, assuming that `file1` exists:

```
$ ./cp_signal - file1 - /tmp
```

Chapter 10. Process Lifecycle

Students will learn more about the process lifecycle on POSIX systems. They already have some experience with `fork()` and `execvp()`, but this chapter adds details on open file descriptors and `dup()`, how a parent detects and handles job control signals, and some "behind the scenes" details.

- Creating a new process: `fork()`
- File descriptors and `fork()`
- Process termination
- Monitoring child processes
- Orphans and zombies
- The `SIGCHLD` signal

Introduction

So far, this course has discussed how to open files and how to read/write them, how buffering works, how directories are managed, what hard links and symbolic links are and how they're used, and many more topics. But the concept that underpins all of these is that of the *process*. It has been loosely defined as, "the environment in which an application runs", such as open file descriptors and the virtual memory allocated to it.

This topic will go into the details:

- Processes are created by `fork()`, but what happens behind the scenes?
- What happens when a process terminates?
- What are *orphan processes* and *zombie processes*? What effect do they have on the system?
- How can a parent be notified that a child process has changed state? Would it even want to be notified?

Process Creation

The student knows that `fork()` creates new processes. What is happening behind the scenes to make this work?

1. The existing process (called the *parent*) invokes `fork()`.
2. The new process (called the *child*) is created as a duplicate of the parent:
    a. Per POSIX, these things are copied from the parent:
        i. All open file descriptors are duplicated into the child.
        ii. All credentials are copied into the child.
        iii. All filesystem data (current directory, root directory) are copied into the child.
        iv. All CPU scheduling data is copied (such as process priority).
        v. All virtual memory pages of the parent are changed to read-only and then copied into the child. Both parent and child can share the same RAM, saving a significant amount of overhead on the system. But if either process tries to modify the virtual memory, it will page fault. The operating system will examine the virtual address and recognize this situation; it will copy the single physical frame of RAM into a different physical frame, giving the parent and child their own copies of the page.

            Now the process that attempted to write to the page can attempt it again — it will work this time because the newly allocated memory was marked read/write, not read-only. This technique is called *copy-on-write*, or just COW, for short. This means the child starts with the exact same executable code, data, heap, and stack as the parent. In fact, all shared memory blocks and memory mapped files are copied as well. This means the child starts with the exact same executable code, data, heap, and stack as the parent. In fact, all shared memory blocks and memory mapped files are copied as well. They are copied *virtually*, not *physically*. It would not be efficient to *physically* copy all of the memory an application might allocate!

    b. Also per POSIX, these things are allocated fresh:
        i. Process id and parent process id are set (guaranteed to be unique on the system at any given time; NOT sequential).
        ii. Accounting data is initialized (CPU usage, amount of I/O performed, maximum memory used, etc).
        iii. Memory locks (such as via `mlock()`) are not inherited.
        iv. File locks are dropped (both processes cannot have the same region of a file locked). (However, POSIX "open file description" locks via `fnctl()` and `flock()` locks are inherited.)

v. Semaphores are held by the parent only (again, same as with file locks).

vi. The pending signal set is initially empty.

vii. No timers are inherited (such as from `setitimer()`, `alarm()`, or `timer_create()`).

viii. Asynchronous I/O operations already enqueued will not happen inside the child.

ix. The child has a single thread (referred to as the *main thread*) and that is the one that is running when `fork()` returns; other threads are not duplicated. However, the memory state in the child *is* duplicated, so any mutex locks that might have been held by a thread somewhere in the parent will appear held in the child. This essentially means that until the child calls `execve()`, only async-signal-safe functions can be safely called.

c. Linux has additional platform-specific differences between parent and child:

i. The child does not inherit directory change notifications.

ii. The `prctl()` `PR_SET_PDEATHSIG` setting is reset so the child does not receive a signal when its parent terminates.

iii. The `prctl()` `PR_SET_TIMERSLACK` setting is not inherited.

iv. Memory mappings using the `madvise()` `MADV_DONTFORK` flag are not inherited.

v. Memory in address ranges marked with `madvise()` `MADV_WIPEONFORK` are zeroed in the child.

vi. Port access permissions set by `ioperm()` are not inherited by the child.

vii. Although the child receives a copy of the parent's file descriptors, they still point to the same `struct file` within the kernel. This means file offset is shared, as well as status flags and signal-driven I/O attributes (see the description of `F_SETOWN` and `F_SETSIG` in the `fcntl()` man page).

viii. Opened directory streams (as returned by `opendir()`) are shared and this the stream positioning *may* be shared (per POSIX). However, Linux does not share position information between child and parent.

The `fork()` system call is unusual from the application programmer's perspective because it is only called once, but it returns twice. While the `fork()` is inside the kernel—and the kernel is allocating and initializing the virtual memory for the child—the return address that is on the stack is the virtual address of the instruction immediately after `fork()`. When the child runs, it will return to that point; and when the parent runs, it will also return to that point. If the function is invoked once and returns twice, how can the application tell whether the returning process is the parent or the child?

One way to do it would be for the parent to store its process id in a variable prior to calling `fork()`. When `fork()` returns, compare the current process id against the one in the variable. If they are the same, the running process must be the parent. If they are different, the running process must be the child.

But the operating system uses a different trick that doesn't require any setup on the part of the

application.

The return value of `fork()` is always zero in the child, and always non-zero in the parent. The non-zero value could be `-1`, indicating that the `fork()` failed, or it could be the actual process id of the child. The child doesn't need this information. If the `fork()` failed, then the child doesn't exist. And if the `fork()` succeeded, the child can call `getpid()` or `getppid()`, if it needs that information.

WARNING

The `fork()` system call can fail for a wide variety of reasons. It is imperative that the application check the return value from `fork()` and handle errors appropriately. Failure could be the result of a lack of either physical or virtual resources.

Example of calling `fork()`

```
pid_t child = fork();

if (child < 0) {
    // Error when trying fork()
    perror("Unable to fork");
    exit(...);
} else if (child > 0) {
    // Parent, do work (often involves waitpid(child))
} else {
    // Child, do work (often involves execve(...))
}
```

Once the child has been created, it runs in parallel with the parent. The child will start with the same process priority as the parent, so it is indeterminate which one will execute first on a single-core system. It will begin to accumulate CPU time, I/O bandwidth, and so on, and the operating system will track those values. They can be displayed using a variety of command line tools, such as `ps`, `pidstat`, and `top`.

File Descriptors vs. `FILE *` With `fork()`

An application needs to be careful when calling `fork()`. If they are not, data corruption of output can occur. Here is why.

When an application calls `fork()`, it's invoking the kernel directly. This is not a function that is part of the ANSI C library, for example. But speaking of ANSI C, what happens to the buffers being used by any `FILE *` data types that might be in use? For example, stdin, stdout, and stderr are all automatically opened when the application starts and the C library allocates a buffer for each one. When the application calls `fork()`, is the buffer duplicated?

Yes, of course—it's part of the contents of virtual memory. This means that if there is data sitting in the buffer that has not been flushed yet, that data will also appear in the child's virtual address space. And when each process eventually flushes its own buffer, the data will appear twice in the output. Oops.

The solution to this is simple: empty the buffers before calling `fork()`. For ANSI C library buffers, this can be accomplished by calling `fflush()` and passing it a `FILE *`. If an application has implemented its own buffering scheme internally, it will need to provide a mechanism for those buffers to be flushed as well.

Process Termination

Processes that call `exit()` will terminate. There are multiple things that happen within the process environment before the operating system can actually remove the task (remember the discussion of the `task_struct` data structure?).

Among them are:

1. Flush all process IO buffers and streams (these are the ANSI C buffers)
2. Close all open file descriptors
3. Release all references to memory structures (such as process credentials, shared `vm_area` regions, shared libraries, held semaphores, and others)
4. Accumulate the CPU of the process into its parent's *child process times*

When all resources of the process have been reclaimed, the process moves from the `TASK_RUNNING` state to the `TASK_DEAD` state. This is represented in the output of `ps` as the letter `Z` in the ***flags*** column and the word `defunct` in the command column. Each thread that belongs to the process is changed to the `EXIT_DEAD` state as well.

How Can a Parent Monitor a Child Process

But how are those “zombie” processes cleaned up?

When a process calls `fork()` and creates a child process, the operating system records the parent’s PID in the child’s `ppid` field of the `task_struct`. Whenever the state of the child changes in a way which is meaningful to the parent, the operating system will send the `SIGCHLD` signal to the parent. If the parent then calls one of the `wait()` family of system calls (which includes `waitpid()`, `waitid()`, and `wait4()`), it can retrieve the state change information. The most important field returned by `wait()` is the ***exit code*** (known to shell programmers as `$?`).

A process that uses `fork()` to create a child should therefore setup a signal handler for `SIGCHLD`.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int options);
int   waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

In each of the `wait()` functions, there is the capability of retrieving the status of the child process. This is the `wstatus` field in `wait()` and `waitpid()`, and is a member of the `siginfo_t` structure in the `waitid()` function.

This book will refer to the functions in the above list as the singular "`wait()` function".

Orphans and Zombies (and Other Process States)

The term *zombie* means a process that is dead, but which hasn't been "buried" yet. Meaning, the process has finished its job and called `exit()`, but the parent hasn't called `wait()` and retrieved the exit code yet. Until the exit code is retrieved, the operating system must continue to maintain an empty `task_struct` (meaning the job will continue to appear in `ps` listings). When the exit code is finally obtained by the parent, there will be no reason for the `task_struct` to continue to exist and it will be released (it goes back into a pool of available `task_struct` objects, to be reused).

The Purpose and Use of SIGCHLD

Because the SIGCHLD signal is sent whenever the state of a child changes, it's possible for the signal to represent more than just child termination. Here is a table showing the possible causes, with the most likely ones listed first.

WIFEXITED(wstatus)	Process terminated. Use WEXITSTATUS(wstatus) to retrieve the exit code.
WIFSIGNALED(wstatus)	Process terminated via signal. Use WTERMSIG(wstatus) to retrieve the signal number.
WIFSTOPPED(wstatus)	Process stopped via signal. Use WSTOPSIG(wstatus) to retrieve the signal number.
WIFCONTINUED(wstatus)	Process was resumed via SIGCONT.

Summary

Students have learned about the process lifecycle on POSIX systems. This covers process creation, process termination, and the affects in the parent of transition in the state of child processes.

- Creating new processes with `fork()`
- File descriptors and `fork()`
- Process termination
- Monitoring child processes
- Orphans and zombies
- The `SIGCHLD` signal

Exercises

{exercise!}

Exercise 4: `counter.c`

Write a program that spawns a child process that prints the numbers 1–10 to the `stdout`. The number *N* of such children should be specified with a `-c N` command-line option. Each such child process should sleep a random amount of time (from 0.5–1.5 seconds) per number printed out, and each number should be prefixed with the child process ID.

Exercise 5: `counter_signal.c`

Extend the `counter` program to respond to signals. If the parent process receives `SIGUSR1`, another child should be spawned.

Chapter 11. Executing Programs

Students will learn how applications can invoke other applications, passing them environment variables, command line parameters, and even open file descriptors. They have been introduced to these topics already, but there are many more details to cover.

- Executing programs: `execve()`
- The `exec()` library functions
- File descriptors and `exec()`

Introduction

A previous chapter has already covered an overview of how an application can execute another application. A simplistic view of passing command line arguments and environment variables was provided, but there are other important details that will be covered in this chapter.

For example, the `execve()` system call replaces the contents of virtual memory with another application. Therefore, all virtual memory mappings are removed (shared libraries, shared memory, memory mapped files) prior to the new application being loaded. However, any aspects of the process that are not tied to its virtual memory are preserved, such as file descriptors and credentials. This chapter will explore how those components might be used by applications.

This chapter will use the singular "`exec()` function" to mean all functions within the `execve()` familiar (`execl()`, `execlp()`, and so on).

Executing programs with `execve()`

Previous chapters have covered a simple use of `exec()` to execute other applications and pass environment variables. This section will discuss the use of `exec()` in more detail.

1. How many arguments can be passed?
2. How many environment variables can be passed?
3. How much memory can the above two items consume and the call still succeed?
4. Can file descriptors be automatically closed on `exec()`?
5. If file descriptors are not closed, how can the new application be notified of them?

First, a quick list of other functions in this family of function calls and a review of what has already been covered:

```
#include <unistd.h>

// This is the underlying function used by the ones below.
int execve(const char *pathname, char *const argv[], char *const envp[]);

// These require a pathname to the executable (first parameter)
int execl(const char *path, const char *arg0, ... /*, NULL */);
int execle(const char *path, const char *arg0, ... /*, NULL, char *const envp[] */);
int execv(const char *path, char *const argv[]);

// These only need a basename; the PATH environment variable will be searched
int execlp(const char *file, const char *arg0, ... /*, NULL */);
int execvp(const char *file, char *const argv[]);
int execvP(const char *file, const char *search_path, char *const argv[]);
```

All argument lists and environment variables lists must be `NULL` terminated.

This system call is given the name of the application to execute and a list of command line arguments and environment variables:

1. It causes the operating system to replace the virtual memory of the current process with the image specified.
2. The operating system clears the virtual address space of all existing mappings and loads new ones.
3. The maximum memory available to hold the `argv` and `envp` parameters is limited to 25% of the stack size limit (as specified by `getrlimit()`) and 75% of the kernel constant `_STK_LIM` (8MB), with a minimum size of 32 pages (at 4KB each, 128KB).

In addition, there is a limit of 32 pages (128KB) per string, and a limit of two billion (minus one) strings.

4. The actual stack is set just below the `argv` and `envp` area, and a startup stub is loaded and invoked (the dynamic loader, `ld.so`), passing it the locations of `argv` and `envp`.
5. The dynamic loader identifies which shared libraries are required, if any, and maps them into memory.
6. After all libraries are mapped, the `main()` function is invoked and is passed the number of elements in `argv`, along with the locations of where `argv` and `envp` were copied (although POSIX doesn't specify `envp` as a parameter and instead requires the use of the `environ` global variable).
7. When `main()` returns, it invokes `exit()` and some ANSI C housecleaning is done and the application terminates.

How many arguments can be passed to `exec()`?

From the summary in the previous section, you can see that the number of elements in the `argv` array are constrained only by memory availability. For a system with an 8MB stack size (the default on modern Linux systems), the upper limit on memory consumed is 2MB. On 64-bit systems, the minimum size of one entry would be a 64-bit pointer, the string itself (assume it's empty), and the terminating null byte, for a total of 9 bytes. If the environment were empty of variables, this leads to a maximum of 233,016 command line parameters, all of which must be empty. Assuming a more typical eight characters per option, the limit drops to 123,361 command line parameters. That should be enough for most uses!

Since `argv` and `envp` share the same allocation area, each consumes memory available to both; as the `argv` becomes larger, there is less available for `envp`, and vice versa.

Alternate front-ends to `execve()`

The difference in the last group of `exec()` functions (those that take a basename instead of a pathname), is that they search the `PATH` environment variable looking for the executable. Consider these two sample function calls, assuming that `bash` only appears in `/bin`:

```
extern int execv(
    char *path, char *const argv[]);
...
execv("/bin/bash", argv);
```

```
extern int execvp(
    char *file, char *const argv[]);
...
execvp("bash", argv);
```

The one on the left specifics an absolute pathname to the executable, while the one on the right uses a relative pathname.

The `execv()` function on the left requires that the first parameter use a pathname to the executable, but does not require it to be an absolute pathname. This means if `execv()` is called and given `"bash"` as the first parameter, it will operate properly only if the process has `/bin` as its current directory.

The `execvp()` function on the right can be given a basename (not a pathname) and the `PATH` environment variable will be automatically searched. If the command name contains a slash anywhere in its name, no search is performed and the string is treated as a pathname. If there is no `PATH` variable in the environment, the results of `confstr(_CS_PATH)` are used, which defaults to `"/bin:/usr/bin"`. (Execute `getconf PATH` to see the defaults.)

The documentation states that the `*p()` functions (those ending with `p`) "duplicate the actions of the shell" in searching for an executable file. However, the Bash, Korn, and Zsh shells have additional functionality, such as caching the results in a hash table to increase performance on lookup.

File descriptors and `exec()`

Be default, file descriptors are not automatically closed when the process calls `exec()`. Instead, opened file descriptors remain open in the new application.

In addition, the references to a shared set of `struct file` objects created during `fork()` are also copied into a new set of `struct file` objects. This allows the new application to manipulate its file descriptors without impugning any aspect of file descriptors in its parent.

In fact, this is how the typical shell works: it has stdin, stdout, and stderr open already, so it calls `fork()` to create the child process, and the child then calls `exec()` to run the command line, preserving file descriptors zero through two along the way.

Could the shell leave other file descriptors open as well? Of course—there's nothing to stop it from doing so. And the Bash shell even does this—it opens file descriptor 255 and leaves it open. This allows the shell to have a ***controlling tty***, meaning that the shell will receive `SIGINT` if the user types `Ctrl + C` on the terminal. There is even an option to `open()` to prevent the allocation of a controlling tty if a process opens a terminal (see `O_NOCTTY`).

How can an application notify the one to be `exec()`'d that there was a file descriptor purposefully left open for it to use? There are two techniques, the first more portable than the second. The first is to provide the file descriptor number on the command to the new program. The restriction for this approach is that the new program needs to know to expect the file descriptor number; very few (existing) programs fall into this category.

The second technique is to pass a filename to the new program that refers to the file descriptor. This approach allows the new program to treat the file descriptor just like any other filename. Linux (and other Unix systems) accomplish this by providing a list of open file descriptors under the `/proc/self/fd` directory. For example, in order to pass file descriptor 6 as a filename, add `/proc/self/fd/6` as an argument to `argv`. When the new program tries to read that file, it will read from file descriptor 6.

Miscellaneous Notes

A few notes on executing other programs.

In general, executing command lines is fraught with portability issues. At a minimum, they include:

1. The inability to know in advance that a given program will exist.
2. Supported options can vary across systems even when the commands are there.
3. ANSI C defines the `system()` function but it is inefficient in that it invokes a shell to process the command string.
4. Additionally, the `system()` function waits until its command finishes and blocks `SIGCHLD` during this time, which makes it impossible to handle other children that might terminate before `system()` returns.

Summary

Students have learned more details behind how POSIX applications can execute other applications.

- Details of executing programs with `exec()` functions
- Use of alternate `exec()` functions
- File descriptors and `exec()`

Chapter 12. Threads: Concepts

Students will learn what POSIX threads are and how they can be used to add concurrency (and parallelism) to application logic. This chapter is concerned primarily with concepts, such as what threads are, how POSIX and C manage threads, and operational concerns like deadlocks.

- Pthreads API basics
- Thread creation and termination
- Thread IDs
- Joining and detaching threads
- Thread attributes
- Signals and threads
- Overview of race conditions

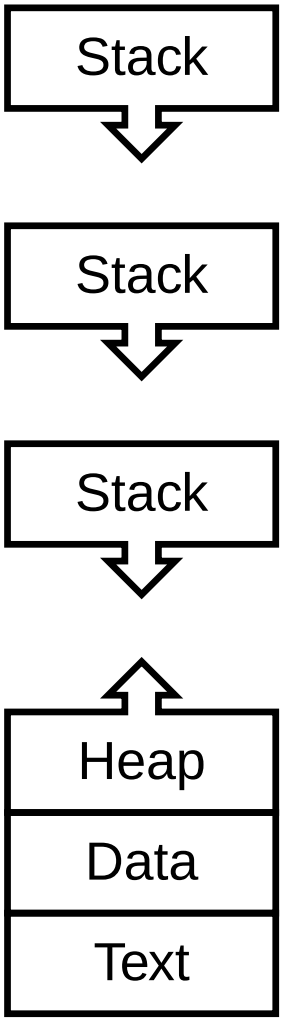
Introduction

A thread is short for *thread of execution*.

It represents a code that the CPU will take as it executes instructions. It goes beyond the concept of *process* by allowing multiple "virtual" CPUs to be executing code within the process simultaneously.

There are numerous advantages to this approach:

- Memory is shared, so there is no overhead to communicating between threads.
- Each thread receives its own CPU stack, allowing multiple threads to invoke the same code simultaneously.
- Threads are very lightweight structures, making creation and destruction of them significantly faster than processes.



There are also some disadvantages:

- Because memory is shared, any access to data that might be accessed by two threads must have

some kind of synchronization protection. (Rule of thumb: avoid `static`-storage data!)

- The main thread of a process has its stack allocated for it when the process begins, but if the application wants to specify the size of a thread stack, it must also allocate the memory itself. Planning for this can be complicated in 32-bit environments where virtual memory may be limited.
- Because certain implementation details of threads can vary from one operating system to the next, there may be user-visible differences. For example, one system might allocate process ids and thread ids out of different pools of numbers, while another system might use the same pool. This is user-visible when it comes to commands like `kill` and `ps`.

Threads are good solutions for increasing application performance in certain cases:

1. When an application implements an algorithm that can benefit from parallelism. Examples include sorting and searching.
2. When an application has a mix of IO bounds and CPU bound functions. One or more threads can queue up an IO operation and then wait for them to finish, while other threads continue their CPU bound tasks. Such an organization makes application design simpler.
3. When multiple functional units of the application can proceed without the need to access shared data. If shared data ***is*** needed but is access infrequently, threads may still be a good approach. But excessive access to shared data will require extensive synchronization, and will ultimately hurt overall performance and each thread waits for another thread to finish with a resource.
4. When an application is executed on a multicore hardware platform. Multithreading when there is only a single core is self-defeating—the added overhead of managing multiple threads and switching between them is likely slower than the equivalent application designed to use only a single thread.

Multiple processes may be preferred in some cases:

1. Security: processes don't share any of their environment and therefore operate like "sandboxes", where errors in the code are confined to that process.
2. Compatibility: if an existing program already exists to perform some function, it likely runs as a process (running as a thread sinde a different process would require a completely different design).
3. When synchronization becomes a bottleneck. (Using processes doesn't automatically solve this issue and, in fact, may create synchronization issues of its own, but it is a possible reason for choosing processes over threads.)
4. On operating systems that don't have ***native thread support*** in their kernel, multiple threads in a single process mean that when one thread blocks, ALL threads block! (Fortunately, very few modern operating systems fall into this category. Even embedded systems often run multithreaded kernels.)
5. When communication between functional blocks is minimal and doesn't require the high-speed communication abilities of threads, processes may be a good choice given the other considerations listed above.

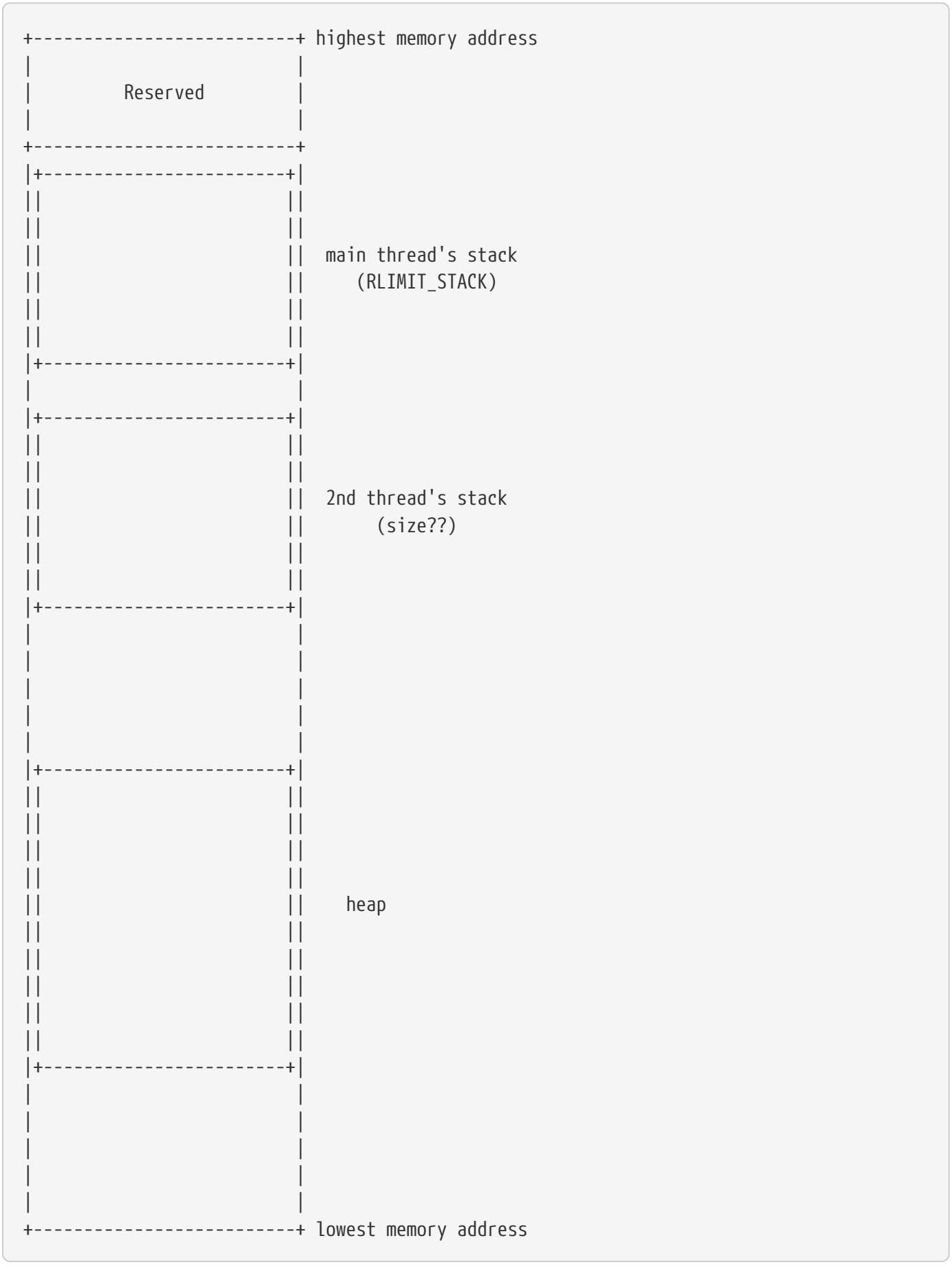
Thread Internals

An important concern for writing applications that utilize multiple threads is stack allocation and size.

When the CPU executes an instruction that causes a function call, the location of the instruction following the function call is pushed onto the ***stack***. Later, when the function returns, that address is ***popped*** back off the stack and the CPU jumps to that location. (There's more happening than this, but we're only concerned about the stack concept at this point.)

When a process starts, it has a single ***main thread*** that is where the application starts. This is the thread that executes `main()`. The operating system automatically allocates the stack (the `RLIMIT_STACK` value returned by `getrlimit()`) for this main thread. Other threads that `main()` creates by calling `pthread_create()` either receive a default stack size, or receive a stack as specified by the caller. On Linux, the options when using the Native POSIX Threads Library (NPTL) are:

1. The application calls `pthread_create()` without specifying stack location or stack size: default size is the `RLIMIT_STACK` value if it is not "unlimited" and 2MB otherwise.
2. The application calls `pthread_create()` and specifies an explicit size; the minimum size is `PTHREAD_STACK_MIN` (16KB) and must be a multiple of the page size (see `sysconf(_SC_PAGESIZE)`).
3. The application calls `pthread_create()` and specifies an explicit location and size; the minimum size is the same as the previous option, and the location must be page boundary aligned as well.


highest memory address
Reserved
main thread's stack
(RLIMIT_STACK)
2nd thread's stack
(size??)
heap
lowest memory address

The operating system will assign one CPU core to each thread, as long as enough cores are available. Otherwise, remaining threads will be kept in the ***run queue*** awaiting their opportunity to be scheduled for CPU time. On Linux systems, each thread is given a priority and a scheduling class and these can be tuned (along with other parameters) to control how the scheduler decides which threads should be assigned to which CPU cores, and for how long. Further discussion of thread scheduling is beyond the scope of this course (many college level operating system design courses cover this topic).

Simple Usage

Here is the API for creating threads:

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine)(void *), void *arg);
```

There are four parameters. (All data structures should be treated as opaque by the programmer. However, looking inside these types can sometimes be enlightening. This module won't be doing that.)

1. The first is a `pthread_t` object.

   It represents the thread being created and all future calls that manipulate the thread will require it to be passed as the first parameter.

2. The second is `pthread_attr_t` and represents the options used to create the thread (such as stack location and size).

   The attribute structure is not modified and can be reused for multiple `pthread_create()` calls.

3. The third is the address of the function to be invoked by the thread when it begins execution.

   It can be tough to read, but this parameter is essentially the same type as a signal handler: a pointer to a function that takes a `void` pointer as a parameter and returns a `void` pointer. (Signal handlers take an `int` parameter and return `void`.)

   If this function returns, it effectively calls `pthread_exit()` and terminates the thread. It takes one parameter, a `void` pointer.

4. The fourth parameter is a `void` pointer that is passed to the `start_routine` as a parameter.

   This is frequently an application-defined structure that contains all of the information needed by the function to accomplish its job. For example, a thread that downloads a file may be given a URL and a file descriptor opened for `O_WRONLY` as members of a single structure whose address is passed as this parameter.

This example demonstrates the usage of threads at a high level. Later sections will discuss each piece of this example in more detail.

Very simple thread example

```
#include <stdio.h>
#include <pthread.h>

void *greet(void *ptr)
{
    fputs("Hello ", stdout);
    puts(ptr);

    return NULL;
}

void *part(void *ptr)
{
    fputs("Goodbye ", stdout);
    puts(ptr);

    return NULL;
}

int main(void)
{
    void* unused_result;
    pthread_t alpha, bravo;

    pthread_create(&alpha, NULL, greet, (void*) "World!");
    pthread_create(&bravo, NULL, part, (void*) "World!");

    pthread_join(alpha, &unused_result);
    pthread_join(bravo, &unused_result);
}
```

Already threads can produce surprising results when run. There are four different ways that the output from this program can be printed, depending on the order and timing of the threads.

Overview of the Thread Lifecycle API

When interfacing with threads (creation, destruction, etc), there are a few common functions that almost every application will use.

- `pthread_create` was briefly discussed, above.

  The most important aspect of this function that once a thread is created, many of its attributes cannot be changed. Therefore, it's important to create and use a `pthread_attr_t` object to control thread creation parameters.

- `pthread_cancel` is a function that requests the termination of a thread.

  It looks like this may be a good way to forcefully terminate a thread, but the reality is that this function can be dangerous to use. A thread could die whenever it next makes a system call (this function essentially sets a flag is checked by the operating system upon entry to the kernel). This means resources used within the process may not be reclaimed properly. For example, a lock held by a thread would not be released, then blocking all other threads from accessing the object it protects.

  In addition, a thread can control its own ***cancelability state*** and ***type***. This means it can decide when it is cancelable and when it isn't. This allows a thread to turn off its cancelability while holding locks, for example. But then what's the point of cancelability if the thread can't be canceled?

- `pthread_exit` allows a thread to terminate itself. This is effectively invoked automatically when the `start_routine` returns.

  If no other threads are alive, the pthreads library will automatically terminate the process, too. It is common for `main()` in a multithreaded application to call this function, so as to allow other threads to run to completion. (Normally the `exit()` function is invoked when `main()` returns, and `exit()` is defined to terminate the entire process.)

  Note that `pthread_exit` creates zombie threads. To prevent long-lasting zombies, either call `pthread_join` on them, or put the thread into the ***detached state*** before it exits.

- `pthread_join` waits for the given thread to complete and stores its return value into the second parameter. This is the only way to determine via the pthreads API that another thread has terminated.
- `pthread_detach` (and the corresponding attribute flag, `PTHREAD_CREATE_DETACHED`) puts a thread into the ***detached*** state. In this state, the system will automatically cleanup any zombie created when the thread terminates. It is not a joinable thread when it is in this state.

  Use the detached state for threads that you will not be waiting for. Their resources will be claimed immediately upon exit. The default state is `PTHREAD_CREATE_JOINABLE`. Joinable threads should either be

joined by calling `pthread_join`, or set to the detached state via `pthread_detach`.

For details on the thread lifecycle, see the man page for `pthreads(7)`.

Thread Attributes

Threads can be created with a variety of attributes applied to them. These attributes control the operation of the thread, or how the application interacts with the thread.

Attributes are represented by `pthread_attr_t`. The second parameter to `pthread_create()` is a pointer to an object of this type (or `NULL` if the defaults are not being changed).

Once the space for the object has been defined, the object should be initialized. Because they are opaque data types, it is possible that initialization of the attribute requires some behind-the-scenes resource allocation. For this reason, they should never be initialized directly, and they should always be destroyed when no longer in use by calling `pthread_attr_destroy()`.

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

The following list of attribute functions are available. Many applications don't need to query or adjust any of these settings away from their default, except perhaps the stack size.

`pthread_attr_setstack()`
`pthread_attr_setstackaddr()`
`pthread_attr_setstacksize()`
	Set the stack location and size. The first function sets both, the others each set one aspect of the stack.

`pthread_attr_setguardsize()`	Set stack guard size; default is one page. Not used if the stack address has been set.
`pthread_attr_setscope()`	Defines the contention scope for resources such as CPU time. `PTHREAD_SCOPE_PROCESS` means a thread competes against other threads within the process, while `PTHREAD_SCOPE_SYSTEM` means the thread competes against again all other threads on the system that are in the same scheduling domain. POSIX requires that a system must support at least one of these; Linux only supports `PTHREAD_SCOPE_SYSTEM`.
`pthread_attr_setschedparam()`	Sets scheduling params. The only support parameter on Linux is priority. For this option to take effect, the inherit-sched flag, below, must be set to `PTHREAD_EXPLICIT_SCHED`.

`pthread_attr_setschedpolicy()`	Set the scheduling policy to `SCHED_FIFO`, `SCHED_RR`, or `SCHED_OTHER`.
`pthread_attr_setinheritsched()`	Defines whether scheduling attributes are inherited or come from the `pthread_attr_t` object. The parameters affected are the contention scope, scheduling priority (param), and scheduling policy. The default is `PTHREAD_INHERIT_SCHED`. See the man page for an important bug note.
`pthread_attr_setaffinity_np()`	Set CPU affinity; more portable to use `sched_setaffinity()`

Signal Handling in Threads

Signals are a property of processes that involve assigning a function to be invoked when a signal is delivered. This means that all threads share signal handlers (because all threads share the same virtual address space). Threads also share a process-global signal mask that controls which signals are blocked (see `sigprocmask()`).

However, signals can be synchronous (they occur because of something the software did) or asynchronous (they occur for a reason other than software). These two types of signals are handled differently:

- synchronous signals are delivered in the context of the thread which triggered them, and
- asynchronous signals are delivered in the context of any thread within the application.

This means that an asynchronous signal such as SIGINT (generated by kbd:[Ctrl + C]) could occur in the context (i.e., could interrupt the execution of) any thread in the process. This makes sense, since the signal wasn't caused by any specific thread. But it can add complexity to the application if the signal requires a significant amount of time to handle and the thread that was interrupted is time-sensitive. Therefore, it is fairly common for the main application to block all signals during thread creation because the new threads will inherit the block mask. Then each individual thread can decide which signals it wants to allow and unblock just those.

Any thread can call `sigprocmask()` to change the block mask for the entire process, but given the above strategy, it makes sense for only `main()` to use it. Similarly, `pthread_sigmask()` allows each thread to change its own block mask, and a given thread may want to unblock signals it will respond to.

Example 1. Thread creation pseudo-code

Inside the main application thread (or the one used to create other threads):

1. create **blocked** `sigset_t` containing signals that should always be blocked;
2. create **all** `sigset_t` containing all signals;
3. to create a new thread:
    a. set process signal mask to block **all** signals (use `sigprocmask()` or `pthread_sigmask()`);
    b. create new thread (it inherits the block mask);
    c. set process signal mask to block **blocked** signals;

Meanwhile, inside the new thread:

1. perform application-define tasks;
2. when ready to receive a given signal(s), unblock those using *only* `pthread_sigmask()`;
3. block them again when the thread no longer wishes to be interrupted using the same function;

The trade-off in blocking and unblocking signals is that asynchronous signals are deliverable to any thread with the signal unblocked. If there are many such threads, signals may be responded to quickly than if there is only one such thread.

It is common for the main thread to be the only thread leaving asynchronous signals unblocked, while all threads block every signal except the ones they specifically want to accept.

Race Conditions

A common problem when writing multithreaded applications is the *race condition*. This is a situation in which the scheduling of two threads can cause errors in application logic. This means the operation of the code is *non-deterministic*. Or to rephrase that in English, "the code is unreliable".

The following code sample is a prime example of a race condition using two threads:

Race condition example code

```
#include <stdio.h>
#include <pthread.h>

void *problem(void *arg)
{
    int *val = arg;

    int i = *val;
    i += 10;
    *val = i;

    return NULL;
}

int main(void)
{
    int data = 99;
    pthread_t alpha, bravo;

    pthread_create(&alpha, NULL, problem, &data);
    pthread_create(&bravo, NULL, problem, &data);
    pthread_join(alpha, NULL);
    pthread_join(bravo, NULL);

    printf("data = %d\n", data);
}
```

The problem is that both threads could start and when one is about to execute line 8, the scheduler could switch to the other one. This means thread 1 could retrieve the value and get "99", then the scheduler could switch to thread 2. Thread 2 would also read the value "99". No matter which one increments it and writes it back out, the result will be "109" instead of the correct "119".

To see this in action, try running the program 1,000 or so times.

Summary

Students have learned what POSIX threads are and have been introduced to concepts like deadlocks and synchronization. The next topic will delve into race condition mitigation.

- The `pthreads` API
- Waiting for another thread to terminate
- Signal handling in a multithreaded environment
- Race conditions and their effects

Exercises

Exercise 1: `dircounter.c`

Write a program that takes any number of directory names as command-line arguments. The program should spawn a thread for each directory that counts the number of files in it. Then, the program should print each directory, plus its count of files. Finally, it should print a total count of files seen across all threads.

Sample Output

```
$ ./dircounter /dev /etc

/etc  83
/dev  332
Total: 415
```

Chapter 13. Threads: Synchronization

Students will learn how to identify and correct race conditions in multithreaded applications using locking and condition variables.

- Shared resources and critical sections
- Mutexes
- Locking and unlocking a mutex
- Condition variables
- Signaling and waiting on condition variables
- Dynamically initialized mutexes
- Dynamically initialized condition variables
- Other synchronization primitives

Introduction

The previous chapter introduced the idea of *race conditions*. Race conditions are program errors which are timing related and thus create non-deterministic results in an application. This chapter will discuss how to synchronize threads to correct race conditions.

Here is the sample code from the previous topic that demonstrated a race condition:

Fixed race condition

```
#include <stdio.h>
#include <pthread.h>

// Added this line and the two inside fixed()
pthread_mutex_t lock_for_data = PTHREAD_MUTEX_INITIALIZER;

void *fixed(void *arg)
{
    int *val = arg;

    pthread_mutex_lock(&lock_for_data);
    int i = *val;
    i += 10;
    *val = i;
    pthread_mutex_unlock(&lock_for_data);

    return NULL;
}

int main(void)
{
    int data = 99;
    pthread_t alpha, bravo;

    pthread_create(&alpha, NULL, fixed, &data);
    pthread_create(&bravo, NULL, fixed, &data);
    pthread_join(alpha, NULL);
    pthread_join(bravo, NULL);

    printf("data = %d\n", data);
}
```

Race conditions occur when concurrently executing code attempts to access shared data. The above code example could play out like this:

Table 6. Ideal sequence

1
2
3
4
5
6
	`int i = *val;`
	`i += 10;`
	`*val = i;`
	`.`
	`.`
	`.`
		`.`
		`.`
		`.`
		`int i = *val;`
		`i += 10;`
		`*val = i;`

Unfortunately, that order is not guaranteed. The following sequence is also possible, and it fails to increment `i` by 20:

Table 7. Failing sequence

1
2
3
4
5
6
	`int i = *val;`
	`.`
	`.`
	`.`
	`i += 10;`
	`*val = i;`
		`.`
		`int i = *val;`
		`i += 10;`
		`*val = i;`
		`.`
		`.`

Mutual Exclusion Locks

As shown in the previous section, the problem is that multiple threads could be accessing the value at the same time. There is a critical section of code, i.e., code that should be treated atomically as a single transaction, which must not be interrupted or executed simultaneously. That code should be treated as a single transaction.

This is accomplished using a *mutex lock*, or often just *mutex*. (This is short for *mutual exclusion*, as only one thread may access the critical code at a time and its *mutual* friends are *excluded*.)

Mutual exclusion is used in the real world all the time!

- Our traffic lights and street signs are designed to prevent two vehicles from attempting to occupy the same space at the same time (crash!). That's mutual exclusion (it is also queueing to decide what order they enter the space, but mutual exclusion is used to prevent two vehicles at once).
- Many agencies have a "take a ticket" system in which the visitor collects a number when they arrive and they are called to the counter when it's their turn. That's mutual exclusion (it is also queueing, but mutual exclusion is used to prevent two visitors from taking the same number).

How is this going to be done in software, though? This requires some way for one thread to enter the critical section of code but block other threads from doing so. To add complexity, the system may have one core, four cores, or an even higher number. Isn't this going to get complicated?

Fortunately, the complicated aspects have already been worked out by someone else! The programmer simply needs to place an instruction at the beginning of the critical code to "claim" the mutex lock, then "release" the lock when it exits the critical code. The functions that lock and unlock the mutex are shown below.

```
#include <pthread.h>

// Initializing at allocation time
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

// initializing at runtime
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);

// Always delete mutexes when they're no longer needed
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

The opaque data structure this time is `pthread_mutex_t`. There are multiple ways of initializing the mutex depending on how it is being allocated.

- When it is a global or static variable, simply assign `PTHREAD_MUTEX_INITIALIZER` to it. (This is actually a flag value; the initialization takes place transparently the first time the mutex is locked.)
- When it's a member of a structure, such as one allocated dynamically, call the `pthread_mutex_init()` function and pass it the address of the mutex. This function has a second parameter that can be used to control various options on the mutex. (This module will not be examining these options.)
- Whichever technique is used, do not forget to destroy dynamically allocated mutex locks when they are no longer needed! This should only be done when the lock is not being held (the `pthread_mutex_destroy` function does not check for this). Global and static locks do not need to be destroyed (although doing so is not an error).

Keep to the pattern of one—and only one—thread initializing mutex locks!

```
#include <pthread.h>

// Acquiring and releasing the mutex lock
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

The `pthread_mutex_lock()` function is placed above the critical code and it claims the lock.

- It does so in an atomic way, that guarantees no other cpu can claim the lock at the exact same instant. (How it does this is quite fascinating, considering that two cpus could be executing the ***exact same instruction*** and the ***exact same time***, and this still works. However, that's beyond the scope of this material.)

Place a call to `pthread_mutex_unlock()` at the end of the critical code to release the lock. It is very important to release every lock acquired! If a thread terminates without releasing a lock, ***all other threads will block waiting for the lock to be released*** and there's no way to correct this!

Some Gotchas When Using Mutex Locks

These are listed in no particular order. Some have clear consequences and others are more obscure. This chapter will look at a few of these, but not all of them.

1. If using threads, be careful if your application needs to `fork()`.

   The child will only have a single thread (the main thread), yet will receive copies of all of the locks, so it any mutex was locked at the time of the `fork()`, things get difficult. The `pthread_atfork()` function can help with this.

2. The thread that owns a lock is the only thread that should unlock it.

Technically, the mutex lock doesn't enforce ownership in terms of unlocking, but an application that allows one thread to release a lock obtained by another thread is just asking for trouble!

3. Locks protect a code path, not a data element.

   This means an application can have multiple locks, each protecting code that tries to modify shared data. For example, the head of a linked list might contain a lock so that attempts to modify the linked list by two different threads are serialized so that the list doesn't become corrupted.

4. Locks are a tool to correct synchronization issues — the programmer must spot which pieces of code are critical sections!
5. Be careful that functions that obtain locks always release them before returning!
6. The implementation of mutex locks may cause a memory leak if they are not destroyed.
7. Don't use or destroy a mutex that hasn't been initialized, and don't use a mutex that has been destroyed.
8. Don't lock a mutex twice in a single thread without unlocking it in-between.

   A mutex lock only that it has or has not been claimed, it doesn't know by which thread. So any thread that tries to obtain a lock will wait if the lock is already claimed (even if its the same thread).

9. Watch out for deadlocks.

Semaphores

Semaphores are another synchronization primtive. It is initialized to some value, and threads can then `sem_wait()` or `sem_post()` to lower or raise the value. The key is that `sem_wait()` blocks if the semaphore contains zero; otherwise, it decrements the value and returns immediately. The `sem_post()` function is used when a thread finishes with a resource and wants to release it.

Because semaphores contain a value and provide their own locking around the increment and decrement of that value, they are a more sophisticated structure than a mutex lock. This means they have slightly more overhead than a simple mutex lock, but are easier for application programmers to use in many cases.

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);

int sem_post(sem_t *sem);
```

Unlike with mutex locks, `sem_wait()` and `sem_post()` can be invoked by different threads! In fact, this is a fairly common case. If one considers the value stored in the semaphore to be the number of resources available in some pool of resources, then one thread may allocate such a resource and release it when it is done. But it's also possible for a thread to allocate a resource and then pass the information to another thread for use, leaving it to that other thread to release the resource when it is no longer needed.

Semaphore example

```
#include <stdio.h>
#include <pthread.h>
#include <signal.h>
#include <semaphore.h>
#include <unistd.h>

// Unlike mutex, cannot initialize here.
sem_t s;

void handler(int unused)
{
    (void)unused;
```

```
    // Release the semaphore by incrementing it by one.
    sem_post(&s);
}

void *thread(void *unused)
{
    (void)unused;

    // Wait for the semaphore to post...
    sem_wait(&s);
    puts("Waiting until a signal releases...");

    return NULL;
}

int main()
{
    // Shared between threads in the current process, initial value 0
    int err = sem_init(&s, 0, 0);

    if (err < 0) {
        perror("Could not create semaphore");
        return 1;
    }

    struct sigaction sig_int = { .sa_handler = handler };
    sigaction(SIGINT, &sig_int, NULL);

    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);

    // Process will exit when all threads have exited
    pthread_exit(NULL);
}
```

Condition Variables

Mutex locks and semaphores provide a way for different sections of code to synchronize their execution of critical sections of code. Condition variables allow a set of threads to sleep until specifically awoken.

The API allows either one or all threads waiting on a condition variable to be woken up. If a program only wakes one thread, the operating system will decide which thread to wake up. Threads do not wake other threads directory. Instead, a thread "signals" the condition variable, which then will wake up one (or all) threads that are sleeping inside the condition variable. (This use of the word "signal" is not referring to the `signal` facility within the operating system. It's simply a synonym for "trigger" or "activate", but the "signal" is used because it is part of the function name, `pthread_cond_signal()`.)

Condition variables are also used with a mutex and a loop, so when woken up, they have to check some state within a critical section of code before continuing. If a thread needs to be woken up outside of a critical section, there are other ways to do this in POSIX. Threads sleeping inside a condition variable are woken up by another thread calling `pthread_cond_broadcast()` to wake up all threads, or `pthread_cond_signal()` to wake up a single thread.

Occasionally, a waiting thread may appear to wake up for no reason. This is called a *spurious wakeup*. This is one reason why loops and a mutex are used to check the application state before simply assuming that all requirements are met. Why do spurious wakeups occur? For performance reasons. On multi-core systems, it is possible that a race condition could cause a wake-up request to be unnoticed. The kernel may not detect this lost wake-up call but can detect when it might occur. To avoid the potentially lost wake-up, the sleeping thread is woken up so that the program code can test the condition again.

```
#include <pthread.h>

// Initialize at allocation time
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);

// Call this to wait for a ..._signal() or ..._broadcast() call.
// This function will atomically release the lock, sleep, then reacquire the lock.
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

// Notify other threads to wake up
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

Putting it All Together

This section will provide an example of an array that is being modified by multiple threads. An exhaustive discussion of application thread design can easily be an entire course by itself, so this chapter will only look at a few simple cases. To prevent the array from being corrupted by simultaneous updates, it will be protected by a mutex lock.

First, here's the existing array code (NOT thread-safe):

Array code, first try (NOT thread-safe)

```
enum { STACK_SIZE=20 };

size_t count;
double values[STACK_SIZE];

void push(double v)
{
    if (count == STACK_SIZE) {
        fprintf(stderr, "Stack Overflow\n");
        return;
    }

    values[count++] = v;
}

double pop(void)
{
    if (count == 0) {
        return NAN;
    }
    return values[--count];
}

bool is_empty(void)
{
    return count == 0;
}
```

It is not thread-safe because if two threads try to modify the array at the same time (calling some combination of `push()` or `pop()`), the array can become corrupted.

To make this code thread-safe, the critical section(s) of code must be identified. These critical sections are those that modify the data structure, or somehow rely on a consistent view of the data structure. In the example code, all three functions are critical sections (a modification occurring in one thread could affect

the results of other threads). (It's possible for two threads to call `is_empty()` at the same time without corruption, but if `is_empty()` is called while one of the other functions is being executed, then `is_empty()` could fail. Therefore, it needs to be considered a single atomic operation.)

Look at the next example. This code has added mutex locks to control which code sections can be executed simultaneously. Clearly, `push()` and `is_empty()` cannot now be executed simultaneously, but `pop()` can be executed while either of the other two are running, so the problem still exists.

Array code, second try (NOT thread-safe)

```
#include <pthread.h>

enum { STACK_SIZE=20 };

size_t count;
double values[STACK_SIZE];

pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;

void push(double v)
{
    pthread_mutex_lock(&m1);
    if (count == STACK_SIZE) {
        fprintf(stderr, "Stack Overflow\n");
        return;
    }
    values[count++] = v;
    pthread_mutex_unlock(&m1);
}

double pop(void)
{
    pthread_mutex_lock(&m2);
    if (count == 0) {
        return NAN;
    }
    return values[--count];
    pthread_mutex_unlock(&m2);
}

bool is_empty(void)
{
    pthread_mutex_lock(&m1);
    return count == 0;
    pthread_mutex_unlock(&m1);
}
```

(There is a second error: the `return` statement inside `is_empty()` prevents the mutex from being unlocked. Fortunately, modern compilers will complain—loudly—about unreachable code after the `return` statement, so this error is unlikely to slip through. But if the `return` statement were buried inside a nest `if` statement, the compiler wouldn't be able to provide a warning and a severe bug would go uncaught!)

The solution in this case is obvious—use a single lock. Be warned, however: the more places that require locking, the more lock contention there will be. And using a single lock when multiple locks ***can*** be used, also increase lock contention. This will affect scalability of the application (more threads means more lock contention, which results in additional threads not increasing throughput by a proportional amount).

The next example fixes those errors and the code is now thread-safe.

Final array code (thread-safe)

```
#include <pthread.h>

enum { STACK_SIZE=20 };

size_t count;
double values[STACK_SIZE];

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void push(double v)
{
    pthread_mutex_lock(&m);
    if (count == STACK_SIZE) {
        fprintf(stderr, "Stack Overflow\n");
        return;
    }
    values[count++] = v;
    pthread_mutex_unlock(&m);
}

double pop(void)
{
    pthread_mutex_lock(&m);
    if (count == 0) {
        return NAN;
    }
    double rv = values[--count];
    pthread_mutex_unlock(&m);
    return rv;
}

bool is_empty(void)
{
    pthread_mutex_lock(&m);
    bool rv = (count == 0);
    pthread_mutex_unlock(&m);
    return rv;
}
```

While the code is thread-safe, there are still two issues:

1. By the time `is_empty()` returns the result, it could already be out of date! Many thread-safe data structures remove functions that return sizes or deprecate them.

2. There is no protection against underflow or overflow (popping or pushing too many values).

The second problem can be tackled using semaphores. When a thread tries to push data onto the array and it's full, it could wait for another thread to release some space. Similarly, when a thread tries to pop some data from the array, if there is none, it can wait for another thread to add some.

To make this solution more generic, move the locking primitives into an opaque data structure that holds the array. This listing shows the new structure and a sample of using it inside `main()`. It is not complete yet, but discussion will start here:

Array with primitives embedded

```
#include <stdlib.h>
#include <pthread.h>

// Abbreviated opaque type
typedef struct {
    double *values;
    size_t sz;
    size_t cap;
    pthread_mutex_t m;
} array;

array* stack_create(int size)
{
    array *tmp = malloc(sizeof(*tmp));
    if (!tmp) {
        return NULL;
    }

    // add return value error checking
    tmp->sz = 0;
    tmp->cap = size;
    tmp->values = malloc(tmp->cap * sizeof(tmp->values));
    if (!tmp->values) {
        free(tmp);
        return NULL;
    }

    pthread_mutex_init(&tmp->m, NULL);
    return tmp;
}

void stack_destroy(array *s)
{
    if (!s) {
```

```
        return;
    }

    free(s->values);
    pthread_mutex_destroy(&s->m);
    free(s);
}

void push(array *s, double v);
double pop(array *s);

int main(void)
{
    array_t *s1 = stack_create(10);
    array_t *s2 = stack_create(10);

    push(s1, 3.1415);
    push(s2, pop(s1));

    stack_destroy(s1);
    stack_destroy(s2);
}
```

Array with `push()` and `pop()`

```
#include <pthread.h>

// Abbreviated opaque type
typedef struct {
    double *values;
    size_t sz;
    size_t cap;
    pthread_mutex_t m;
    pthread_cond_t cv;
} array;

void push(array *s, double v)
{
    if (!s) {
        return;
    }

    pthread_mutex_lock(&s->m);

    while (s->count == s->size) {
        pthread_cond_wait(&s->cv, &s->m);
```

```
    }

    s->values[(s->count)++] = v;
    pthread_cond_signal(&s->cv);
    pthread_mutex_unlock(&s->m);
}

double pop(array *s)
{
    if (!s) {
        return NAN;
    }

    pthread_mutex_lock(&s->m);

    if (s->count == 0) {
        pthread_cond_wait(&s->cv, &s->m);
    }

    double rv = s->values[--(s->count)];
    pthread_cond_signal(&s->cv);
    pthread_mutex_unlock(&s->m);

    return rv;
}
```

The above code uses `pthread_cond_signal()` for pushing to wake up popping, but `pthread_cond_broadcast()` for popping to wake up pushing. Does it matter? The former only wakes up a single waiting thread while the latter wakes up all waiting threads. Is there any benefit to waking up *all* waiting threads? Probably not. Both were used in this example as a hook for discussion, but waking up a single thread would work in both locations.

In all of the code written so far, the management of the count is done “manually”, so to speak. Now the discussion turns towards using semaphores instead, in which the capacity of the stack is stored inside the semaphore.

However, remember that semaphores only go to sleep waiting for zero. The programmer cannot code something that says, "wait for the semaphore to become its maximum value", for example. This means the solution will need two semaphores: one that counts the number of items in the stack, and another that counts the number of available slots in the stack. Then both can use the “wait for zero” semantics that semaphores provide.

Array switched to use semaphores

```
#include <pthread.h>
#include <semaphore.h>

// Abbreviated opaque type
typedef struct {
    double *values;
    size_t sz, cap;
    sem_t items, slots;
} array;

void push(array *s, double v)
{
    if (!s) {
        return;
    }

    // Wait until a slot opens up
    sem_wait(&s->slots);
    s->values[(s->count)++] = v;
    // Notify others that there's an item now
    sem_post(&s->items);
}

double pop(array *s)
{
    if (!s) {
        return NAN;
    }

    sem_wait(&s->items);
    double rv = s->values[--(s->count)];
    sem_post(&s->slots);
    return rv;
}
```

The above code manages the buffer full and buffer empty conditions using semaphores, but it has lost the protection of the critical sections of code!

Adding the locks back in around the critical sections provides the full solution:

Array using semaphores and mutexes

```
#include <pthread.h>
#include <semaphore.h>

typedef struct {
    double *values;
    int count, size;
    size_t sz, cap;
    pthread_mutex_t m;
    sem_t items, slots;
} array;

void push(array *s, double v)
{
    if (!s) {
        return;
    }

    sem_wait(&s->slots);

    pthread_mutex_lock(&s->m);
    s->values[s->count++] = v;
    pthread_mutex_unlock(&s->m);

    sem_post(&s->items);
}

double pop(array *s)
{
    if (!s) {
        return NAN;
    }

    sem_wait(&s->items);

    pthread_mutex_lock(&s->m);
    double rv = s->values[--s->count];
    pthread_mutex_unlock(&s->m);

    sem_post(&s->slots);
    return rv;
}
```

As mentioned previously, there is much more to be discussed concerning thread synchronization. There are *barriers*, *reader-writer locks*, and more. There are numerous Internet resources available that discuss

multithreading in detail.

Summary

Students have learned how synchronization affects multithreaded applications, including how to identify and correct race conditions using locking and condition variables.

- Shared resources and critical sections
- Initializing synchronization primitives
- Locking and unlocking
- Condition variables
- Signaling and waiting on condition variables
- Use of semaphores

Exercises

Exercise 1: `pongping.c`

Revisit the `pingpong` program from an earlier chapter. Implement a `pongping` program in which one thread prints `pong!\t` and another thread prints `ping!\n`. There should be a ½-second delay between each message.

Sample output

```
$ ./pongping
pong!   ping!
pong!   ping!
pong!   ^C
$
```

Chapter 14. Interprocess Communication: Introduction and Overview

Students will learn how POSIX systems support various inter-process communication. This chapter is strictly an overview of the available features and a discussion of when each might be most appropriately used. More details on both are covered in following chapters.

- Categorizing IPC
- Choosing an IPC mechanism

Introduction

Interprocess communication, more commonly known as *IPC*, is the name for a broad array of facilities that allow processes to communicate with each other.

One could consider ordinary file IO to be a form of IPC, and while that's appropriate on some level, this chapter is focused on the *formal* IPC facilities.

IPC mechanisms generally fall into two categories: those that share memory, and those that pass messages.

- Sharing memory is the the fastest way for two processes to communicate data back and forth.

  One process writes into the memory and there is no delay before another can read what was written. The complexity with using shared memory is in coordinating which process can can read and/or write the memory at any given time.

- Message passing provides a way to *enqueue* data from one or more processes to be later retrieved and operated on by one or more processes.

  Message passing involves the overhead of invoking a "coordinator" who acts to add or remove data from the queue in a thread-safe manner. Interacting with the coordinator takes time, particularly when the coordinator is code that lives inside the operating system (because of mode switches and potential context switches).

Some of the IPC mechanisms to be discussed overlap these boundaries and could thus be considered as members of both, but not exclusively to either one.

What Options Are Available?

Two forms of IPC have already been discussed: file IO and locking, and signals. The next few chapters cover what are considered the ***formal IPC*** mechanisms of POSIX operating systems.

NOTE

Not all POSIX operating systems implement all mechanisms (POSIX 1003.1 Appendix 9 allows conforming systems to optionally support particular facilities), but most POSIX systems support all of these mechanisms at least partially. For example, OS X does not support POSIX unnamed semaphores, but does support named semaphores.

The available techniques are:

1. Pipes and FIFOs (these are very similar in implementation, but creation is different)
2. UNIX domain sockets (operation is similar to pipes)
3. Message Queues
4. Semaphores
5. Shared memory
6. Memory mapped files (a specific type of shared memory)

This module focuses on the POSIX approach for maximum portability of application code. Summaries of the differences compared to SysV IPC will be made, where appropriate.

Overview of Each Type

1. Pipes and FIFOs

   Pipes are the mechanism used by the shell to implement command line pipes via the `|` operation. The command line `ps -eaf | grep bash | sort | uniq -c` demonstrates the use of pipes three times. A FIFO (First In, First Out) is sometimes called a ***named pipe***. They function just like pipes, but have a name within the file system, which allows two unrelated processes to communicate (unnamed pipes, like those used by the shell, require a common ancestor to be useful).

2. UNIX domain sockets

   These are essentially network sockets with the "network" part removed so that they only work on the same machine. SInce they require a rendezvous point (a name within the file system), their use is similar to FIFOs, except that as sockets, they have the capability of sending out-of-band data.

3. Message Queues

   As their name implies, these are queues of messages in which all messages and metadata are kept within the kernel. Messages are added to the queue or removed from the queue by the kernel, which uses locks to ensure a consistent view of the queue at any given time. A process can add to the queue, remove from the queue, or both, during its lifetime.

4. Semaphores

   POSIX unnamed semaphores were discussed in a previous chapter. Their coverage in this portion of the module will focus primarily on differences between POSIX semaphores and SysV semaphores.

5. Shared memory

   This is literally ***shared virtual memory*** between processes. The memory management unit in the system hardware is programmed such that virtual addresses for two processes point to the same physical frame of memory. When each process accesses its virtual memory location, it is seeing the shared RAM.

6. Memory mapped files (a specific type of shared memory)

   These are essentially shared memory, but data blocks within a filesystem are used as backing store instead of the system's swap space. Otherwise, all functional aspects are the same as shared memory.

Summary

Students have learned what the various IPC mechanisms are in POSIX. Further exploration of each method is covered in later chapters.

- What mechanisms are available
- Overview of each one

Chapter 15. IPC: Pipes and FIFOs

Students will learn about named and unnamed pipes and how (and when) each can be used.

- Creating and using pipes
- FIFOs
- Other options and features

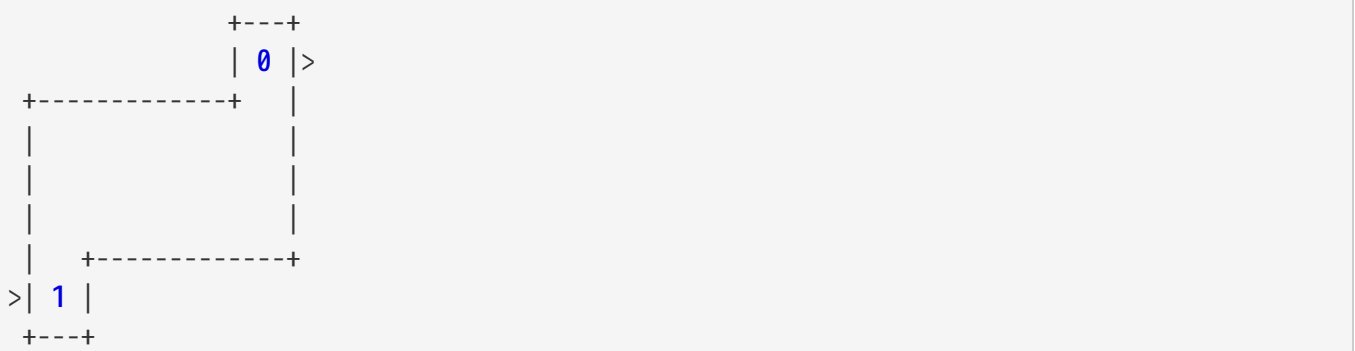
Introduction

Pipes are one-way data transfer conduits. They are implemented using a fixed size buffer and are accessed as file descriptors (meaning that normal file IO operations can be used).

Each pipe consists of a buffer, with two file descriptors attached to it. One file descriptor is opened for reading and the other is opened for writing. From the application's perspective, the first one should be read from (as stdin would be) and the second one should be written to (as stdout would be).

Reading from a pipe will block unless there is data available, and writing to a pipe will block unless there is space available to hold the data.

Representation of a single pipe buffer

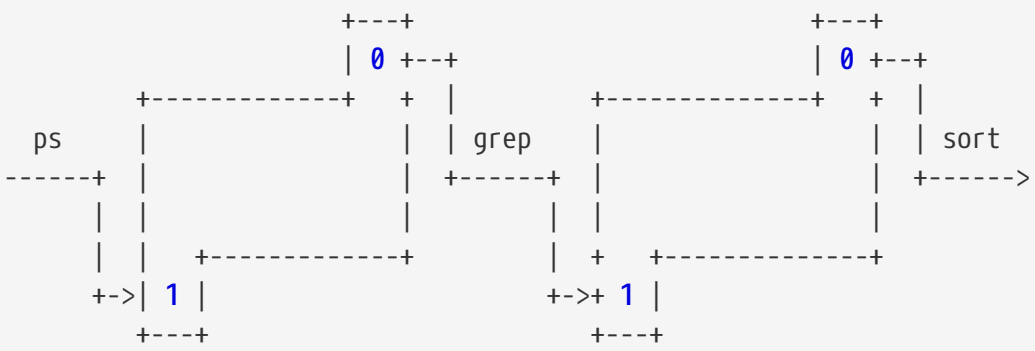


The size of the buffer is system-dependent, but common values include 512 bytes and 4096 bytes. The actual size doesn't affect the functionality, but may affect an application's performance when accessing pipes. (Most vendors tune the pipe size for performance and pick an appropriate size.)

The Linux man page for `pipe(7)` says that Linux currently has a default pipe size of 16 4KB pages, or 64K. (See the man page for various system-wide and per-user limits.) The output of Bash's `ulimit -p` shows 4KB. Which is correct? Likely both: the size of the pipe can be 64KB, but Bash can be reporting the result of `sysconf(PIPE_BUF)` which is a kernel compile-time constant that recommends the maximum atomic write size.

The following picture shows the effect of executing `ps -eaf | grep bash | sort`:

The result of running three commands with pipes



The buffer used by the pipe is essentially a circular queue. Data goes in where the **0** appears in the picture, and data comes out where the **1** appears. When the buffer fills up, no more data is accepted and the writer blocks. When the buffer is empty, no more data can be read and the reader blocks. Pipes allow both concurrency (the appearance of overlapping execution) and parallelism (actual overlapping execution).

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

One of the beautiful features of pipes is how simple they are! A process calls `pipe()` and a new buffer is allocated. The side of the buffer that accepts data is in array element zero, and the side of the buffer that provides data is in array element one. This directly maps to `stdin` and `stdout`—an application writes to `pipefd[0]` (what the buffer sees as `stdin`) and reads from `pipefd[1]` (what the buffer sees as `stdout`).

One glaring problems with pipes is that file descriptors are unique to a given process. How can two separate processes, such as `ps` and `grep`, share the pipe buffer? The answer? The pipe is created, then `fork()` is called to create a child. Since children inherit the file descriptors of the parent, both processes have the pipe. One of them can close `pipefd[1]` and write to `pipefd[0]`, while the other can close `pipefd[0]` and read from `pipefd[1]`. Extend this for a third processes and the result is the picture shown above.

Writing to a pipe which all readers have closed will cause the writer to receive `SIGPIPE` (or if the signal is being ignored, writes will return `-1` and set `errno` to `EPIPE`). Trying to read on a pipe in which all writers have been closed results in the expected `EOF`.

Example of using unnamed pipes

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

```c
int main(void)
{
    int io[2];
    if (pipe(io) < 0) {
        perror("Unable to create pipe");
        exit(1);
    }

    pid_t pid = fork();
    if (pid < 0) {
        perror("Unable to fork");
        return 1;
    } else if (pid == 0) {
        // Child
        // Close the writing side
        close(io[1]);

        char buf[8];

        ssize_t received;
        while ((received = read(io[0], buf, sizeof(buf)-1)) > 0) {
            buf[received] = '\0';
            printf("%s", buf);
        }

        if (received < 0) {
            perror("Unable to read");
        } else {
            putchar('\n');
        }

        close(io[0]);
    } else {
        // Parent
        // Close the reading side
        close(io[0]);

        const char *msg = "Hello there";

        ssize_t written = write(io[1], msg, strlen(msg));
        if (written < 0) {
            perror("Unable to write");
        }

        close(io[1]);
    }
```

```
}
```

FIFOs (a.k.a. Named Pipes)

FIFOs work very similarly, but instead of requiring a common ancestor, as unnamed pipes do, FIFOs have a name within the file system (created using `mkfifo()` or the `mkfifo` command).

- This allows process(es) running under one set of credentials to write to a FIFO while another process(es) running with different credentials read from the FIFO.
- (This presumes that file permissions will permit that access.)

FIFOs have another advantage: because they are opened like ordinary files (using the `open()` system call), other options can be passed at that time.

- The most common such option is `O_NONBLOCK`, which turns on the nonblocking mode mentioned above. (See the description in the next section, below.)

Because named pipes don't require a common ancestor, they also don't require `fork()`. This means ordinary shell commands can use them directly. In the example below, either shell session can create the FIFO. Once created, either one can execute first and will block waiting for the other.

Table 8. Example of using named pipes

```
mkfifo the_fifo
ps -eaf > the_fifo  # this will block
.
```

```
.
.
grep bash < the_fifo
```

Optional Configuration

Multiple readers and multiple writers

It is possible to have multiple readers and multiple writers.

- This is not practical for many applications because there is no predicting which reader will read data from which writer, nor how much might be read in a single system call.
- In other words, if three different writers each write two lines of data, it is possible for six readers to each read one line!
- Or for two readers to read two lines and four lines, respectively.
- Or any combination.

POSIX does require that writes to a pipe (and to files, as well) must be performed atomically up to a specified minimum size (512 bytes on most systems).

- This guarantees that two processes writing at the exact same instant cannot have their data interspersed or overwritten within the buffer.
- However, there are no constraints on how the data may be read.
- And no requirement that a process that writes in a loop must finish its loop before another can write.

Blocking mode (default) vs. nonblocking mode

Normally, a FIFO blocks processes that open it until both a reader and a writer have opened it, thus basic synchronization is built in. However, FIFOs can be opened in *nonblocking* mode. In nonblocking mode:

- Opening with O_RDONLY immediately succeeds, even if no writer exists yet.
- Opening with O_WRONLY immediately fails (with `ENXIO`) if there is no reader.
- Opening with O_RDWR results in undefined behaviour in POSIX (although Linux allows this and treats the processes as both reader and writer — be careful of deadlocks!).

Any file descriptor on which it is possible to block can ostensibly be set in nonblocking mode using the `fcntl()` system call (see the F_SETFL command under **File status flags** in the man page). However:

- Nonblocking mode has no effect on regular files and block devices (although no error is thrown when the mode change is attempted).
- It does work for IO devices such as:
  - terminals,

- tape drives,
- network sockets, and
- unnamed and named pipes

In addition, reading or writing on a file descriptor in nonblocking mode will immediately return — either it succeeds and data is read/written, or the attempt fails (returns `-1`) and `errno` is set to `EAGAIN`.

The ANSI C popen() Function

For simple situations, it may be plausible to use `popen()`. This is an ANSI C function that forks, executes a command line in the child, and provides the output of that process as a `FILE *` that the application can read from or write to.

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

Similar to opening a file with `fopen()`, the `popen()` function requires a command line instead of a filename, and a string representing the type of IO the application wants to perform.

This example is a derivative of one shown much earlier in the module. (Error handling has been omitted to make the algorithm clearer.)

Example of using `popen()`

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

int main(int argc, char *argv[])
{
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <command> <output_file>\n", argv[0]);
        return 1;
    }

    FILE *in = popen(argv[1], "r");
    if (!in) {
        perror("popen");
        return 2;
    }

    FILE *out = fopen(argv[2], "w");
    if (!out) {
        perror(argv[2]);
        pclose(in);
        return 3;
    }

    // Size chosen arbitrarily
    char buf[1024];
```

```
    for (;;) {
        size_t received = fread(buf, sizeof(buf[0]), sizeof(buf), in);
        if (ferror(in)) {
            perror("Error when reading");
            fclose(out);
            pclose(in);
            return 5;
        }

        if (received == 0) {
            break;
        }

        size_t written = fwrite(buf, sizeof(buf[0]), received, out);
        if (written != received || ferror(out)) {
            perror("Unable to write");
            fclose(out);
            pclose(in);
            return 4;
        }
    }


    fclose(out);
    pclose(in);
}
```

There are numerous issues to be aware of when using `popen()`:

- Because it executes a command line, it creates a dependency on other installed programs.
- The `popen()` function invokes the `/bin/sh` shell using the `-c` option and passes in the command string for execution.
  - This means vulnerabilities in the shell are now vulnerabilities of this program.
  - Performance is reduced as the shell has significant startup overhead.
  - Memory use and cpu use are higher than if the application did the work directly.
- Failure to execute the shell is indistinguishable from failure to execute the command.
- `popen()` can fail when any of `fork()`, `exec()`, or `wait()` fails.
- The returned `FILE *` must be closed via `pclose()` to avoid zombie processes.
- All of the same issues that apply to `system()` apply to `popen()`.

Some of the "issues" listed above could be considered features:

- Because the command is executed by a shell, full shell syntax is available:
  - Wildcards will be expanded.
  - Variable substitution, command substitution, and arithmetic substitution will work (if the system's shell supports those features).
  - Pipes will work.
  - IO redirection will work.
  - And so on.
- Provides easy access for application that want to allow the user to "shell out".
  - This is a technique in which an interactive program allows the user to generate an interactive shell prompt.
  - This is done in `vi` or `vim` using `:sh`, from within `mail` by using `!!`, and so on.

Summary

Students have learned about named and unnamed pipes and the particulars about their usage. The use of unnamed and named pipes was compared.

- Creating and using unnamed pipes
- FIFOs
- Nonblocking mode
- Using `popen()`

Chapter 16. IPC: Unix Domain Sockets

Students will learn the POSIX socket API and how to create both STREAM- and DATAGRAM-oriented UNIX domain sockets, and the differences between them.

- Socket types and domains
- UNIX domain stream sockets
- Creating and binding a socket
- Accepting incoming connections
- How a client makes outbound connection requests
- Differences between connected and connectionless sockets

Introduction

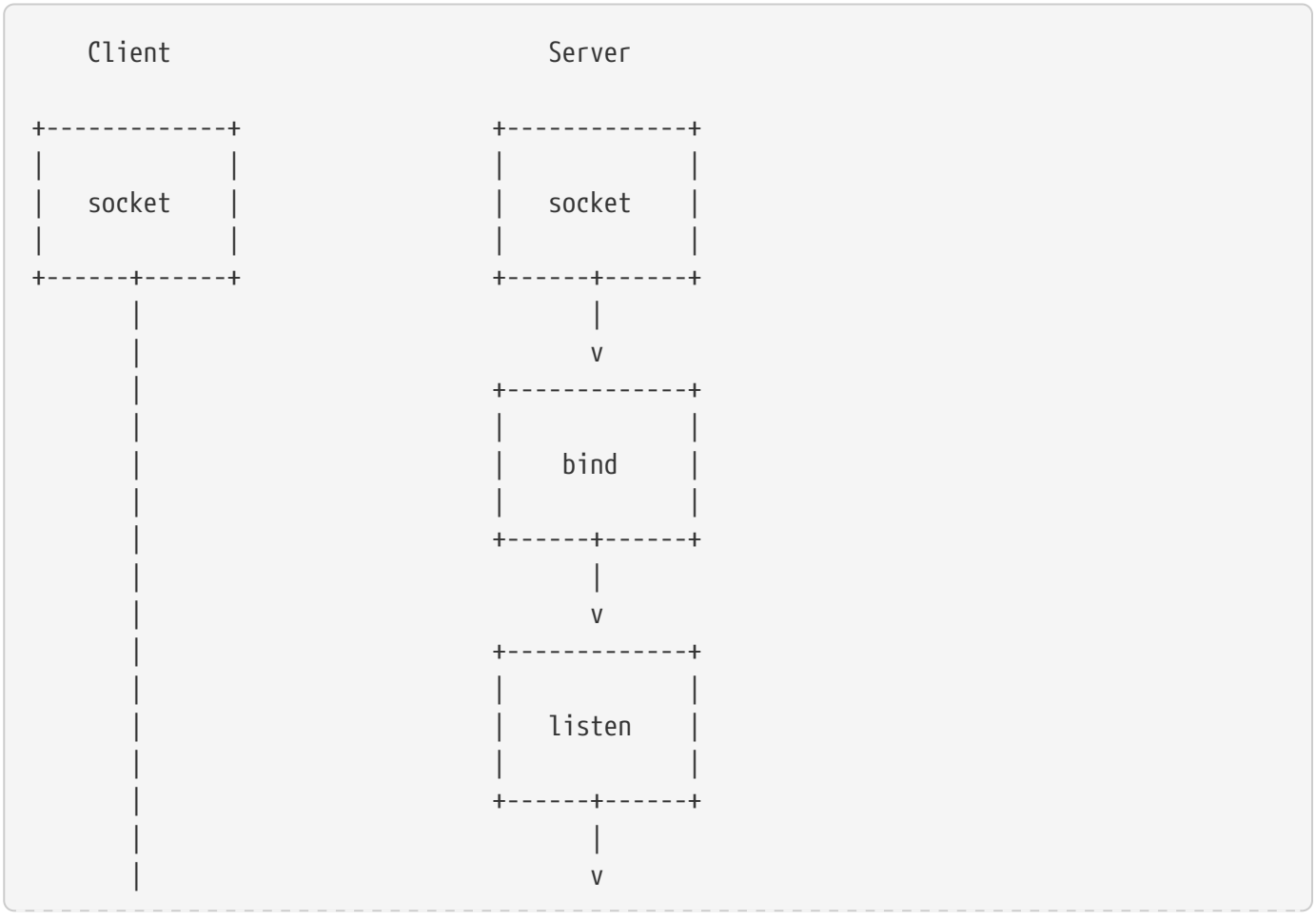
Sockets are an API that goes back to the earliest days of Unix networking. The API allows for easy client-server communications between two processes and, potentially, between two computers on a network. This module will cover local sockets, known as *Unix domain sockets*; a later module will cover intersystem network programming.

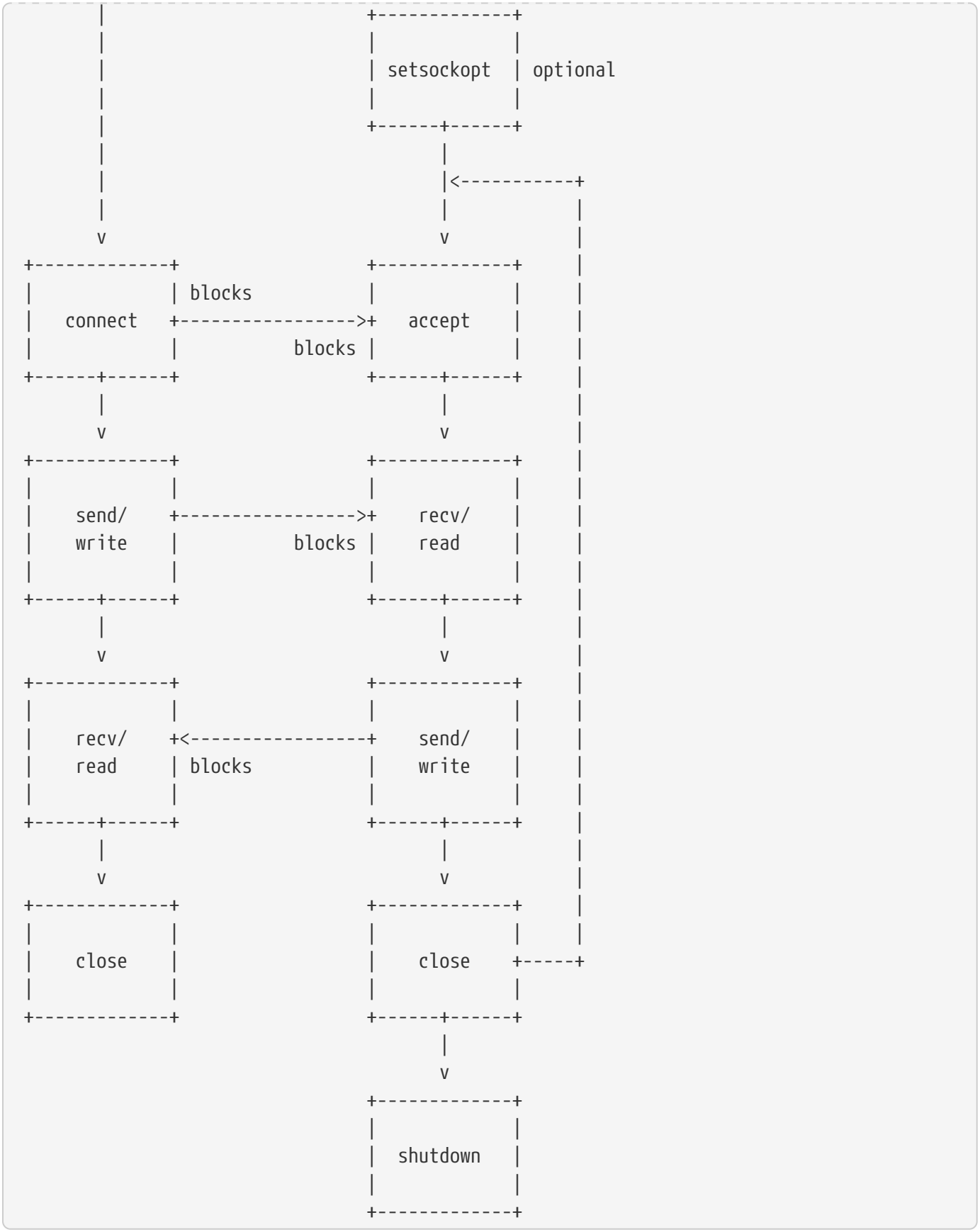
When the TCP/IP network stack was being developed, a programming API was necessary that would abstract away the low-level implementation details of the protocol. A server should be able to "register" some kind of endpoint, and a client should be able to target that endpoint with packets. The result is a now rather well-known paradigm: sockets.

This chapter is not concerned with the specifics of low-level protocol implementations. When used properly, the socket API is generic enough that changing from one protocol to another takes few changes to the code, if any.

The following picture is a block diagram showing the function calls that are typically made by the client and server side of a connected socket. (There are also connectionless sockets, which changes the diagram by removing a few of the boxes.)

Overview of connected socket communication flow




setsockopt
optional
connect
blocks
accept
send/
write
recv/
read
recv/
read
blocks
send/
write
close
close
shutdown

Socket Types and Domains

All sockets fall into two broad categories, known as ***types***, regardless of the protocols used to actually communicate. They are ***connected*** sockets (a.k.a. stream-oriented, `SOCK_STREAM`) and ***connectionless*** sockets (a.k.a. datagram-oriented, `SOCK_DATAGRAM`). There is a third common type, the ***raw*** socket (`SOCK_RAW`), which is used primarily to bypass the normal network protocol handlers and allow for crafting and transmission of custom packets directly on the underlying hardware. (Some systems may have even more, but those are outliers.) Raw sockets are used by commands such as `ping` and `traceroute`, for example, and will not be discussed further in this module.

There are also socket ***domains***. The domains are `AF_INET`, `AF_INET6`, `AF_IPX`, `AF_APPLETALK`, and others. (The focus of this chapter will be `AF_UNIX`.) The domain specifies how host identifiers will be expressed.

The original intent was that the programmer would identify the domain and type of socket they wanted, and the operating system would then choose the appropriate protocol. So the domain would identify IPv4, or IPv6, or AppleTalk, and then the type would specify stream-oriented or datagram-oriented. That would be enough information for the operating system to pick the best protocol (or the programmer could still specify a particular protocol, if they wished).

This chapter will focus on UNIX domain sockets. The stream-oriented version works like a bidirectional FIFO, while the datagram-oriented version is similar to a message queue (a future topic).

Creating a Socket

The `socket()` function call is used to create the socket. This is where the domain, type, and protocol are specified. Once this has been accomplished, an address can be bound to the socket so that it can receive unsolicited incoming connections (if desired).

Note the addition of `sys/types.h` to the header file list. Linux doesn't require this but some older BSDs do, so if one is interested in maximum source level portability, it is safest to include it.

```
#include <sys/socket.h>
#include <sys/types.h>

int socket(int domain, int type, int protocol);
```

The return value from the `socket()` function is a ***socket descriptor***, a small integer number that represents an opened socket. In actuality, it is an index into the process's file descriptor table. Just as file descriptors keep track of information about how a process opened a file, what the seek offset is set to, which filesystem actually performs the reads and writes, and so on, the socket descriptor holds information about which domain is being used and what options are set on the socket.

Creating the socket is simple. Here's an example of creating a stream-oriented socket in the UNIX domain.

```
#include <sys/socket.h>
#include <sys/types.h>
    ...
int s = socket(AF_UNIX, SOCK_STREAM, 0);
if (s == -1) {
    ...
}
```

Binding an Address and Port (server only)

On the server side, the socket needs to be bound to a particular IP address and port number. The port number is a 16-bit unsigned integer that identifies the service. Well-known services are those below 5,000. For example, http (port 80) and https (port 443), mail delivery (smtp on port 25), and ssh (port 22). Port numbers below 1024 are restricted on POSIX systems to users with appropriate privileges.

```
#include <sys/socket.h>
#include <sys/types.h>

int bind(int socket, const struct sockaddr *address, socklen_t address_len);
```

The only complicated part of the `bind()` function is the second parameter, the `struct sockaddr`. Because a socket can represent so many different domains, the format of the structure can be radically different from one domain to the next. So the `sockaddr` structure is generic: the first field is **always** and `unsigned short` that identifies the address family (i.e., the domain). The remaining fields in the structure are specific to the domain. When the application calls `bind()`, it first checks to ensure the address family in the structure is the same as the address family used to create the socket. Only if they match is the rest of the structure examined.

This means the application needs to include the proper header file that defines the structure it wants to use, and that the structure itself be initialized properly. Most applications will do this work in a function that can easily be swapped out if/when the application needs to support other domains.

The `bind()` function can error with `EADDRINUSE` which means the given socket address is already taken and the application should choose another. How to handle this varies depending on the socket domain. For UNIX domain sockets, this means that the pathname identified in the sockaddr structure refers to an existing directory entry. The server should remove this entry prior to terminating.

Here is some code that creates a new UNIX domain sockaddr structure and populates it with data. Note that the `sun_path` field is a character array with a maximum length. (Note that a truly portable application may want to make use of the `struct sockaddr_storage` data type which is not discussed here.)

```
#include <sys/un.h>             // For AF_UNIX domain sockets
    ...
char* my_personal_socket;       // Pick a pathname for the socket
struct sockaddr mysockaddr = {
    .sun_family = AF_UNIX,
};
strncpy(mysockaddr.sun_path, my_personal_socket, sizeof(mysockaddr.sun_path)-1);
mysockaddr.sun_path[ sizeof(mysockaddr.sun_path)-1 ] = '\0';
    ...
int rv = bind(s, &mysockaddr, sizeof(mysockaddr));
if (rv == -1) {
    ...
}
```

Note that `bind()` can fail with error code EADDRINUSE

Setting Backlog Queue Size (server only)

A frequent design for the server side is a main loop that accepts incoming connection requests, handles the communication with the client, then loops and waits for another connection request.

However, if the client makes a connection request and the server is not waiting in an `accept()` function, the client will eventually time out and fail (the error is `ETIMEDOUT`). In order to avoid this, the server may call `listen()` and specify the length of a *backlog queue*. This is a list of all clients who have made a connection request but whose request has not been acknowledged by the server calling `accept()`. The maximum size of this queue is system-dependent, but even very old systems typically allowed 8 such pending requests and modern Linux systems allow up to 128!

```
#include <sys/socket.h>
#include <sys/types.h>

int listen(int socket, int backlog);
```

The proper setting depends on the overall usage patterns on the server. For example, if the clients stay connected for a very short amount of time (on average), then the server may want a larger queue size. Clients at the end of the queue likely won't have to wait very long. If each client interaction can last seconds (or minutes), then a shorter queue length might be more appropriate. That way clients don't wait forever in a long queue, but return an error when the queue is full (the error is `ECONNREFUSED`).

The `listen()` function **must** be invoked to allow receipt of incoming connections, although the value for the queue size can be zero.

```
#include <sys/socket.h>
#include <sys/types.h>
    ...
int rv = listen(s, 5);
if (rv == -1) {
    ...
}
```

Accepting Incoming Connection Requests (server only)

Once the server-side socket is listening for connections, it can `accept()` one from the queue. If there are no connections in the queue, this call will block.

Note that parameters two and three are return values. When `accept()` returns without an error, the `addr` field is set to contain the socket information of the originating client, and `addrlen` is set to the size of the structure (for those domains that have variable size structures, such as the UNIX domain).

```
#include <sys/socket.h>
#include <sys/types.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

The return value from `accept()` is a new socket descriptor that represents the connection. The original socket is not modified in any way (it is still in listening mode, for example).

Because the newly returned socket is *connected*, the application can use either `read()` and `write()` or `recv()` and `send()` system calls to transfer data. A common approach is for the server to accept incoming connections, then `fork()` and let the child handle the new connection, probably `dup()`'ing the new socket into stdin, stdout, and stderr, while the server loops around and accepts another connection.

```
#include <sys/socket.h>
#include <sys/types.h>
    ...
int newfd = accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
if (newfd == -1) {
    ...
}
```

How the Client Initiates a Connection (client only)

At this point, the server has created the socket for listening, bound an address, set listening mode and the backlog size, and is waiting for connection to come in.

It's time for the client to connect!

```
#include <sys/socket.h>
#include <sys/types.h>

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

The second parameter identifies the address used in the server's `bind()` function.

It's possible in some domains to choose an address that is generic, such as the `AF_INET` domain allowing the special IP address, `INADDR_ANY`, to mean any and all available IP addresses on the machine. As long as the client specifies one of these IP addresses, the connection request will be routed to the correct system.

Once the `connect()` function returns, the socket is in *connected* mode. Because the newly returned socket is *connected*, the application can use either `read()` and `write()` or `recv()` and `send()` system calls to transfer data. (The `recv()` and `send()` functions are meant for generic socket communication, while `read()` and `write()` can only be used on connected sockets.)

```
#include <sys/socket.h>
#include <sys/types.h>
    ...
int s = socket(AF_UNIX, SOCK_STREAM, 0);
    ...
// Populate sockaddr_un with server's information
int rv = connect(s, &server_addr, addrlen);
if (rv == -1) {
    ...
}
```

Communication is Key

The server now has a socket it use to listen for incoming connections, and the client has a socket that it uses for initiating those connections. The only thing that remains is to actually send data. Connected sockets work like ordinary files, so `read()` and `write()` are often used. Because sockets are bidirectional, they are easier to setup than FIFOs.

NOTE

In fact, there is a special system call, `socketpair()`, that is a cross between unnamed pipes and UNIX domain sockets! It creates two socket endpoints (each of which are bidirectional, like usual). This makes it easy to fork and have the parent and child each use one of the endpoints.

One question that comes up is... "Which side should initiate the conversation?"

But that question doesn't have a single answer, because it depends on the application-level protocol. For example, the HTTP protocol dictates that the client connects and the server doesn't respond at all (beyond setting up the TCP session). Instead, the client starts sending HTTP commands, using an empty line to indicate the end of the command stream. That triggers the server to process those commands and send back some metadata about the response, followed by an empty line, followed by the actual data.

Other protocols might work differently. Connecting to an SSH server, for example, causes the server to immediately begin the challenge-response authentication sequence.

When one creates their own application-level protocol, one gets to decide how it is going to work.

Datagram-oriented Sockets

The above sections have dealt primarily with stream-oriented sockets.

Here are some differences between those and datagram-oriented sockets:

1. Connected sockets send their data as a stream of bytes, and delivery is guaranteed and received bytes will be in the same order as they were sent.
2. Connectionless sockets send separate, individual packets, that are not considered by the underlying socket to be a sequence.

   This means there is no attempt to verify receipt made by the protocol, and there is no guarantee that packets will arrive in the same order they were sent.

3. Connectionless sockets support the concept of ***out of band*** data (orr just *OOB* data).

   These are typically "control packets" that are not to be treated as data. Examples include flow control instructions and interruption of normal processing, when those features are implemented at the application level and not in the protocol.

4. Connectionless sockets can use broadcast addresses to send a single packet to multiple address at once.

   Connected sockets cannot do this, as the connected state doesn't allow for an socket address other than the one specified in the initial connection.

In closing, think of connectionless sockets like push-to-talk radios, and connected sockets like telephones.

Summary

Students have learned the POSIX socket API as it relates to UNIX domain sockets. This includes the differences between stream-oriented and datagram-oriented sockets.

- Socket types and domains
- Creating and binding a socket
- System calls: sockets
- UNIX domain stream sockets
- Differences between connected and connectionless sockets

Chapter 17. IPC: Alternative Techniques

Students will learn about extensions to previous I/O models that allow for non-blocking I/O and detection of received data awaiting processing.

- Nonblocking I/O
- Signal-driven I/O
- I/O multiplexing: `poll()` and `select()`
- Comparing `poll()` with `select()`
- The Linux epoll API

Introduction

Previous discussions so far in this module have described how IO operations will, generally speaking, block until data becomes available. For example, reading from a file automatically waits until the IO is complete before the next line of application code is executed. (Writes are not handled the same way, however; writes are typically asynchronous and application code continues executing unabated.)

With the advent of multithreading, operations that cause a process to block are not the bottleneck they used to be. Now, applications can spawn a thread to handle the IO operation while other threads continue to perform work. However, it can be difficult for an existing application to take advantage of threads if it wasn't designed with them in mind.

The chapter focuses on techniques that prevent or eliminate blocking IO calls ***without*** the need for adding threads. It is important to note that some of these techniques involve periodic polling to see if a resource is capable of IO. In general, polling should be avoided as it consumes cpu without doing any useful work. Other approaches should be used first, if possible.

The possible techniques are:

1. Use nonblocking mode on file descriptors
2. Use signal-driven IO; signals are triggered by IO conditions
3. Use some form of polling:
    a. Legacy approach is `select()`
    b. More modern approach is `poll()`
    c. The Linux-specific `epoll()` API

This chapter will examine each one, focusing primarily on POSIX compatible techniques.

Nonblocking IO

Previous chapters have mentioned nonblocking IO, but this will be the first one to explore it in detail.

Because many devices are slower than the cpu, and hence, slower than application code, it may be necessary for the device driver to put a process to sleep while it waits for data from the device. It does not *have to* do this — it could just busy-wait in a tight loop for the data to arrive — but busy-waiting means the cpu cannot be used for anything else.

Yet there may be situations where an application wants to read from a device *if data is available* and return immediately if it is not. This is the purpose of nonblocking mode on POSIX operating systems.

When `open()` is invoked, flags can be passed as the second parameter that describe how the `open` should be performed. Common flag include `O_RDONLY`, `O_WRONLY`, and `O_RDWR`. Those all control the direction of data flow (from the file, into the file, or both, respectively). Other options are also available, such as `O_CREAT` (create if necessary), `O_TRUNC` (truncate if the file already exists), and `O_EXCL` (used with `O_CREAT` to indicate the file *must NOT* already exist).

There are many more such flags, however. This topic will discuss this one:

- `O_NONBLOCK` (or `O_NDELAY`) turns on nonblocking mode.

  This causes system calls that would normally block to return `-1` with `errno` set to `EAGAIN` instead. This option is not implemented for regular files and block devices on Linux. However, because a future implementation of Linux *may* implement these flags, applications should be coded as though they have their intended effect.

Turning on `O_NONBLOCK`

The `open()` function was discussed in the early part of this module. Here are the two function signatures that apply:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Both of those prototypes include the `flags` argument as the second parameter. That is where the additional `O_NONBLOCK` flag should go.

If the application is dealing with a file descriptor, the `fcntl()` function can be used instead. This is the same system call that is used for file record locking, as discussed previously in the module.

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* arg */ );
```

The `fcntl()` system call is used to perform arbitrary operations on file descriptors that are not covered by other system calls. For this chapter, we are interested in the *file status flags*. These are initially the *flags* parameter provided to the `open()` system call, but `fcntl()` allows them to be queried and changed:

- The current flags are returned from the call `fcntl(fd, F_GETFL)`.
- New flags are set with the call `fcntl(fd, F_SETFL, new_flags)`.

So, to add the `O_NONBLOCK` flag:

```
    ...
int flags = fcntl(fd, F_GETFL);
if (fcntl(fd, F_SETFL, flags | O_NONBLOCK) == -1) {
    // handle error
}
    ...
```

NOTE

When dealing with a file descriptor that represents a terminal (the `isatty()` function returns true), turning on nonblocking mode is only the first part of obtaining keystrokes character by character. The terminal driver will ordinarily only return data when certain control characters are typed (such as CR, or LF, or EOF). That is called *canonical mode*.

The driver must be put into non-canonical mode to activate one keystroke at a time. This is done using the `termios.h` library and the `tcgetattr()` / `tcsetattr()` functions and turning off the `ICANON` bit.

Nonblocking mode for other file descriptor types (sockets, pipes, and devices) don't need any additional processing.

Signal-driven IO

Nonblocking mode requires that an application be written to expect that `read()` will fail with `-1` and `errno` set to `EWOULDBLOCK`. In some cases, this can add significant complexity to the data input loop, particularly when the application is trying to get other work done simultaneously.

Consider an application that invokes a third-party library to perform a sort algorithm, or a decryption algorithm, or some other potentially long running function. Unless the library were designed for it, there's no way for the application to process new data when it arrives if the library is busy in a cpu-bound loop. So another technique is needed — signal-driven IO.

Signal-driven IO means exactly what it says. The application configures a signal handler to be invoked with IO can be performed on a file descriptor, and the operating system will deliver the signal at the appropriate time.

NOTE

Signal-driven IO is not used much in modern applications. The complexities of performing IO in signal handlers (where dynamic memory allocation cannot be performed) make threads the preferable choice for newly designed and written applications. However, legacy code may still use this feature, and for simple cases signal-driven IO may still be a pragmatic choice.

For example, the application may be waiting for data to arrive on a pipe or socket, but still want to invoke that third-party library function. The file descriptor is put into `O_ASYNC` mode and a signal handler is registered for `SIGIO`. The application then calls the library function. When there is data waiting to be read, the signal will be triggered and the handler can perform the IO. The tricky aspect of this approach is that signal handlers are limited to async-signal-safe functions and that means no memory allocation.

When using signal-driven IO, there are two things to be configured:

- Turn on `O_ASYNC` in the file status flags.

  This causes the operating system to generate `SIGIO` (by default) when input or output becomes possible on the file descriptor. This feature is available only for certain devices: terminals, pseudoterminals, sockets, and pipes/FIFOs.

WARNING

A bug in the Linux library currently makes it impossible to enable signal-driven IO using this flag when calling `open()`. Use the `fcntl()` interface instead (discussed later in this chapter).

- Set the pid that should receive the signal via `fcntl(fd, F_SETOWN, pid)`.

  If the value of `pid` is positive, it represents a process id. If the value of `pid` is negative, it represents a

process group id.

In multithreaded applications, use `F_SETOWN_EX` instead (see the man page for `fcntl`).

- Optional: specify a particular signal using `fcntl(fd, F_SETSIG, sig)` (including `SIGIO`).

  This optional setting provides one significant benefit: the signal handler's prototype is modified to allow passing a `siginfo_t` parameter if `SA_SIGINFO` is added to the `sa_flags` field when registering the signal handler. This extra parameter provides information such as which file descriptor triggered the signal. Without this information, the signal handler would have to use `select()` or similar to determine which file descriptor triggered the signal (discussed next).

```
// The file descriptor must be opened (or fcntl'd, as described above) with O_ASYNC.
if (fcntl(fd, F_SETFL, flags | O_ASYNC) == -1) {
    // handle error
}
    ...
int rv = fcntl(fd, F_SETOWN, getpid());
if (rv == -1) {
    // handle error
}
    ...
```

I/O multiplexing: `poll()` and `select()`

Whether using nonblocking IO or signal-driven IO, there may be times when an application needs to determine if IO could be completed *without blocking*. For example, when writing to a pipe, an application might need to know if the write operation would block (and thus delay execution). Or reading from a socket would delay if no packets are currently ready to be processed. In order to determine whether these operations would block, POSIX provides both `select()` and `poll()`. The `select()` function is the older and less flexible of the two, but it as the legacy API it is more well-known and well-used. This section will investigate the use of `select()` and cover `poll()` afterwards.

First, here is the function prototype for `select()`. Note that the header files have changed over time. Since this module prefers the POSIX interface, our example code will always use the `sys/select.h` header file.

```
#include <sys/select.h>

// From <sys/time.h>
struct timeval {
    long    tv_sec;     // seconds
    long    tv_usec;    // microseconds
};

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval
*timeout);
```

The parameters to `select()` are:

1. `nfds` is the highest numbered file descriptor in the `fd_set` that needs to be checked, plus one. If an application wants to check file descriptors 0, 3, and 4, the value of `nfds` would be `5`.
2. `readfds` is a bitmask of which file descriptors should be checked that a read operation would succeed.
3. `writefds` is a bitmask (as `readfds`) for write operations.
4. `exceptfds` is a bitmask (as `readfds`) for "exceptional" operations. Accepting a connection on a socket is considered a *readability* event and not an *exceptional* event.

WARNING	When calling `select()` in a loop, the `fd_set` parameters must be reinitialized prior to calling `select()` *every time through the loop* because they are modified by `select()`.

NOTE

The `fd_set` type is similar to `sigset_t` in that it is an opaque bitmask data type. The programmer is not supposed to know what is in it or how it works, only how to use it.

There are four macros to manipulate the `fd_set`. They are:

1. `FD_ZERO(fd_set *set)` - clears the entire bitmask
2. `FD_CLR(int fd, fd_set *set)` - clears the bit for `fd`
3. `FD_SET(int fd, fd_set *set)` - sets the bit for `fd`
4. `FD_ISSET(int fd, fd_set *set)` - tests whether the bit for `fd` is set

5. `timeout` is the length of time that `select()` will wait before returning. As shown in the code block, above, it contains seconds and microseconds (not all operating systems will support microsecond accuracy).

WARNING

When calling `select()` in a loop, the `timeout` parameter must be reinitialized prior to calling `select()` *every time through the loop* because it is modified by `select()` to contain the amount of time remaining.

The return value from `select()` is the number of file descriptors that have read, write, or exceptional conditions available to be processed, and the `fd_set` bitmasks are modified accordingly. An application that wants to process all events should loop the given number of times, checking one file descriptor at a time from each `fd_set` using `FD_ISSET()`.

Because `fd_set` is defined to be a bitmask, there are a limited number of file descriptors that `select()` can monitor. The macro `FD_SETSIZE` is defined to be this number so that applications can check against it.

`terminal.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <time.h>
#include <termios.h>
#include <sys/select.h>

int fd;
struct termios original;
```

```
void cleanup(void)
{
    tcsetattr(fd, TCSANOW, &original);  // restore original settings
}

void handler(int sig)
{
    (void)sig;
    cleanup();
}

/**
 * This program demonstrates nonblocking mode for user input.
 * By using nonblocking mode, the application can read data
 * when it becomes available, but in the meantime it can perform
 * other functions.
 *
 * This program displays a prompt that consists of the current
 * time, updating the time every second.
 */
int main(void)
{
    fd = open("/dev/tty", O_RDWR|O_NONBLOCK);
    if (fd < 0) {
        perror("/dev/tty");
        return 1;
    }

    tcgetattr(fd, &original);   // save the original settings
    atexit(cleanup);

    signal(SIGINT, handler);

    struct termios terminal;
    tcgetattr(fd, &terminal);
    terminal.c_lflag &= ~ICANON; // See termios(4)
    terminal.c_cc[VMIN] = 1;
    terminal.c_cc[VTIME] = 0;
    tcsetattr(fd, TCSANOW, &terminal);

    fd_set fds;

    char line[4096];
    line[0] = '\0';
    size_t line_len = 0;

    for (;;) {
```

```
        // display prompt
        // '%I:%M:%S %p' plus NUL byte is 12 chars, round up to 16
        char timestamp[16];
        time_t clock = time(NULL);
        struct tm *tm = localtime(&clock);
        size_t len = strftime(timestamp, sizeof(timestamp), "%I:%M:%S %p", tm);
        if (len == 0) {
            fprintf(stderr, "Time format too long; exiting\n");
            return 2;
        }

        printf("\rEnter some text (%s): %s", timestamp, line);
        fflush(stdout);

        struct timeval timeout = {
            .tv_sec  = 1,   // One second timeout
            .tv_usec = 0,
        };
        FD_ZERO(&fds);
        FD_SET(fd, &fds);   // Only tracking a single fd

        int rv = select(fd+1, &fds, NULL, NULL, &timeout);
        if (rv > 0) {
            // There is some data waiting...
            char buff[1024];

            // Nonblocking mode: this reads whatever is available _right now_.
            ssize_t rlen = read(fd, buff, sizeof(buff));
            if (rlen < 0) {
                perror("Unable to read input");
            } else if (line_len + rlen >= sizeof(line)) {
                // TODO Make `line` dynamically allocated
                errno = EMSGSIZE;
                perror("line buffer");
            } else {
                // Handling backspace is left as an exercise for the reader. :)
                buff[rlen] = '\0';
                // +1 for nul terminator
                strncpy(line + line_len, buff, rlen+1);
                line_len += rlen;
                if (line[line_len-1] == '\n') {
                    break;
                }
            }
        }
    }
    printf("\nComplete line is: '%s'\n", line);
```

```
}
```

Comparing `poll()` with `select()`

Both `poll()` and `select()` are defined in POSIX. So which one should an application use? The answer, as in most programming questions, is, "It depends."

The main consideration will be legacy code vs. new code.

- The `select()` system call has been around since the very early days of BSD and is implemented on every Unix system, including some that are not POSIX.
- The `poll()` system call is newer (it was added to System V as a better replacement for `select()`).

Since `select()` is likely to be used in older code, it is reasonable to continue to use it in those cases and to use `poll()` in newer code. However, both can be used within a single process, so the selection might also factor in performance or ease-of-use requirements.

Here is the function prototype for `poll()`:

```
#include <poll.h>

int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

Note that `poll()` doesn't use a bitmask—the application provides an array of file descriptors (and the size of the array). Each array element contains the file descriptor number itself, as well as flags to indicate which events the application is interested in. (In contract, `select()` provides three different bitmasks, one each for reads, writes, and exceptional events.)

Here are the differences between `poll()` and `select()`.

- Functionality

  `poll()` and `select()` provide basically the same functionality. Some of the details do differ:

  1. The `select()` system call overwrites the `fd_set` parameters whose pointers are passed as arguments. This requires a typical loop to either have a backup copy of the `fd_set` values that it can copy onto other variable, or populate the `fd_set` values with a loop each time.
  2. The `poll()` system call does not have an inherent limit on the number of file descriptors that can be monitored at once, unlike `select()` whose bitmask is limited to `FD_SETSIZE` bits.
  3. The `poll()` system call offers a larger variety of events that can be waited for, although for the common case these other events do not add much value.
  4. The `poll()` system call uses a timeout value measured in milliseconds (thousandths of a second) while `select()` uses microseconds (millionths of a second). In reality, the granularity of the

timeout is unlikely to matter much, given that all systems are going to round up the timeout to the nearest interval they support (which is based on the scheduler, the device driver, and possibly other kernel components).

- Performance

Both are likely to be about the same speed. Some notes on where they might differ:

1. They both handle file descriptors in a linear way: the more file descriptors you ask them to monitor, the slower they get.
    a. This applies inside the kernel, where the actual monitoring takes place.
    b. And it occurs inside the application, where a loop is required to check which file descriptors are ready for IO.
2. The `select()` system call uses a bitmask which may have many bit cleared, but the kernel — and the application — must still iterate over those bits to determine which file descriptors to process.
3. The `select()` system call only uses three bits of data per file descriptor (maximum), while `poll()` typically uses 64 bits per file descriptor. This means `poll()` has to copy a lot more data into the kernel on each invocation than `select()` does. This is a small win for `select()`, but little wins add up to large victories.

The Linux epoll API

This is yet another kernel API that is specific to Linux.

Other operating systems provide similar facilities, such as `kqueue` on FreeBSD and `/dev/poll` on Solaris, but there is no governing POSIX standard.

It works by creating a data structure that contains two lists of file descriptors: a "watched" list, and a "ready" list. The former identifies which file descriptors the application is interested in, and the latter flags file descriptors that are ready for some kind of IO. THis data structure is operated on as though it were itself a file descriptor.

Because the data structures are themselves "watchable" file descriptors, an `epoll` structure can be added to the watch list of another `epoll` instance. This allows an application to create sets of watched file descriptors as `epoll` instances, then create a "meta" `epoll` instance that adds and removes the others as needed based on the application state.

Because the data structures are created and maintained inside the kernel, there is very little data that needs to cross the user space/kernel space boundary during normal operation; however, the `epoll_wait()` system call does need to transfer back to user space the list of ready file descriptor.

For more information on this topic, see the man page for `epoll(7)`.

Summary

Students will learn about extensions to previous I/O models that allow for non-blocking I/O and detection of received data awaiting processing.

- Nonblocking I/O
- Signal-driven I/O
- I/O multiplexing: `poll()`
- Comparing `poll()` with `select()`
- The Linux epoll API

Chapter 18. POSIX IPC: Introduction and Overview

Students will learn about POSIX IPC mechanisms, including an overview of semaphores, shared memory, and message queues.

This chapter only introduces the topics and provides information common to all three mechanisms.

- Shared memory
- Semaphores
- Message queues

Introduction

The concepts behind interprocess communication (IPC) have been discussed earlier in this module. Those concepts apply across multiple operating systems, but not all systems implemented them the same way or adhered to the same semantics regarding usage.

POSIX created three new APIs which follow the pattern created by Unix System V, but cleaned up the interface for each one.

Those three are *shared memory*, *semaphores* (considered briefly in the threads chapter), and *message queues*.

All three APIs follow a similar paradigm:

- Open or create the resource, including initialization
- Perform operations on the resource
- Close or destroy the resource

All of these structures have *kernel persistence*—if the kernel is ever rebooted, any existing versions of these data structures are cleared.

POSIX Shared Memory

Shared memory is exactly that—a region of virtual memory whose physical RAM also appears as a virtual region of another process.

Shared memory is used similar to semaphores: open/create the shared memory, map the shared memory into the process's addressing space, perform operations on data at the mapped address, and close/unmap the shared memory when the resource is no longer needed.

The virtual addresses do not have to be the same (and frequently are not), as long as the same physical frame is used by both.

Three terms apply when speaking of memory. All three are the same size, have the same alignment constraints, and are resources managed by the operating system (meaning they are allocated, used, and freed by the kernel).

Pages

virtual blocks of addresses within the process's addressing space, which may or may not have any physical memory behind them.

Frames

physical RAM locations which may or may not be in use at any given point in time.

Slots

physical locations on swap space which act as ***backing store*** for anonymous pages whose frame was stolen for use elsewhere.

Shared memory is frequently used when two or more processes need to share data in as quick a method as possible. Using techniques such as pipes or Unix domain sockets would require that the memory pass through an intermediate buffer before a copy of it arrives at the other end. Such copying is slow and inefficient compared to simply storing the data in memory and letting another application access it there.

There are still synchronization issues, however. Semaphores are a good way to resolve them, storing an unnamed semaphore in the shared memory itself or using a named semaphore.

POSIX Semaphores

Semaphores are designed for relatively simple synchronization between threads or processes. POSIX semaphores come in two varieties: unnamed and named.

- Unnamed Semaphores
  - These were discussed in the threads chapter as a way to synchronize threads using a process-global variable (the `sem_t` semaphore).
  - Unnamed semaphores can also be used between processes, but then must be stored in memory that is shared between the processes (shared memory).
- Named semaphores
  - These types of semaphores operate similar to files: open the file, perform operations on the file, close the file when no longer needed.
  - The operations performed, `sem_wait()` and `sem_post()` are the same as for unnamed semaphores, it is only the open/init and close procedures that differ.

For both types, the process to use them is similar: open/create the semaphore, perform operations on the semaphore, and close/destroy the semaphore when the resource is no longer needed.

POSIX Message Queues

Message queues are data structures that support simultaneous access by multiple threads or processes at once. One or more processes or threads can add messages to the queue, and one or more processes or threads can remove messages from the queue.

They are used similarly to semaphores and shared memory: open/create the message queue, perform operations on the queue, and close the queue when the resource is no longer needed.

Messages are always assigned a priority and are always removed from the queue in priority order. POSIX requires a compliant system to support at least 32 priority levels; Linux supports 32768 priority levels. The range can be determined at runtime and is always from zero (lowest) to `sysconf(_SC_MQ_PRIO_MAX)-1` (highest).

The `/proc` filesystem contains default values for many message queue-related values:

- The maximum number of message queues on the system at once.
- The maximum number of messages and a default limit in a single message queue.
- A system-wide limit and a default limit on the maximum size of individual messages.
- A per-queue limit and a default limit on the maximum size of individual messages.

There is a resource limit (see `RLIMIT_MSGQUEUE` and `getrlimit()`) which is the maximum amount of space that can be consumed by all messages and message queues belonging to a particular real user id.

See the man page for `mq_overview(7)` for details.

Summary

Students have learned about POSIX IPC mechanisms, including an overview of semaphores, shared memory, and message queues. Details on each will be provided in future chapters.

- Shared memory
- Semaphores
- Message queues

Chapter 19. POSIX IPC: Semaphores

Students will learn how to implement synchronization strategies uding POSIX semaphores.

- Review of unnamed semaphores
- Named semaphores
- Semaphore operations

Introduction

This module has already discussed unnamed semaphores, but only in the context of a multithreaded application needing synchronization.

This chapter will finish detailing unnamed semaphores and also cover a new topic, named semaphores.

Both unnamed and named provide the same basic functionality: synchronization of code executing in parallel. But unnamed semaphores have so far only been used within a single addressing space; they ***can*** be used between processes as well if stored in shared memory.

Review of Unnamed Semaphores

Here's a review of the system calls relevant to unnamed semaphores:

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);

int sem_post(sem_t *sem);
```

The second parameter to `sem_init()` has not been discussed yet; it signifies that the semaphore is going to be used by multiple processes. If the value is zero, the semaphore is going to be used by threads in a single process. If the value is non-zero, the semaphore is going to be used by multiple processes, and therefore must be placed in some type of shared memory region.

Named Semaphores

A topic not yet discussed is *named semaphores*. They operate essentially the same way as unnamed semaphores, except the creation and destruction is different. Instead of creating a variable in globally accessible memory, the application calls `sem_open()` and a pointer to the semaphore is returned. Other processes can call the same function and pass it the same `name` parameter in order to access the same semaphore.

Named semaphores are actually allocated out of shared memory, similar to how `shm_open()` allocates shared memory.

Here are the changes compared to unnamed semaphores:

```
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <semaphore.h>

// These replace sem_init() and sem_destroy()
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
int sem_getvalue(sem_t *sem, int *sval);
int sem_close(sem_t *sem);

// Compiled and linked with -pthread
```

Note that the first `sem_open()` prototype is used by processes wanting to connect to and use an existing semaphore. The second prototype is for those processes that need to create and initialize the semaphore.

Example

This example demonstrates the use of named semaphores (since unnamed semaphores were used in the threads chapter).

There are two programs involved: the `reader` and the `writer`. The reader creates and initializes a shared memory block. It then waits for writers to write a string into the shared memory area. Concurrent access to the shared memory is controlled via a named semaphore.

Example `reader` that creates and reads from shared memory

```
#include <assert.h>
#include <fcntl.h>
#include <semaphore.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <sys/mman.h>

#include "sem.h"

int main(void)
{
    // This program assumes that the ring buffer can fit in a single page
    assert(sizeof(struct ring_buffer) <= (unsigned)sysconf(_SC_PAGE_SIZE));

    // Create and initialize the semaphores first.
    sem_t *sem_write = sem_open(sem_to, O_CREAT|O_EXCL, 0666, 0);
    if (sem_write == SEM_FAILED) {
        perror("Unable to open output");
        return 1;
    }

    sem_t *sem_read = sem_open(sem_from, O_CREAT|O_EXCL, 0666, 0);
    if (sem_read == SEM_FAILED) {
        sem_close(sem_write);
        perror("Unable to open input");
        return 2;
    }

    int shmfd = shm_open(share, O_RDWR|O_CREAT|O_EXCL, 0666);
    if (shmfd == -1) {
        perror("Unable to open shared memory");
        sem_close(sem_read);
```

```
        sem_close(sem_write);
        return 3;
    }

    if (ftruncate(shmfd, sysconf(_SC_PAGE_SIZE)) == -1) {
        perror("Unable to size shared memory");
        close(shmfd);
        sem_close(sem_read);
        sem_close(sem_write);
        // This error is unique to the reader
        return 5;
    }

    struct ring_buffer *data = mmap(NULL, sysconf(_SC_PAGE_SIZE),
            PROT_READ|PROT_WRITE, MAP_SHARED, shmfd, 0);

    // Once the shm is mapped, don't need the fd any more.
    close(shmfd);

    if (data == MAP_FAILED) {
        perror("Unable to map memory");
        sem_close(sem_read);
        sem_close(sem_write);
        return 4;
    }

    data->read_pos = 0;
    data->write_pos = 0;
    data->num_chars = 0;
    sem_post(sem_write);
    for (;;) {
        sem_wait(sem_read); {
            printf("%.*s", data->num_chars, data->buffer + data->read_pos);
            data->read_pos += data->num_chars;
            data->num_chars = 0;
        } sem_post(sem_write);
    }

    sem_close(sem_read);
    sem_close(sem_write);
}
```

Example `writer` that opens and writes to shared memory

```
#include <assert.h>
#include <fcntl.h>
```

```c
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <sys/mman.h>

#include "sem.h"

int main(void)
{
    // This program assumes that the ring buffer can fit in a single page
    assert(sizeof(struct ring_buffer) <= (unsigned)sysconf(_SC_PAGE_SIZE));

    // Open the semaphores.
    sem_t *sem_write = sem_open(sem_to, 0);
    if (sem_write == SEM_FAILED) {
        perror("Unable to open output");
        return 1;
    }

    sem_t *sem_read = sem_open(sem_from, 0);
    if (sem_read == SEM_FAILED) {
        perror("Unable to open input");
        sem_close(sem_write);
        return 2;
    }

    int shmfd = shm_open(share, O_RDWR, 0);
    if (shmfd == -1) {
        perror("Unable to open shared memory");
        sem_close(sem_read);
        sem_close(sem_write);
        return 3;
    }

    struct ring_buffer *data = mmap(NULL, sysconf(_SC_PAGE_SIZE),
            PROT_READ|PROT_WRITE, MAP_SHARED, shmfd, 0);

    // Once the shm is mapped, don't need the fd any more.
    close(shmfd);

    if (data == MAP_FAILED) {
        perror("Unable to map memory");
        sem_close(sem_read);
        sem_close(sem_write);
```

```
        return 4;
    }

    char msg[] = "A: This is a test.\n";

    int countdown = random() % 8;
    while (countdown-- > 0) {
        sem_wait(sem_write); {
            strncpy(data->buffer + data->write_pos, msg, sizeof(data->buffer));
            data->write_pos += strlen(msg);
            data->num_chars += strlen(msg);
        } sem_post(sem_read);

        *msg = (*msg >= 'Z' ? 'A' : *msg+1);
        usleep(100000 * (random() % 10)); // 0..1s in multiples of .1s
    }

    munmap(data, sysconf(_SC_PAGE_SIZE));
    sem_close(sem_read);
    sem_close(sem_write);
}
```

Summary

Students have learned how to implement synchronization strategies using POSIX semaphores. An example was shown that uses named semaphores to control access to a shared memory block.

- Review of unnamed semaphores
- Using named semaphores
- Semaphore operations

Chapter 20. POSIX IPC: Shared Memory

Students will learn how to share data between applications using POSIX shared memory.

- Creating and opening shared memory objects
- Using shared memory
- Synchronizing access

Introduction

Shared memory allows a single piece of physical memory to appear in the address space of multiple processes. This provides those processes with instantaneous access to any data shared within that region.

Once created, a shared memory region exists until it is explicitly remove with `shm_unlink()` or the operating system is rebooted. On Linux, they are created under the `/dev/shm` in-memory filesystem. Linux supports the use of access control lists (ACLs) to control access to files within that filesystem.

The general approach is:

1. Call `shm_open()` to open a shared memory identifier (creating it, if necessary).
    - The return value is a file descriptor.
    - Ordinary file permissions apply.
2. Call `ftruncate()` to specify the size of the shared memory region.
3. Call `mmap()` to attach the shared memory to a virtual address within the process.
    - Memory thus attached can be read-only, read-write, or even write-only.
    - The permissions requested when calling `mmap()` cannot those granted by `shm_open()`.

The application can now use the virtual memory address returned by `mmap()` to access the shared region. The file descriptor from `shm_open()` is no longer needed once the `mmap()` is complete, so it can be closed via `close()`. (If the application has other uses for the file descriptor, it can keep it open, but this is rare.)

Then, when the application is done accessing the shared memory:

1. Call `munmap()` to release the virtual address.
    - The shared memory still exists for other processes that might be accessing it.
2. To completely remove the shared memory, call `shm_unlink()`.

Other calls that apply to ordinary files also apply to shared memory:

- Use `fstat()` with the open file descriptor to retrieve metadata about the shared memory resource (owner id, group id, etc).
- Use `fchown()` and `fchmod()` to change the ownership and permissions, respectively.

Opening/Creating a Shared Memory Block

As described on the previous page, there are only two system calls directly related to shared memory.

```
#include <sys/mman.h>
#include <sys/stat.h>        /* For mode constants */
#include <fcntl.h>           /* For O_* constants */

int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);

// Link with -lrt
```

The name passed to `shm_open()` must start with a slash and is limited to `NAME_MAX` characters (255). The remainder of the name must not contain another slash. The name passed is used when accessing `/dev/shm` to create or locate the given shared memory.

The following values may be bitwise OR'd into the `oflag` parameter:

- `O_RDONLY` - opened for read-only; shared memory opened this way may not be later mmap'd with write permission.
- `O_RDWR` - opened for both read and write.
- `O_CREAT` - creates the memory block if it does not already exist.
- `O_EXCL` - when combined with `O_CREAT`, indicates that the shared memory *must NOT* already exist.
- `O_TRUNC` - if the shared memory block already exists, truncate it to zero size.

On success, the lowest available file descriptor is returned.

The size of the shared memory block is defined by calling `ftruncate()` and specifying a size for the file descriptor.

```
#include <unistd.h>
#include <sys/types.h>

int ftruncate(int fd, off_t length);
```

The file descriptor being modified must be open for write permission.

Mapping the Shared Memory for Use

Once the shared memory block is open, the file descriptor can be used in the `mmap()` call. This system call maps a region of a file into memory, and it does the same thing with shared memory. (The shared memory block can be thought of as a file stored in a RAM disk.)

When the application is done with the shared memory block, it is released using `munmap()`.

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

The `mmap()` and `munmap()` system calls have the following parameters:

1. `addr` - the address that the application wants the operating system to use when attaching the shared memory. Putting an address here is not portable and its use is discouraged; use `NULL` instead.
2. `length` - the size of the shared memory block to be mapped. It is possible map an area smaller than the full size of the shared memory block. The operating system may round this size up to the page size.
3. `prot` - specifics the protection to be applied to all pages within the mapping.
    - `PROT_EXEC` - pages may be executed (generally requires `PROT_READ`)
    - `PROT_READ` - pages may be read from
    - `PROT_WRITE` - page may be written to
    - `PROT_NONE` - pages may not be accessed
4. `flags` - specifies whether changes to the mapping are visible in other processes.
    - `MAP_SHARED` - share this mapping, so "yes", changes are visible elsewhere
    - `MAP_PRIVATE` - creates a private mapping (unused when working with shared memory, but is used by the dynamic loader when shaared libraries are being loaded)
5. `fd` - the file descriptor to be mapped.
6. `offset` - the seek offset within the file where the mapping should begin.

Note that `mmap()` returns a `void` pointer, yet it returns `-1` on error. It is important that the return value be compared against `MAP_FAILED` and *not* `-1`; the `MAP_FAILED` macro has the correct cast to ensure the comparison works correctly.

Synchronized Access to Shared Memory

Synchronization can be done using any of the techniques discussed in previous chapters. Some are more likely to be useful than others.

For example, shared memory is typically used between multiple processes, not multiple threads within a process. However, mutex locks can still be used to synchronize access to shared memory if the mutex:

1. is allocated in shared memory, and
2. can be initialized prior to any other threads accessing it.

The latter is difficult to guarantee because while the `shm_open()` can use `O_CREAT|O_EXCL` to make sure the shared memory doesn't already exist, there is no way to guarantee that another process has not done a normal open and started accessing the shared memory prior to `pthread_mutex_init()` being called.

A better synchronization mechanism is POSIX semaphores.

- Unnamed semaphores can be allocated from shared memory and initialized by passing a nonzero value as the second parameter, `pshared`. This suffers from the same race condition as mutex locks, however.
- Named semaphores can be created, opened, and the semaphore initialized, in a single atomic operation. That makes named semaphores the best choice.

Example

Here is a multi-program application.

1. The first program creates a shared memory block, failing if it already exists.
    - It then writes a string into the shared memory and terminates.
2. The second program attaches to a shared memory block that must already exist.
    - It reads the string and prints it to stdout.

Check the example code for race conditions. Are there any?

creator.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>

const char SHM_NAME[] = "share";
enum { block_sz = 4096 };

int main(void)
{
    int fd = shm_open(SHM_NAME, O_RDWR|O_CREAT|O_EXCL, 0666);
    if (fd < 0) {
        perror("Unable to create shared memory");
        return 1;
    }

    if (ftruncate(fd, block_sz) == -1) {
        perror("Unable to reserve file space");
        close(fd);
        shm_unlink(SHM_NAME);
        return 2;
    }

    char* data = mmap(NULL, block_sz, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    close(fd);
    if (data == MAP_FAILED) {
        perror("Unable to create mapped memory");
        shm_unlink(SHM_NAME);
        return 3;
    }

    char msg[] = "This is a test of the emergency shared memory system.";
    strncpy(data, msg, sizeof(msg));

    munmap(data, block_sz);
}
```

reader.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>

const char SHM_NAME[] = "share";
enum { block_sz = 4096 };

int main(void)
{
    int fd = shm_open(SHM_NAME, O_RDONLY, 0666);
    if (fd < 0) {
        perror("Unable to open shared memory");
        return 1;
    }

    char* data = mmap(NULL, block_sz, PROT_READ, MAP_SHARED, fd, 0);
    if (data == MAP_FAILED) {
        perror("Unable to open mapped memory");
        close(fd);
        return 2;
    }
    close(fd);

    puts(data);
    munmap(data, block_sz);

    if (shm_unlink(SHM_NAME) < 0) {
        perror("Unable to close shared memory");
        return 3;
    }
}
```

Summary

Students have learned how to share data between applications using POSIX shared memory.

- Creating and opening shared memory objects
- Using shared memory
- Synchronizing access

Chapter 21. POSIX IPC: Message Queues

Students will learn how to use POSIX message queues for communicating data between processes.

- Opening, closing, and unlinking a message queue
- Message queue attributes
- Sending and receiving messages
- Message notifications

Introduction

Message queues (or just *MQ*s) are the last POSIX IPC mechanism that this module will be discussing.

Conceptually, MQs are similar to FIFOs and Unix domain sockets, but with a few of changes:

- FIFOs and sockets are *data streams* and it can thus be difficult to tell where one message ends and another begins, but MQs send *structured messages* between process, so boundaries are always clear.
- In practice, FIFOs and sockets can only have a single reader.

  Because writes are only guaranteed by POSIX to be atomic up to 512 bytes, it only makes sense for a FIFO to have a single reader that can piece together any partial writes that may occur. (Sockets have a similar issue, but also add that a given endpoint can only have a single process bound to it.)

  For example, suppose a writer puts 800 bytes into a FIFO. Since only 512 bytes are guaranteed atomic, it would be necessary for the reader to obtain the first chunk of data and save it, then retrieve another chunk of data and append it to the first. That needs to continue until the application's definition of a "complete message" has been read. It would be impossible to have multiple readers in that scenario. (If the message size is guaranteed to be less than 512 bytes, then multiple readers might be a possibility.)

  Message queues allow multiple readers more generally, because the maximum size of a message for a given queue is configurable. If the application wishes to use 800 bytes as the maximum message size, it can configure that size when the queue is created. Thus every reader knows that it is receiving a complete message and not a (potentially) partial one.

- With proper setup, FIFOs, sockets, and MQs can all deliver signals when IO is possible.

  For FIFO and socket writers, the signal is sent when space is available in the buffer. For FIFO and socket readers, the signal is sent when there is data in the buffer waiting to be read.

  For MQs, only a single process at a time can register for signal delivery from a particular queue, and a signal is delivered only when an empty queue has a message added to it. In addition, when a signal is delivered, the notification is turned off and the process must re-register to receive it again.

- FIFOs are strictly first-in-first-out, while MQs are more flexible. Sockets have *out-of-band* packets, a sort of "high priority" packet.

  Each message added to a queue is given a priority (POSIX requires at least 0..31, while Linux provides 0..32767). When receiving, the oldest message with the highest priority is returned. In addition, receiving a message can specify a timeout; the call returns when data is available or when the timeout expires. FIFOs are a stream of data with no priorities assigned to buffer content at all. Sockets

do not support priorities on packets either, although out-of-band data acts like a high priority packet (which essentially provides two priority levels).

- Otherwise, FIFOs, sockets, and MQs are very similar.
  - All support `O_NONBLOCK` when reading from or writing to the resource.
  - All will block when reading if no data is available to read.
  - All will block when writing if no space is available for the new data.
  - All use file descriptors for access (POSIX does not require `select()` or `poll()` to work on MQs, although Linux does support this use).

To summarize, MQs are best when the amount of data that will pass through the queue at any given time is limited, but which may need to be retrieved in priority order. (System V message queues support retrieving messages of a particular priority, instead of just the oldest, highest priority message. POSIX does not support that use.)

Opening a Message Queue

Messages queues must be created before they can be used. They are *kernel persistent*, meaning that they will stick around until the machine is rebooted or until a process removes them. (This is identical to POSIX shared memory and POSIX named semaphores.)

The name of the message must be a string of up to `NAMELEN` characters (255) and must start with a slash ("/"). (Linux will use this name to create a directory entry under `/dev/mqueue`.)

```
#include <fcntl.h>     // For the O_* flags
#include <sys/stat.h>  // For mode constants
#include <mqueue.h>

struct mq_attr {
    long mq_flags;    // Flags: 0 or O_NONBLOCK (unused on open)
    long mq_maxmsg;   // Max. # of messages on queue
    long mq_msgsize;  // Max. message size (bytes)
    long mq_curmsgs;  // # of messages currently in queue (unused on open)
};

mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);
int mq_close(mqd_t mqdes);
int mq_unlink(const char *queue_name);

// Link with -lrt
```

The parameters are:

1. `name` - the name of the message queue; must start with a slash and must not contain any other slashes.
2. `oflag` - flags such as `O_RDONLY`, `O_WRONLY`, and `O_RDWR`. The list also includes `O_CREAT` and `O_EXCL`, which work just as in `open()`. The `O_NONBLOCK` flag can also be OR'd into this parameter to turn on nonblocking mode.
3. `mode` - the permission mask used when creating the file (only applies when creating a message queue).
4. `attr` - pointer to attribute structure that defines the queue characteristics (only applies when creating a message queue), or `NULL` to use defaults.

POSIX only says the return value is a small integer; on Linux, the return value is a file descriptor.

Sending and Receiving Messages

Once the queue is open, processes will want to start sending and receiving messages on the queue. Sending is done using `mq_send()` and receiving is done using `mq_receive()`.

```
#include <mqueue.h>
#include <time.h>

struct timespec {
    time_t tv_sec;      /* seconds */
    long   tv_nsec;     /* nanoseconds */
};

int mq_send (mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio);
int mq_timedsend (mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int
msg_prio,
                  const struct timespec *abs_timeout);

// Link with -lrt
```

The parameters are:

1. `mqdes` - the message queue descriptor (on Linux, the file descriptor of the mq).
2. `msg_ptr` - the address of a block of memory to be copied into the queue. (The POSIX standard probably should've specified this as `void*`.)
3. `msg_len` - the size of the memory block that becomes the message.
4. `msg_prio` - the priority of the message; must be non-negative.
5. `abs_timeout` - (only for `mq_timedsend()`) the function will return by this timestamp. The timestamp is measured as number of seconds from Jan 1st, 1970. (See the man pages for `time(3)` and `gettimeofday(3)`.)

```
#include <mqueue.h>
#include <time.h>

ssize_t mq_receive (mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio);
ssize_t mq_timedreceive (mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int
*msg_prio,
                         const struct timespec *abs_timeout);

// Link with -lrt
```

The parameters are the same as specified for `mq_send()`, except:

- `msg_len` - specifies the amount of space available at `msg_ptr` for the retrieved message. (Applications should generally allocate a buffer based on the maximum message size for the queue.)
- `msg_prio` - a pointer to an integer where the priority of the received message will be stored. If `NULL`, the priority will not be available.

WARNING	Be careful to ensure that `msg_len` is always at least as large as the maximum message size for the queue, and the buffer has been sized accordingly. Otherwise, the `mq_receive()` functions will return `-` with `errno` set to `EMSGSIZE`.

Asynchronous Message Arrival Notification

A process may prefer to receive a signal when a message becomes available in the queue. The `mq_notify()` function may be used to register for such notification. Only a single process can be receiving notification at a time, and once notification is delivered, it must be reset.

```
#include <mqueue.h>

// Not all fields are represented here (some are for internal use)
struct sigevent {
    int sigev_notify;               // SIGEV_{NONE,SIGNAL,THREAD}
    int sigev_signo;                // Signal to send; use SA_SIGINFO for more info
    union sigval sigev_value;       // Data passed with notification
    void (*sigev_notify_function)(union sigval);
                                // Function used for thread notification (SIGEV_THREAD)
    void *sigev_notify_attributes;  // Attributes for notification thread (SIGEV_THREAD)
};

int mq_notify (mqd_t mqdes, const struct sigevent *sevp);

// Link with -lrt
```

The parameters are:

- `mqdes` - the message queue descriptor.
- `sevp` - a pointer to a structure that defines which signal to send and how to send it.

The fields in the `struct sigevent` are used according to the value stored in the `sigev_notify` field of the structure:

`sigev_notify`	Field Name	Field Description
`SIGEV_NONE`		No action is taken when the event is triggered

sigev_notify	Field Name	Field Description
SIGEV_SIGNAL	sigev_signo	The specified signal is sent to the process. If the signal is registered via sigaction() with SA_SIGINFO in sa_flags, the following fields are set in the siginfo_t structure passed to the handler: • si_code - set to SI_MESGQ • si_pid - the pid that sent the message • si_uid - the real user id of the sending process • si_signo - the signal number being delivered • si_value - same as the sigev_value field used when the event was registered See the man page for sigaction() for more information on siginfo_t.
SIGEV_THREAD	sigev_notify_function	The given function is invoked "as if" it were the start function of a new thread.
	sigev_value	The function is invoked with sigev_value as its sole argument.
	sigev_notify_attributes	Specifies the attributes for the new thread.

After being notified, a process should:

1. Re-register for notification if it wants to continue the cycle.
2. Begin processing the messages already in the queue.

It is not possible to "lose" a notification using this approach, since notification is only sent when the message queue is empty and a message is added. It does require that the application process all messages in the queue when the signal is delivered. Failure to do so means messages are still in the queue and the signal will not be sent again.

Status Information

While the status information is perhaps not used very often, being able to query a mq to determine the number of messages it contains or the maximum message size can be helpful.

The structure used, `mq_attr`, was described previously.

```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr, struct mq_attr *oldattr);

// Link with -lrt
```

For processes that are not responsible for creating the mq but only accessing it, using `mq_getattr()` allows the maximum message size to be retrieved. That value can be used to dynamically allocate any necessary buffers.

If the mq was put into nonblocking mode, calling `mq_send()` will not block the process but will return `-1` with `errno` set to `EAGAIN`.

Example

Depicted below are two programs, `sender` and `receiver`. The `sender` application creates a mq (it must not already exist) and then writes a single message into it at priority 10. The `receiver` application opens the mq (which must already exist), reads the message and priority, and prints the result.

Neither application removes the mq. This allows the receiver to be executed multiple times. The first time, it will read the message added by `sender`. When executed again, it will produce an error message. This is because the mq is opened with `O_NONBLOCK` and there are no messages waiting to be read. Executing `sender` again in order to add another message will fail because of the `O_EXCL` flag passed to `mq_open()`.

There are multiple ways to adjust the example so that it operates differently, and the student should consider trying some of them to see the results:

- Remove `O_EXCL` from `sender` so that an existing mq can be opened.
- Remove `O_NONBLOCK` in `receiver` so that it waits for a message to be added. This one should be tried in various combinations with the previous one.
- Add `mq_unlink()` in `sender` prior to opening the mq.
- Add `mq_unlink()` in `receiver`, just prior to termination.

`sender.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <mqueue.h>

int main(void)
{
    mqd_t mqd = mq_open("queue", O_CREAT|O_EXCL|O_WRONLY, 0600, NULL);
    if (mqd < 0) {
        perror("Unable to create message queue");
        return 1;
    }

    const char msg[] = "Hello World!";

    // Send a message with priority 10, then close the queue.
    mq_send(mqd, msg, sizeof(msg), 10);
    mq_close(mqd);
}
```

receiver.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <mqueue.h>

int main(void)
{
    mqd_t mqd = mq_open("queue", O_RDONLY|O_NONBLOCK);
    if (mqd < 0) {
        perror("Unable to open message queue");
        return 1;
    }

    struct mq_attr attr;
    if (mq_getattr(mqd, &attr) < 0) {
        perror("Unable to get MQ attributes");
        return 2;
    }

    char *buffer = calloc(attr.mq_msgsize, sizeof(*buffer));
    if (!buffer) {
        perror("Unable to create space for message");
        return 3;
    }

    // Retrieve message from the queue and get its priority level
    unsigned int priority = 0;
    if ((mq_receive(mqd, buffer, attr.mq_msgsize, &priority)) < 0) {
        perror("Unable to get message");
    } else {
        printf("Received [priority %u]: '%s'\n", priority, buffer);
    }

    free(buffer);
    mq_close(mqd);
}
```

Summary

Students have learned how to use POSIX message queues for communicating data between processes.

- Opening, closing, and unlinking a message queue
- Message queue attributes
- Sending and receiving messages
- Message notifications

Appendix A: make

make

The `make` program is a tried and true approach towards building larger projects, or even building smaller ones more easily. It works by having a system of **targets**, **rules**, and **dependencies** that it chains together to carry out constructing a program. For single-file programs, there are probably no dependencies at all. This is the most basic `make` rule:

If the target asked for is the same as the name of a source file, build (compile and link) a program of that name from the source file.

The target requested is the argument given to `make` on the command line.

```
$ make hello-world
 c++ hello-world.cpp -o hello-world
```

Note that `make` defaults to printing out the steps that it takes to produce the requested target. This helps debug issues stemming from failures during the build process. Trying to build the program again will result in a message indicating that the action is not necessary:

```
$ make hello-world
make: 'hello-world' is up to date.
```

This is `make` checking the target (`hello-world`) against its dependency (`hello-world.cpp`), and discovering that no changes have been made to the source file since the program has been built. Since no changes have been made, there's no need to build the program again! This is how using `make` as a build system can reduce time spent waiting for the program to compile dependencies: it already knows which ones need to be built. Sometimes this behavior is not desired; a build can be forced by passing `make` the flag `-B`:

```
$ make -B hello-world
 c++ hello-world.cpp -o hello-world
```

`make` also knows how to build object files. So, the rule that it follows is

If the target asked for is the same as the name of a source file but ends in `.o`, compile an object file of that name from the source file.

```
$ make widget.o
 c++ -c -o widget.o widget.cpp
```

These two rules are known as **implicit `make` rules**. `make` always has these rules in mind when a target is requested. So, when requesting an object file target, or a program with the same name as a source file, no rules need to be written out.

Very few (good) programs are developed with default compiler settings. At the very least, a number of flags should be set that will warn the programmer of possible pitfalls or misunderstandings of the code they have written. At the very least, for nearly every program, a programmer should turn on the following warnings flags.

`-Wall`
: Turn on all (but not really) warnings.

`-Wextra`
: Turn on extra warnings.

`-Wpedantic`
: Turn on warnings that are very nitpicky.

If the compiler was being invoked directly, these flags would just be passed in on the command line.

```
$ c++ -Wall -Wextra -Wpedantic -o main main.cpp
```

Since `make` is a different program, it must be told in a different way that these flags are important for the compiler. `make` has special variables that it uses when executing its implicit rules. By placing these flags into the appropriate variable, the implicit rules will use them automatically.

```
$ make main CXXFLAGS+='-Wall -Wextra -Wpedantic'
 c++ -Wall -Wextra -Wpedantic main.cpp -o main
```

The target is passed to `make`, and then any variables that are used by `make` are adjusted. In the previous example, the compiler flags were added to the `CXXFLAGS` variable, which is updated (`+=`) after the target name. `make` will pull these variables' values from shell variables of the same as well, so the update could have been done at the start of the command. There are five `make` variables that are important to C++ programmers. These will be covered in more detail later.

`CPPFLAGS`
: Flags for the C/C++ preprocessor.

`CXXFLAGS`
: Flags for the C++ compiler.

`ASFLAGS`

Flags for the assembler (rarely used).

`LDFLAGS`

Flags for the linker.

`LDLIBS`

Libraries for the linker.

It is tedious to type out these flags for every single build of a program or object file. As such, `make` allows these variables to be set in a configuration file that will be checked every time `make` is invoked. This file is known as a `Makefile`.

`Makefile` *contents*

```
CXXFLAGS += -Wall -Wextra -Wpedantic
```

Putting this line into a file named `Makefile` in the same directory as the source code means that every `make` rule that needs to use C++ compiler flags will add on the requested flags.

```
$ make main
 c++ -Wall -Wextra -Wpedantic main.cpp -o main
```

Per-Target `make` Variables

It is not uncommon for a specific program to need an additional library or flag to build successfully. Suppose that a program requires an encryption library to be linked in.

```
$ make secret
 c++ secret.cpp -o secret -lscrypt
```

Here, the program links in a separate library, `scrypt`. A `Makefile` could be altered to add this to the `LDLIBS` variable. However, the five variables are *global*, and adding this library to `LDLIBS` in the same manner as adding the flags to `CXXFLAGS` would add this library to every single build in the current directory. This is probably not desired; instead only the `secret` program should be built with this library. It is possible to alter these variables on a per-target basis in the `Makefile`:

```
secret: LDLIBS += -lscrypt
```

Note that the form of the line is the target, followed by the variables to alter. To modify multiple variables, use multiple lines.

```
complex: LDFLAGS += -pie
complex: LDLIBS += -lscrypt -lbignum
```

Complex Target Dependencies

However, if a program is made up of multiple object files, then a rule should be written that describes the program's dependencies. This relationship is complex enough that rather than trying to cram it into a command line, it instead should be placed into the `Makefile` as a **rule**. Consider the earlier example:

```
$ ls
 main.cpp  ninja.h    poly.h    widget.h
 Makefile  ninja.cpp  poly.cpp  widget.cpp
 main.o    ninja.o    poly.o    widget.o
$ c++ -o main main.o ninja.o poly.o widget.o
```

The final binary (`main`) depends on all the object files. A `Makefile` rule is generally of the following form:

```
target: dependencies⋯
    actions
    ⋮
```

However, the *action*s may be omitted, in which case the implicit rules will be used, which is generally exactly what is desired. The rule to build the final `main` program would look like this:

```
main: main.o ninja.o poly.o widget.o
```

Note that the first dependency's name matches that of the target! This matching is what allows `make`'s implicit rules to take over:

```
$ ls
 main.cpp  ninja.h    poly.h    widget.h
 Makefile  ninja.cpp  poly.cpp  widget.cpp
$ make main
 c++ -Wall -Wextra -Wpedantic -o main.o main.cpp
 c++ -Wall -Wextra -Wpedantic -o ninja.o ninja.cpp
 c++ -Wall -Wextra -Wpedantic -o poly.o poly.cpp
 c++ -Wall -Wextra -Wpedantic -o widget.o widget.cpp
 cc main.o ninja.o poly.o widget.o -o main
main.o: In function `main':
main.cpp:(.text+0x19): undefined reference to ...
.
. # Elided error messages
.
collect2: error: ld returned 1 exit status
<builtin>: recipe for target 'main' failed
make: *** [main] Error 1
```

Because `make` knew that `main` depended on all the object files, it built them first, one at a time, using whatever was set in `CXXFLAGS` for each compilation. However, the final *linking step* failed. This can be determined by the third-last line, which states `error: ld returned 1 exit status`, `ld` being the canonical name of the linker. Looking at the command generated by `make` reveals the error: `make` invokes the C compiler program `cc` to do the final linking step, rather than `c++`. As such, the Standard Library and the `std` namespace of objects are not found.

There are three ways to fix this problem:

1. Add explicit actions for the rule:
   `c++ main.o ninja.o poly.o widget.o -o main`
2. Add the `-lstdc++` library to `LDLIBS`
3. Change the linker program to use a C++ linker: `LD = c++`.

Good `Makefile` design involves using *as few explicit rules or actions as possible*. Similarly, global settings or changes should generally be avoided. As such, the preferred way to solve this problem is the second solution:

`Makefile`

```
CXXFLAGS += -Wall -Wextra -Wpedantic

main: LDLIBS += -lstdc++
main: main.o ninja.o poly.o widget.o
```

This third line indicates what variable update needs to happen for the given target. The fourth line is the target and its dependencies.

```
$ ls
 main.cpp  ninja.h    poly.h    widget.h
 Makefile  ninja.cpp  poly.cpp  widget.cpp
$ make main
 c++ -Wall -Wextra -Wpedantic -o main.o main.cpp
 c++ -Wall -Wextra -Wpedantic -o ninja.o ninja.cpp
 c++ -Wall -Wextra -Wpedantic -o poly.o poly.cpp
 c++ -Wall -Wextra -Wpedantic -o widget.o widget.cpp
 cc main.o ninja.o poly.o widget.o -o main -lstdc++
```

Custom Actions

The default actions for targets tend to solve 90% of all cases, once the dependencies are described and default `make` variables set. But sometimes, in order to build a particular target, a custom action may be required. For instance, it may be useful to have a target that is a zipped archive of all source code.

```
$ ls
 main.cpp  ninja.h    poly.h    widget.h
 Makefile  ninja.cpp  poly.cpp  widget.cpp
$ make code.tgz
make: *** No rule to make target 'code.tgz'. Stop.
```

Here there exists a desired target, but `make` does not have a default rule for building said target. An explicit rule with a custom action is required in the `Makefile`.

```
code.tgz: *.cpp *.h
    tar -zcf code.tgz *.cpp *.h
```

Now, when `make` detects a more recent change in one of the dependencies, it will execute the custom action when this target is requested. It is extremely important to note that the whitespace in front of the custom action is *not* made up of spaces, but rather a tab character. `make` only understands actions that begin with such a character.

This rule could and should be improved. There is a lot of repetition in the action that was already written in the first part of the rule. As such, `make` has special variables to refer to different parts of the rule.

`$@`

The name of the current target

`$<`

The first dependency

`$^`

List of all dependencies

`$?`

List of changed dependencies

Which allows the rule's action to be written in a much more robust and reusable way as:

```
code.tgz: *.cpp *.h
    tar -zcf $@ $^
```

Cleaning Up After Builds

The act of building software can leave a number of files, in various states of use, lying about in the current directory. It is an extremely good habit to offer a target that will clean up the current directory, removing all generated files. This target is traditionally called `clean`.

```
$ ls
 main.cpp  ninja.h    poly.h    widget.h
 Makefile  ninja.cpp  poly.cpp  widget.cpp
 main.o    ninja.o    poly.o    widget.o
$ make clean
 rm -f main *.o
```

Unfortunately, this needs to be written as a custom rule, since `make` does not know which files to clean up by default.

`Makefile`

```
TARGET=main
clean:
    $(RM) $(TARGET) *.o
```

In this case, there are two variables being used. One of them, `RM`, is given a default by `make`, while the other, `TARGET`, was defined for purposes of this `Makefile`. The `RM` variable holds the name of a program that will

remove files from disk; in this case, `rm{nsbp}-f`. Many of `make`'s implicit rules will use variables in some way. Following is a list of some of `make's implicit rules, and some of the default implicit variables.

Target	Given	Action
`n`	`n.o`	`$(CC) $(LDFLAGS) n.o $(LDLIBS) -o n`
`n`	`n.c`	`$(CC) $(CPPFLAGS) $(CFLAGS) $(LDFLAGS) n.c $(LDLIBS) -o n`

`n`	`n.cpp`
	`n.cc`
	`n.C`
		`$(CXX) $(CPPFLAGS) $(CXXFLAGS) $(LDFLAGS) n.cpp $(LDLIBS) -o n`

`n.o`	`n.c`	`$(CC) $(CPPFLAGS) $(CFLAGS) -c n.c -o n.o`

`n.o`	`n.cpp`
	`n.cc`
	`n.C`
		`$(CXX) $(CPPFLAGS) $(CXXFLAGS) -c n.cpp -o n.o`

`n.o`	`n.s`	`$(AS) $(ASFLAGS) n.s -o n.o`
`n.s`	`n.S`	`$(CPP) $(CPPFLAGS)`

`AR`	`ar`	Archiver
`AS`	`as`	Assembler
`CC`	`cc`	C compiler
`CXX`	`c++`	C++ compiler
`CPP`	`cpp`	C preprocessor
`RM`	`rm -f`	Removal command

So, when the target `clean` is requested, `make` sees that there is no file called `clean`, and that building it is not dependent on any other targets or files, so it runs the custom action, removing all generated files.

What if there was a file called clean already in the directory? This illustrates a problem with `make` targets that are intended to be *procedures* rather than *files*.

```
$ ls
 main.cpp  ninja.h    poly.h    widget.h
 Makefile  ninja.cpp  poly.cpp  widget.cpp
 main.o    clean      poly.o    widget.o
$ make clean
make: 'clean' is up to date.
```

As such, any target that is not a file should have an entry as a dependency in a special `.PHONY` target in the `Makefile`.

`Makefile` *'clean' Target*

```
.PHONY: clean

clean:
    $(RM) $(TARGET) *.o
```

This special target is just a listing of other targets that are “phony”, they do not generate a file by the same name as their target. So, if that target is requested, it is always run, even when a file by that name exists.

Other common phony targets include:

`all`

: Build every target

`clean`

: Remove `make`-generated files

`distclean`

: Like `clean`, but removes additional files, usually created by non-`make` tools or downloads

`install`

: Install the program on the computer; ususally needs to be run as user `root`

The Default Target

It is possible to invoke `make` with no specified target whatsoever. In this case, `make` will assume that the first listed target in the `Makefile` is what is desired, and build that target. So, it is usual to make the first target the desired binary program.

Key Ideas

- `make` is a build system.
- A `make` rule is composed of a target, its dependencies, and actions to build that target.
- `make` has a number of implicit rules that it follows to build requested targets.
- Given a `.c` or `.cpp` file, `make` can build a `.o` file of the same name, called an object file.
- Given a `.c` or `.cpp` file, `make` can build an binary program of the same name.
- `make`'s implicit rules can be augmented by adjusting certain variables that it uses to build.
- The `Makefile` can be used to store custom `make` rules, variables, and dependencies.

Appendix B: Bits

All values in programming are represented by bits. Whether the value in question is an integer, a letter, a floating-point value, or a pointer to something else is purely based on **how those bits are intepreted**.

It is the responsibility of the programmer to interpret bits correctly. A programming language like C or C++ can bring such misinterpretations to light, requiring the enterprising programmer to explicitly cast or convert bits of one interpretation to another.

Integer Values

Integer values interpret bits based on binary powers of two.

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	0	1	0	1	0	1	0	1
	128	64	32	16	8	4	2	1
85	=	64	+	16	+	4	+	1

One of the useful properties of binary is how simple the arithmetic tables are.

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	0	1	0	1	0	1	0	1
+	0	0	0	1	1	1	1	1
	0	1	1	1	0	1	0	0

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	0	0	0	1	0	1	0	1
x	0	0	0	0	0	1	1	1
	0	0	0	1	0	1	0	1
	0	0	1	0	1	0	1	0
	0	1	0	1	0	1	0	0
	1	0	0	1	0	0	1	1

Unsigned Integers

The easiest way to work with integers is treating them as unsigned integers. This means the value of such an integer ranges between 0 and 2^N-1, where N is the number of bits or binary places in the number.

Signed Integers—Two's Complement

There are many ways of implementing signed integers; integers that may be negative or positive. C++ uses **two's complement** encoding to interpret signed integers.

In two's complement, the most significant bit is considered to be negative, and all other bits are still positive.

	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	0	1	0	1	0	1	0	1
	-128	64	32	16	8	4	2	1
85	=	64	+	16	+	4	+	1

	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	1	0	1	0	1	0	1	0
	-128	64	32	16	8	4	2	1
-86	-128	+	32	+	8	+	2	

Character Values—Encodings

By associating any enumerable set with the integers, that set can be represented in binary. The most common set is that of letters or characters. This mapping of integers to characters is called an **encoding**.

The most well-known encoding is ASCII, the American Standard Code for Information Interchange. ASCII encodes digits, some punctuation, and the English alphabet in bother upper- and lower-case letters. In addition, there are a number of nonprinting entities designed to control devices or organize data.

Since other languages have other letters or glyphs, ASCII is insufficient to represent all possible means of communicating. While there are many other encodings for specific character sets (some written languages may even have multiple competing encodings), the emerging solution is that of Unicode.

Unicode is designed to be a universal encoding for all possible character sets. It currently has capacity for over one million distinct characters, of which about 140,000 are used (as of Unicode 13.0, published March 2020).

Unicode is merely the enumeration of characters, mapping them to integer values called **code points**. The actual encoding of those numbers into binary representation is a bit more fractured. Some encodings require that every glyph or character uses 32 bits, even if most of those bits are 0. Some encodings require up to 48 bits.

Pointers

Pointers in C and C++ are explicitly not integer values, but are intended to be abstracted away as black boxes. In reality, they can be treated like unsigned integers, especially at the level of assembly language.

Any register or memory location could hold what is intended to be a pointer to some other location in memory. The `NULL` pointer is defined as being the integer value 0.

Floating-Point Values

IEEE Floating-Point values sacrifice some simplicity, precision, and speed at additions for a larger range of numeric values and increased speed at multiplication and trigonometric calculations.

IEEE Floating-Point Numbers are scientific notation, but in binary. A number in scientific notation is of the form

6.02214×10^{23}

It is composed of a **mantissa**, the decimal number, multiplied by the number 10 raised to an **exponent**. The exponent may be any integer, and the mantissa is in the range (-10, -1], [1, 10). Note that in base-ten, decimal, the mantissa is limited to being less than the base, and the exponent is applied to the base. A number in this form is more compact than 602,214,000,000,000,000,000,000, and also carries with it the level of precision (in this case, 6 decimal digits).

For binary, therefore, a similar number would look like this:

$1.11111110000110000{1}_2 \times 10_{two}^{1001110_2}$

The mantissa is still clamped to (-10_{two}, -1], [1, 10_{two}), which means that the leading digit of the mantissa must always be 1. The same amount of precision, one part in a million, requires more binary digits.

A floating-point number encodes the mantissa and exponent in a given set of bits, with a few optimizations.

S	Exponent								Mantissa																			

Since the leading digit of the mantissa is always 1, it can be presumed to always be 1, and omitted from the encoding. The sign of the mantissa (positive or negative) is encoded as a single bit in its own section.

S	Exponent								Mantissa																			
0									1	1	1	1	1	1	1	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0

To express both positive and negative exponents, a **bias** is added to the scientific notation's exponent before being stored in the encoding. When the number is retrieved, the bias is subtracted. This ends up being more efficient for calculations than using two's-complement for signing the exponent. (For 32-bit floats, the bias is 127.)

S	Exponent								Mantissa																							
0	1	1	0	0	1	1	0	1	1	1	1	1	1	1	1	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0

Programmers are not expected to manipulate these bits directly; but it is important to understand that floats are limited in the amount of precision that they allow. Different floating-point formats allow for more bits of precision.

Type	Exponent Bits	Mantissa Bits	Decimal Digits
`float`	8	23	7
`double`	11	52	15
`long double`	15	64	18

Key Ideas

- The same sequence of bits can be interpreted in multiple ways.
- Nonnegative integers are represented as the sum of powers of two for each bit.
- Negative integers are usually encoded as **twos-complement**, which means that the most significant bit adds its value as a negative instead of a positive.
- Character values have a variety of encodings, but are based on the idea of each encoded value representing a glyph.
- Unicode is a de facto “universal enumeration” of characters.
- Unicode has a number of different encodings of its enumerated symbols.
- Floating-point values are composed of sequential binary fields reprsenting parts of the number.
- Floating-point numbers have a limited degree of accuracy; larger formats have higher accuracy.

Exercises

1. Which ASCII character is 0x61?
2. What is 0b10001011 as a signed byte value?
3. What is the value of 0b00110101 ?
4. Read up on the Patriot Missile Failure at Dhahran. How many hours would it take for the error to be 1 second?