POSIX Programming

Preface:

This document covers many, but nowhere near all of the POSIX APIs and syscalls. I strongly implore the reader to have a look through the POSIX reference maintained by The Open Group: <https://pubs.opengroup.org/onlinepubs/9699919799/nframe.html>. Have a look through the alphabetical listings for all of the function definitions and get an idea of what is available to you as a system’s developer writing in C.

Introduction:

This document is intended to be an extension of my C programming notes. POSIX, if you are not familiar, is a standard which most Unix/BSD/Linux distributions comply with. It aids with the portability of code and forces the maintainer to provide specific functionality so that end users can expect a certain set of uniform behaviour between platforms. If you use Linux or any other Unix-like OS, you may have been under the impression that POSIX was a standard for shell scripts. While this is true, POSIX compliancy applies to the entire operating system, and is more broad in scope than simply a standard for shell scripts. The intent of the C Programming notes was to familiarize the new C programmer with the syntax and common-use functions. The intent of this document is to shed light on more POSIX-defined functions so that you may know how to interface with your operating system through code.

In the Beginning:

In the 1960’s the personal computer as we know it did not exist. Large corporations like IBM and other government funded organizations only had access to mainframes - monolithic pieces of machinery, typically using vaccum tubes and/or punch cards, and eventually terminals with teletypewriters for entering commands.

Assembly code had to be written for each individual computer (e.g., the GE 645 Mainframe). As the complexity of these mainframes grew, so did the assembly code that had to be written for them. These machines could only run 1 program at a time. You would load the program into memory using a large stack of punch cards. The computer would crunch the numbers and spit our the results to a tape or printer. Eventually, as technology advanced, multiple users could connect to a single computer using teletypewritters. Computers could run multiple programs at the same time. Now concepts arose such as sheduling and prioritization.

Originally, Unix was written in assembly, but after Ken Thompson and Dennis Richie had developed the C programming language, they rewrote Unix in 1973 in C. Many standards for writing code that would interface with hardware began to spring up. Some examples include VAX VMS, CP/M, TRON, ITRON, MuITRON, etc. Eventually POSIX was born amongst the many standardizationsn, and would ultimately overrule its competitors.

POSIX:

POSIX stands for Portable Operating System Interface (the x is sort of just there for reasons that aren’t very profound). This standard began in 1985 and was officially under the IEEE as IEEE 1003.1-1988 and ISO/IEC 9945. But what does it mean to be “compliant” with the POSIX standard? In a nutshell, it indicates that compilation and linking should function in the same manner across platforms. The runtime behaviour of programs is expected to run the same for the most part, but that doesn’t mean that the behaviour must be *identical* across all platforms.

Before 1997, POSIX was divided into parts:

**POSIX.1** - Core services (1988)

**POSIX.1b** - Realtime extensions (1993)

**POSIX.1c** - Thread extensions (1995)

**POSIX.2** - Shell and utilities (1992)

Resources and Handles:

A system resource is defined to be a virtual component with limited availability. System resources might include command line parameters, environment variables, timers, file descriptors, signal handlers, and the program stack.

A “handle” typically refers to a pointer which points to a system resource. So for instance, if I mention that a function takes in a file handle as a parameter, this most likely just means that the function accepts some reference/pointer to a file.

Processes:

I’m embarrassed to admit the amount of time it took for me to actually understand what a process is. Obviously, I’m sure we’ve all heard of “processes”, but do we actually understand what a process is? In simple terms, a process is an instance of a running program. In other words, it is a program in execution. In Linux, each process is ascribed a unique Process ID, or PID for short. All processes must have at least one thread, but if a process uses more than one thread, it is considered to be a multi-threaded process. Threads are short for “thread of execution”. It is almost certain at the time of writing this document that you have a processor which is multi-core. That means that the physical unit that is your CPU is split internally into multiple physical dies (CPUs) called cores. Therefore, if a CPU has 4 cores, then 4 threads can execute in parallel i.e., at the same time (the exceptions to this rule are Intel’s hyperthreading and AMD’s Simultaneous Multi-Threading (SMT) which allow each core to run two threads in parallel). Note: two processes running in parallel is different from them running concurrently. A process which requests 4 threads could very well be using all 4 of the CPU’s cores at a time. CPU cores can only truly run one thread of execution at a time (unless they are hyperthreaded which will will discuss later). Threads also have Thread IDs or TIDs for short. Thread IDs are unique to their parent process, but not necessarily unique across the system (unlike PIDs). Processes can contain system resources which are shared amongst the processes threads.

Creating Processes:

While not really super relevant to multithreading, it seems appropriate to discuss how processes are creating while we’re on the subject. When you boot up your PC, a bunch of harware initialization and checks occur, and eventually, depending on your boot firmware (BIOS or UEFI), some region of disk space is loaded and executed. In the case of Linux systems, the program that is executed is often times your boot loader (usually GRUB). GRUB then starts the init system which is the initial process of your OS. The init system always has a PID of 0. From here, the init system creates children processes using the system call fork().

fork(): pid\_t fork(void);

The fork() system call creates a clone of the parent process. This newly created process is known as the child process, and the calling process is known as the parent process. The child process will be allocated stack space that is separate from the parent’s stack space, but the contents of the parent’s stack will be copied to the child’s stack. This does not mean that they are exactly the same in every regard (see man pages). For instance, the child will receive a unique PID, and regardless of whether or not the parent was multithreaded, the child process will start with a single thread of execution. This is because it is too difficult to copy over the state of threads to a child process, and therefore, calling fork() on multithreaded applications is discouraged.

#include <stdio.h>

#include <unistd.h>

int main(void) {

const pid\_t pid = fork();

if (-1 != pid) {

if (0 == pid) {

printf("Child PID: %d\n", getpid());

} else {

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

}

}

}

// Output:

Parent PID 15323

Child PID 15324

Let’s analyse the output from above. As soon as fork() is invoked, the process’ stack along with all of its resources (file descriptors, priority, signal mask, I/O privillege, etc.) are copied into a new stack space which is allocated for the child process. The program’s stack pointer will not change, so we resume execution in both processes immediately. In the parent process, fork() returns the PID of the new child process, whereas in the child process, fork() returns 0. If fork() were to fail for whatever reason (e.g., not enough stack space), then it would return -1 and the parent process would simply terminate. Assuming that fork() succeeded, the parent process will output its PID and the child will output its PID (both retreived from a call to getpid()). If you run the program above multiple times, you will note that sometimes the parent exits first, and sometimes the child exits first. The order in which the print statements are executed within the example above is essentially arbitrary. For other programs that invoke fork(), other factors may play into this, such as the fact that the parent is still capable of using multiple threads, whereas the child only begins with one (it can create more afterwards, but it only begins with one), or depending on how long the thread is blocked for e.g., if the program requires user input.

exec\*():

The exec family of functions work in unison with fork(). Here is the of exec functions before I continue with the explanations of what they do:

- int execl(const char \*path, const char \*arg0, ..., (char \*)0);

- int execle(const char \*path, const char \*arg0, ..., (char \*)0, char \*const envp[]);

- int execlp(const char \*file, const char \*arg0, ..., (char \*)0);

- int execv(const char \*path, char \*const argv[]);

- int execve(const char \*path, char \*const argv[], char \*const envp[]);

- int execvp(const char \*file, char \*const argv[]);

- int fexecve(int fd, char \*const argv[], char \*const envp[]);

It may be overwhelming to see this many versions of one function, but I assure you, their differences are not too hard to commit to memory. Before delving into the differences of each, allow me to explain the general idea behind exec\*(). A useful resource in systems programming is the ability to be able to run programs from within an already running program. fork() provides us with the ability to copy the running process, which in turn, allocates a brand new stack for us to use. But what if we could use the stack that was allocated for this new child process for different program entirely? This is essentially what exec does. Think of it almost like memcpy(), where we copy data into an existing buffer, overwriting the previous contents, but this time, we’re doing it with processes.

Let’s look at the subtle differences of each version of exec. First is execl(). This function takes a path to the binary of the application that we want to copy into our stack created with fork() as its first argument. The second argument is the mandatory 0th argument for the program, followed by a vararg list of additional options to give the program. The list of varargs must be NULL terminated i.e., the last argument must be NULL. Note that in linux, the 0th argument is always the name of the program. Next I’ll cover execlp(). This function is identical to execl(), with the only difference being that it will search the user’s $PATH environment variable for the binary. This means that you no longer have to provide the absolute path to where the program is located, assuming that it exists somewhere in the user’s $PATH. Next is execv(). This version is again, identical to execl(), with the main difference being that instead of a list of comma separated varargs, we provide an array of arguments instead. Note that the array of arguments must still be NULL terminated. execvp(), as you can probably guess, combines the utility of execlp() and execv() into one. execle() is again, identical to execl(), with the difference being that there is now an additional mandatory argument which comes after the NULL terminator for the program arguments vararg list. This mandatory argument (env) is a another array, similar to execv(), but for exporting environment variables at runtime. The syntax for this is how you would expect when using the export command on Linux. For example: char \*const env[] = {“HOSTNAME=[www.mysite.com](http://www.mysite.com/)”, “PORT=8080”, NULL};. This will export the environment variables HOSTNAME and PORT with their respective assigned values. execve() combines the functionality of execv() and of execle(). Finally is fexecve(). This is identical to execve(), but rather than executing a binary ELF file, it executes whichever file is associated with the file descriptor supplied in the fd argument. The two caveats with this are that 1. the file descriptor must be opened as O\_RDONLY or O\_PATH, and 2. the caller must have permission to execute the file referenced by fd.

Note that child processes which do not call one of the exec functions, or that call exec but it fails, should invoke \_exit() rather than exit(). This is so that the child process does not interfere with the parent process’ external data (files) by calling its atexit handlers, signal handlers, and/or flushing buffers.

wait() & waitpid(): pid\_t wait(int \*stat\_loc);

pid\_t waitpid(pid\_t pid, int \*stat\_loc, int options);

When a child process created by fork() is created, it will innevitably die (exit) at some point. The following scenarios assume that the child process dies before the parent dies. When a child process dies, it sends out a SIGCHLD signal to the parent process. The parent process is able to invoke wait() or waitpid() to determine the reason for why the child process died. In the case that the parent does not invoke wait() or waitpid(), the child frees most of its resources, but its PID remains in the process table in order to hold its exit status. This is called a zombie process. Zombie processes use no CPU, as they are not running. wait() causes the calling thread to become blocked until either any child returns its exit status, or another signal is sent telling the parent process to terminate the child. If the stat\_loc argument is not NULL, information about the child’s exit status will be stored at the location pointed to by stat\_loc. stat\_loc will contain 0 if and only if one of the following conditions is true:

1. The child process returned 0 from main()

2. The child process called \_exit() or exit() with a status of 0

3. The process was terminated because the last thread in the process was terminated

If the stat\_loc argument *is* NULL, the return status info is discarded and all zombie processes which are descendants of the parent are cleared. waitpid() waits for any child process to return its exit status if the argument for pid is equal to -1. In this regard, it is the same as a call to wait(). If the argument for pid is greater than 0, it specifies the PID of a single child process for which status is requested. The stat\_loc argument functions in the same manner as it does in wait(). The options argument can be 0 if no flags are desired, or 1 or more of the following flags, joined with a bitwise OR:

WCONTINUED: Report the status of any continued child specified by pid whose status has not been reported since it continued from a job control stop.

WNOHANG: waitpid() shall not suspend execution of the calling thread if status is not immediately available for one of the child processes specified by pid.

WUNTRUNCATED: The status of any child process specified by pid that are stopped, and whose status has not yet been reported since they stopped, shall also be reported to the requesting process.

Note that zombie processes can be avoided entirely by invoking signal() like so: signal(SIGCHLD, SIG\_IGN); This tells the process to ignore the SIGCHLD signal completely, and thus, the child will just die instead of becoming a zombie.

vfork(): pid\_t vfork(void);

vfork() is an outdated function that probably should not be used anymore, but I figured we might as well discuss it. The idea behind vfork() was to optimize the rare case where you would want to call fork() immediately followed by execve(). The idea is that execve() must copy over the entire process to the newly reserved process space which can be a heavy operation. vfork() attempted to solve this by blocking the parent process’ calling thread and using its memory (including the stack) for operations within the child process instead of duplicating it. This blocking of the parent process’ calling thread occurs until the child either calls execve(), \_exit(), or terminates abnormally (e.g. via a signal). Nowadays, fork() has had many improvements, the most notable being it’s ability to do copy-on-write (COW). The way that COW works, is that when a program requests to do a write operation, small pages of data are copied from the process to the process’ address space right before the write occurs, rather than copying over the entire process all at once. In other words, we only copy over what’s necessary when it’s necessary to do so. Long story short, vfork() is essentially deprecated.

Here is an example program showcasing fork(), execl(), and waitpid():

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <signal.h>

#include <sys/wait.h>

void chld\_died(int status) {

printf("Child died with status: %d\n");

}

int main(void) {

pid\_t pid = fork();

if (pid != -1) {

if (pid == 0) {

execlp("/usr/bin/sleep", "sleep", "10", NULL);

}

else {

signal(SIGCHLD, chld\_died);

waitpid(pid, NULL, 0);

}

}

return 0;

}

In this example, we fork the parent process and create a new child process. Remember that fork() returns 0 for the child process, so it invokes execl() and replaces itself with the binary ELF for the sleep utility. The second argument (10) is the amount of seconds to sleep for. Meanwhile, in the parent process we set up a signal handler using signal() to run a print statement and notify us when the child dies. Immediately afterwards, we call waitpid() which blocks the parent process until the child process dies (which happens after 10 seconds). Once 10 seconds has expired, the child process calls exit() and the parent process can continue execution (in this case there is nothing left to execute, so it terminates).

posix\_spawn(): int posix\_spawn(

pid\_t \*restrict pid,

const char \*restrict path,

const posix\_spawn\_file\_actions\_t \*file\_actions,

const posix\_spawnattr\_t \*restrict attrp,

char \*const argv[restrict],

char \*const envp[restrict]);

One function that still is used as an alternative to exec() is posix\_spawn(). The main difference between the exec() functions and posix\_spawn() is that we can specify file descriptors that we want the child process to inherit from the parent.

File Operations:

As you know, the underlying mechanism used by POSIX systems for file maintenance is the file descriptor. A file descriptor is simply an unsigned integer that must fall within a specific range and acts as a handle to a region of memory where the file’s contents are stored. File descriptors are considered as system resources, thus each process that is created either takes or shares ownership over certain file descriptors. The stdin, stdout, and stderr file descriptors (which are always equal to 0, 1, and 2, respectively) are shared by each running process on the system and are reserved so that they are not overwritten. There are certain cases in which it is beneficial to duplicate a file descriptor, the most popular example being redirection. Say that you want stdin to be redirected to a file. We can duplicate the stdin file descriptor (0) to another file descriptor (e.g. 5). Now we have two file descriptors, 0 and 5, which both point to stdin. We can open file descriptor 5 as a file within our program so that during the programs lifetime, anything that the user enters into the command line also gets written to the file. Other examples of where this is useful include reading from stdout e.g. if you wanted to compare the output of the program with a variable in a unit test, or if you wanted to pipe the output of one program to another. So how is this accomplished? Traditionally, the fcntl() (file control) function was used for this purpose. Two helper functions, dup() and dup2() achieve the same results (as they invoke fcntl() under the hood), but with greater simplicity. I want to first look at the fcntl() function by itself considering that it can do more aside from just duplicate file descriptors. After that, I’ll touch upon the dup() and dup2() functions with an example of how they can be used effectively.

fcntl(): int fcntl(int filedes, int cmd, ...)

The fcntl() function (which stands for file control) resides in its own header file (<fcntl.h>). The fcntl() API expects a file descriptor as its first argument, which is the file descriptor of the file we wish to operate on. The cmd argument can be any of the commands defined in fcntl.h. These include things like F\_DUPFD to duplicate a file descriptor, F\_GETFD and F\_SETFD to get/set a file descriptor, F\_GETOWN or F\_SETOWN to get/set file ownership, and F\_GETLK or F\_SETLK to get or set read/write locks on a file.

dup() and dup2(): int dup(int filedes);

int dup2(int filedes, int filedes2);

The dup() an dup2() APIs are wrappers around fcntl() which utilize the F\_DUPFD command. In fact, dup() is equivallent to calling fcntl(fd, F\_DUPFD);. The difference between dup() and dup2() is that dup() binds and returns the lowest available file descriptor, whereas dup2() allows us to specify the number of the file descriptor explicitly in the second parameter. Note that when we call dup() or dup2(), the close on exec flag (FD\_CLOEXEC) of the original file is not transfered to the duplicated file descriptor.

pipe(): int pipe(int filedes[2]);

The pipe() API lets us create a Unix pipe. You are perhaps familiar with the pipe operator (|) if you’ve used the CLI before. Unix pipes allow us pass data bi-directionally between processes. Passing data between processes is formally known as inter-process communication (IPC). The pipe() function accepts an integer array with a length of 2. Two file descriptors will be inserted into the array – the first file descriptor being the read end of the pipe is placed as the first element within the array, followed by the write end of the pipe as the second element. The Linux man page provides a great usage example:

#include <[sys/wait.h](https://linux.die.net/include/sys/wait.h)>

#include <[stdio.h](https://linux.die.net/include/stdio.h)>

#include <[stdlib.h](https://linux.die.net/include/stdlib.h)>

#include <[unistd.h](https://linux.die.net/include/unistd.h)>

#include <[string.h](https://linux.die.net/include/string.h)>

int main(int argc, char \*argv[])

{

int pipefd[2];

pid\_t cpid;

char buf;

if (argc != 2) {

fprintf(stderr, "Usage: %s <string>\n", argv[0]);

exit(EXIT\_FAILURE);

}

if (pipe(pipefd) == -1) {

perror("pipe");

exit(EXIT\_FAILURE);

}

cpid = fork();

if (cpid == -1) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (cpid == 0) { /\* Child reads from pipe \*/

close(pipefd[1]); /\* Close unused write end \*/

while (read(pipefd[0], &buf, 1) > 0)

write(STDOUT\_FILENO, &buf, 1);

write(STDOUT\_FILENO, "\n", 1);

close(pipefd[0]);

\_exit(EXIT\_SUCCESS);

} else { /\* Parent writes argv[1] to pipe \*/

close(pipefd[0]); /\* Close unused read end \*/

write(pipefd[1], argv[1], strlen(argv[1]));

close(pipefd[1]); /\* Reader will see EOF \*/

wait(NULL); /\* Wait for child \*/

exit(EXIT\_SUCCESS);

}

}

This example demonstrates a lot of the things we’ve learned so far. We supply our int array of length 2 (called pipefd) into the pipe() function. Index 0 of the array now contains the file descriptor of the read end and index 1 contains the file descriptor for the write end. We fork the current process, which spawns a new child process. In the child process we close the unused write descriptor, and vice versa in the parent process. While there are still bytes remaining in the read end of the pipe, we read them and write them to stdout. Note the use of the \_exit() syscall in the child process as opposed to the regular exit() command. In the parent process, we close the unused read end and write whatever data was passed into the program as a CLI argument to the pipe. We wait for the child process to die and then exit normally.

Named Pipes/FIFOs:

The pipe() function that we looked at previously creates an anonymous pipe. The anonymous aspect of anonymous pipes is derived from the fact that anonymous pipes are not backed by a file. This means that the data is streamed directly from the write end to the read end (usually utilizing kernel buffers for efficiency) and then the data is discarded once received. A named pipe, also known as a FIFO, is a file type on Unix and Unix-like systems which functions similarly to a pipe, but can persist due to the fact that the data is stored in a temporary file. This temporary file can be opened using the open() syscall by other processes, similar to other files. On Linux, we can create a FIFO using the mknod or mkfifo commands.

POSIX defines the mknod() and mkfifo() APIs so that we can create named pipes programatically. All files in Linux can be uniquely identified by their index nodes (inodes). You can view the inode of a file using the stat command, which we’ll look at shortly. The mknod() API is more broad than mkfifo() since it enables us to create block, character, or FIFO files. Only FIFOs happen to be portable, however, whereas other types may not be compatible with other POSIX compliant systems. In order to use the mknod() function, we simply provide a path to the file that should back the named pipe, a series of flags – one of which must be the type of file that we want to create, and then a dev\_t object which is only used if you’re making a block or character device and ought to be left as 0 if making a FIFO. Here’s an example of using mknod():

#include <stdlib.h>

#include <sys/types.h>

#include <sys/stat.h>

int main(void) {

int status;

dev\_t dev = 0;

status = mknod(“/home/user/foo”, (S\_IFIFO | S\_IWUSR | S\_IRUSR | S\_IRGRP | S\_IROTH), dev);

if (status != 0) {

perror(“Failed to create fifo”);

exit(EXIT\_FAILURE);

}

return EXIT\_SUCCESS;

}

The man pages for mknod() recommend using mkfifo() instead of mknod() for creating FIFO/named pipes, however, I wanted to introduce mknod() for the sake of coverage. mkfifo() works exactly the same as mknod(), but without the final dev argument.

ioctl: int ioctl(int filedes, int request, ... /\* arg \*/)

fstat(): int fstat(int fildes, struct stat \*buf);

fstat is likely the simplest way of obtaining information regarding a file given a file descriptor. The file’s information will be stored in a stat struct referenced by buf. E.g:

int fdopen();

struct stat file\_stat;  
fstat(

Process Lifetime:

When a process is destroyed (either the process exits normally, or exits abnormally and becomes a zombie process, and then gets queried by some other process), then the process is destroyed and all OS resources that were being used by the process are returned for reuse. This includes physical memory (the VAP is destroyed), file descriptors are closed, timers, mutexes, semaphores, etc. are destroyed. Take the following code for example:

/\* On a computer with 16G of memory,

and exactly 15G of memory free,

how many times will this successfully run? \*/

#define ONE\_GIG ((size\_t)1 \* 1024 \* 1024 \* 1024)

int main(void) {

void const \* const p = malloc(ONE\_GIG);

return (NULL == p) ? EXIT\_FAILURE : EXIT\_SUCCESS;

}

The answer to this question is that the program will run an infinite amount of times. Despite the program (pressumably) allocating one gigabyte of memory using malloc each time that it is executed, once the process is destroyed, that memory is returned to the OS for re-use, thus we never run out of memory. Additionally, malloc() uses COW mappings, so even if this code was in an infinite while() loop, it would presumably still run forever, since we never actually write to the address contained in p.

Thread Scheduling:

Since we have many threads competing to be ran by the CPU, each thread must be assigned a priority to specify which one should get executed first. This does not fully solve the issue, however, as multiple threads may be assigned the same priority. Because of this, we have various algorithms that a thread may be assigned. The two that we will focus on are FIFO and Round Robin (RR). If a thread is given FIFO scheduling, that means that the thread can execute as long as it wishes to run for. The thread may call sleep() which allows other threads to have their turn at executing, but once sleep() returns, the FIFO thread can resume and take up as much time as it wants. The RR scheduling algorithm is a bit more common. The idea is that each thread can only run for a certain N amount of “scheduling quanta”. For example, a thread may be given 4ms before it must pass on execution to the next thread in the priority queue.

Thread Pre-empting:

Running threads can be *pre-empted* by threads with a higher priority. On a slightly more technical level, we would say that a thread in the RUNNING state can be preempted by a thread of higher priority that is in the READY state. A thread is considered to be in the RUNNING state if it is currently executing tasks, and a thread is in the READY state if it is in the priority queue waiting to run, but is not currently running. When we say that a thread can be preempted, we mean that it can be interrupted by a process of a higher priority. The state of the RUNNING thread is saved, then the higher priority thread runs until it either finishes, runs out of scheduling quanta (if it has the RR algorithm) where it is put at the back of the priority queue, is preempted by a thread that has an even higher priority, or exits abnormally. Once all higher priority threads have finished executing, the state of the original RUNNING thread is restored and it can proceed.

pthread\_create(): int pthread\_create(

pthread\_t \*restrict thread,

const pthread\_attr\_t \*restrict attr,

void \*(\*start\_routine)(void\*),

void \*restrict arg);

In order to create a new thread, we can use the POSIX function pthread\_create(). The first parameter would more aptly be called “tid” because this typedef is really just an int pointer that will store the TID upon successful creation of the new thread. The attr parameter can be a pthread\_attr\_t struct, which contains certain members like the scheduling priority of the thread. If left NULL, a default attribute struct will be used. The third parameter expects a function pointer. This function pointer can be of any return type since the parameter is of type void\*. If provided, the thread will execute the specified function. Threads always begin in a function called start\_routine(). This is equivallent to main() for programs. If the third argument is given a value of NULL, then the thread will only execute start\_routine(). The final parameter specifies miscellaneous data of your choosing to be passed to start\_routine().

pthread\_attr\_t:

Let’s delve a bit more into that pthread\_attr\_t struct that we can pass into pthread\_create(). There are quite a few functions that can be used to initialize and setup this struct. These functions are listed below:

- pthread\_attr\_init(): Initializes pthread\_attr\_t struct.

- pthread\_attr\_destroy(): Destroys pthread\_attr\_t struct.

- pthread\_attr\_setdetachstate(): Set detach state of pthread\_attr\_t

- pthread\_attr\_setinheritsched(): Specifies that the thread attributes shall be set to the corresponding values from this attributes object.

- pthread\_attr\_setchedparam(): Sets scheduling priority of the pthread\_attr\_t struct.

- pthread\_attr\_setschedpolicy(): Specifies the scheduling algorithm (SCHED\_FIFO, SCHED\_RR, SCHEDOTHER).

- pthread\_attr\_setstacksize(): Sets thread creation stack size.

This may seem like a lot, but it’s really just one function per attribute in the pthread\_attr\_t struct. An example of setting up a pthread\_attr\_t struct can be seen below:

pthread\_attr\_t attr; /\* Attribute struct for thread creation \*/

struct sched\_param param;

pthread\_attr\_setinheritsched (&attr, PTHREAD\_EXPLICIT\_SCHED); /\*Manually specify sched\*/

param.sched\_priority = 15; /\* Threads prio. \*/

pthread\_attr\_setschedparam (&attr, &param); /\* Set prio. of pthread\_attr\_t \*/

pthread\_attr\_setschedpolicy (&attr, SCHED\_RR); /\* Set sched algorithm to Round Robin \*/

pthread\_create (NULL, &attr, func, arg); /\* Create thread using pthread\_attr\_t struct \*/

Thread Operations:

I bet you’d love to see even more functions to remember. Don’t worry, they will actually stick in your mind easier than you’d think. The following functions all operate on the calling thread. If the thread’s function parameter was set, then calling these functions will operate on the thread associated with the calling function, otherwise, the operation may be applied to any random thread.

- pthread\_exit(retval): Terminate the calling thread

- pthread\_join(tid, &retval): Wait for thread to die and get the return value in retval

- pthread\_kill(tid, signo): Set signal signo on thread tid

- pthread\_cancel(tid): Cancel a thread (request that it terminate)

- pthread\_detach(tid): Make the thread detached (ie. unjoinable)

- pthread\_self(): Find out the calling threads tid

Here is an example of obtaining the thread priority of the calling thread:

int policy;

struct sched\_param param;

pthread\_getschedparam (0, &policy, &param); /\* Setting tid arg to 0 means get calling thread \*/

printf(“Thread %d’s priority is %d\n”, pthread\_self(), param.sched\_priority);

Waiting for Threads to Die And Finding Out Why (pthread\_join):

If a thread is “joinable” then you can wait for it to die. If a thread dies before the call to pthread\_join(), then pthread\_join() will return immediately and the variable that is passed as the second argument will contain the thread’s return value or the value passed to pthread\_exit(). Once a pthread\_join() is done, the information about the dead thread is gone. Ex:

pthread\_create(&tid, NULL, worker\_thread, NULL);

// Do stuff

pthread\_join(tid, &return\_status); //Wait for thread to die if it hasn’t already

Thread Termination and Cleanup (pthread\_exit):

Threads may self-terminate using pthread\_exit(). pthread\_exit() will handle some of the cleanup that we would otherwise have to do behind the scenes. pthread\_exit() will store the return status of the function in the argument variable, similar to pthread\_join().

Threads may also terminate other threads within the same process. This can be accomplished using pthread\_cancel() or pthread\_abort(). pthread\_cancel() will request that the thread die. This may be delayed, as the thread will try to do some cleanup and exit properly. The alternative is to use pthread\_abort() which will kill the sibling thread immediately. A third function, pthread\_kill() will send a signal of your choosing which can be handled by the recipient accordingly. This signal can be masked, ignored, or handled. Some Unix signals are pre-defined in signal.h.

Process Death:

If any thread that belongs to a parent process calls exit(), then the process dies, and any/all remaining threads are terminated. The first thread within a process is called the “main thread” because it is responsible for calling main().

Processes may also die if all children threads die. This is abnormal behaviour, and is not considered as “normal termination” of the process. As mentioned earlier, when a process dies, all process resources are returned to the OS.

Thread Synchronization:

As you are likely aware, most modern processors have multiple cores i.e., multiple physical cores that are essentially their own individual microprocessor. This gives programmers the opportunity to divide the work that is necessary for a program to run into smaller units called threads. The term thread is actually short for “thread of execution”. The term “synchronous” or “concurrent”, as you may also hear it, means that each sub-task, or thread, is truly running in parallel i.e. at the same time. Threads can be running simultaneously, doing separate tasks. This is in contrast to asynchronous programming, which has gained popularity since JavaScript developed its async/await protocol. In asynchronous programming, there is no concurrency; tasks do not run on multiple threads. The goal of async is to reduce the time that a thread remains in a blocked state i.e. is waiting for a resource. It implements function callbacks so that it can jump to a new location of the code and execute that while it waits for another task to complete, at which point, it can return to that location and continue execution. We will not be focusing on async here, since plain C does not really have such features.

There are many troubles with synchronous programming, most of which are quite difficult to debug. As I’ve explained, each process must have at least one thread, often referred to as the main thread. When a new thread is created, it receives a copy of the stack from its parent thread. This means that it is inconsequential if we mutate stack-allocated variables, since they are local to the scope of the function. Anytime a thread invokes a function, it pushes its own stack frame onto its own stack for that function call, thus, variables which are created within that stack frame are untouched by other threads. The trouble is when we either tamper with heap allocated variables via pointers (since two threads accessing the same pointer will be referencing the same memory in the heap) or when we touch global variables. The issue that arises is known as a race condition. If one thread (T1) tries to write to a buffer in memory, and T2 comes along and attempts to read from that same memory, T1 may very well overwrite the contents of the buffer at the same time that T2 is reading from it, resulting in a corrupted result. There are a few techniques to deal with race conditions, which we call synchonization techniques. The most fundamental and basic techniques / data structures are sometimes referred to as “synchronization primitives”. These primitives typically prevent certain portions of the code from executing synchronously. The protected regions of code are known as “critical sections”. Not all synchronization primitives operate in this way; for example, atomic operations do not create any sort of critical zone, but this is an exception to the rule.

Mutexes:

Mutexes (short for “mutual exclusion”) are the most fundamental synchronization primitive, from which, all other primitives stem. The premise is fairly simple, but we use some terms that may confuse new programmers. You can think of a mutex like an entity e.g. a person. We may have multiple mutexes (people) trying to enter a room (the critical section of code). The catch is that only one person can hold the key to that room at a time, and whenever a person enters the room, it is immediately locked so that no other person can come in. Once that person is finished occupying the room, they unlock the door, and return they key for someone else to access. In other words, we say that a mutex “acquires the lock”, meaning that the thread which owns that mutex gets to run the code in our critical section. Once finished, we say that the mutex “releases the lock”, giving a chance for other mutexes to obtain it.

Mutex API Calls:

In order to create a mutex, we can use the function

pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex, const pthread\_mutexattr\_t \*restrict attr).  
This function creates a mutex which will be owned by the thread that initialized it. In order to destroy a mutex (which should be done when it is no longer in use), we can use the function pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex) which will simply destroy the mutex. The pthread.h header also provides a macro (PTHREAD\_MUTEX\_INITIALIZER) which returns a pre-allocated mutex object for us to use.

Here’s a basic example that utilizes a mutex:

static pthread\_mutex\_t mutex;

init() {

pthread\_mutex\_init(&mutex, NULL); /\* NULL implies default attributes for mutex \*/

/\* do something... \*/

}

thread\_func() {

pthread\_mutex\_lock(&mutex); /\* Acquire the mutex/lock \*/

/\*\*\* Critical section \*\*\*/

pthread\_mutex\_unlock(&mutex); /\* Release the mutex/lock \*/

}

cleanup() {

pthread\_mutex\_destroy(&mutex);

}

In this example, each thread will presumably be invoking thread\_func(). There is a single mutex in this example, belonging to the main thread. Note that since the ownership of that mutex belongs to the main thread, the main thread is responsible for calling pthread\_mutex\_destroy() during cleanup. It is not permissible for one thread to destroy a mutex belonging to another thread. Anyhow, in this example, if two threads are invoking thread\_func() at the same time i.e. they are competing for a resource, then the thread that makes it there first will ultimately acquire the mutex first, leaving the other thread(s) to be blocked until the mutex is locked by the mutex holder.

Mutex Attribute:

Much like threads, we can give mutexes an attribute object using a few functions listed here:

- pthread\_mutexattr\_init(pthread\_mutexattr\_t \*attr)

- pthread\_mutexattr\_setpshared(pthread\_mutexattr\_t \*attr, int pshared)

-

For example, let’s say that we want to create a multithreaded application that uses shared memory, but we don’t want two threads to access the shared memory at the same time from two different functions. In this case, we want to set the attribute of the mutex such that this is not allowed.

pthread\_mutexattr\_t mutex\_attr;

pthread\_mutex\_t \*mutex;

pthread\_mutexattr\_init(&mutex\_attr, PTHREAD\_PROCESS\_SHARED);

mutex = (pthread\_mutex\_t \*mutex) shmem\_ptr;

pthread\_mutex\_init(mutex, &mutex\_attr);

Dead-lock:

Another issue with mutexes is deadlock. When locking multiple mutexes in multiple threads, this may result in a deadlock. Here is a theoretical sequence of events that might lead to a deadlock:

1. T1 locks M1

2. T2 locks M2

3. T1 attempts to acquire M2, which T2 owns

4. T2 attempts to acquire M1 which T1 owns

5. Deadlock occurs as both threads are blocked waiting for the other to release their mutex

This can also happen with more than two threads, and this can be very difficult to debug. The solution to this issue is to establish a locking heirarchy that specifies the order in which threads acquire mutexes. The way that this works is that threads with a mutex locked will execute the critical section of code with a boosted priority so that they don’t get preempted by other threads. In other words, we set the priority of the thread with a locked mutex such that no other threads that are locked may try to acquire their mutex.

Condvars:

Conditional variables are sort of similar to mutexes in function, but have some key differences. They are typically used when we want to wait on some state to change before performing some work. In other words, we want to block the thread until another thread changes some variables state. E.g.

volatile int state;

thread\_1() {

while(1) {

// Wait until state changes, then perform work

}

}

Mutex Function Calls:

In order to create and destroy condvars, similar to mutex, we have pthread\_cond\_init(pthread\_cond\_t \*restrict cond, const pthread\_condattr\_t \*restrict attr) and pthread\_cond\_destroy(pthread\_cond\_t \*cond). These behave pretty much the same as the functions for mutex initialization and destruction.

There are 3 functions that we can use on condvars (unrelated to the 3 for mutexes). The first is pthread\_cond\_wait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex). This function will essentially map a mutex to a condvar. This tells the mutex that it should wait to allow the thread to execute the critical section until a signal is delivered from the condvar. The second function is pthread\_cond\_signal(pthread\_cond\_t \*cond). This function will send a signal to any mutexes that are mapped to the condvar provided as the argument. Finally, pthread\_cond\_broadcast(pthread\_cond\_t \*cond) will do the same thing as pthread\_signal() but it will send the signal to all theads in the process.

Let’s consider a hypothetical example. Let’s say that a data provider threads receives some data from a client process and adds it to a queue. That data provider thread then notifies the hardware handling thread that there is data in the queue. Finally, the hardware handling thread wakes up, removes the data from the queue, and write it to the hardware. In order to do this, we require a mutex to make sure that the data and hardware threads don’t access the queue structure at the same time, and we also require a mechanism to notify the hardware thread to wake up (ie. a condvar).

// Hardware Handling Thread’s Code:

while (1) {

pthread\_mutex\_lock(&mutex);

while (!data\_ready)

pthread\_cond\_wait(&cond, &mutex);

/\* Get and decouple data from the queue \*/

while((data = get\_data\_and\_remove\_from\_queue()) != NULL) {

pthread\_mutex\_unlock(&mutex);

write\_to\_hardware(data);

free(data);

pthread\_mutex\_lock(&mutex);

}

data\_ready = 0; //Reset flag

pthread\_mutex\_unlock(&mutex);

}

//Data Provider’s Code:

pthread\_mutex\_lock(&mutex);

add\_to\_queue(buf);

data\_ready = 1;

pthread\_cond\_signal(&cond);

pthread\_mutex\_unlock(&mutex);

We require the while(!data\_ready) loop to keep the thread in a waiting state. If it were not waiting, then the signal from pthread\_cond\_signal() would be lost. Therefore, not only do we need to send the signal, but we also need to set a flag to keep the thread waiting.

Semaphores:

Read/Write Locks:

Signaling vs. Broadcasting:

We understand that signalling is used to notify a thread that it can continue executing after some state change, but when would we want to do this for many threads at the same time using broadcasting? Let’s assume that threads 1, 2, and 3 (all at the same priority) are waiting for a change via pthread\_cond\_wait(). If thread 4 signals using pthread\_cond\_signal() then the longest waiting thread is informed of the change, and it tries to acquire the mutex, but the other threads never run. In the case of pthread\_cond\_broadcast(), all 3 threads receive the signal at the same time, and can therefore execute in order of their scheduling algorithms (pressumably RR ie. longest waiting to least longest waiting).

Producer/Consumer Example:

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;

volatile int state = 0;

volatile int product = 0;

void \*consume (void \*arg) {

while(1) {

pthread\_mutex\_lock(&mutex);

while(state == 0)

pthread\_cond\_wait(&cond, &mutex);

printf(“Consumed %d\n”, product);

state = 0;

pthread\_cond\_signal(&cond);

pthread\_mutex\_unlock(&mutex);

do\_consumer\_work();

}

return 0;

}

void \*produce (void \*arg) {

while(1) {

pthread\_mutex\_lock(&mutex);

while (state == 1)

pthread\_cond\_wait(&cond, &mutex);

printf(“Produced %d\n”, product++);

state = 1;

pthread\_cond\_signal(&cond);

pthread\_mutex\_unlock(&mutex);

do\_producer\_work();

}

return 0;

}

int main(void) {

pthread\_create(NULL, NULL, &produce, NULL); //Anonymous thread (no handle)

consume(NULL);

return EXIT\_SUCCESS;

}

Atomic Operations:

An atomic operation is one that is guaranteed to execute within a single instruction. We use the term atomic since atoms are “indivisible”, as are atomic operations. Atomics are generally more simplistic e.g. arithmetic or logical operations, since they can utilize advanced features of the CPU that allow them to execute in a single clock cycle. Due to this, more advanced operations such as copying the contents of one buffer to another are either much less used, or non-existent. Atomic operations depend upon whether or not the hardware supports them, so attempting to perform them on older hardware is not a very wise idea. Another reason that atomics are not used very often is because they don’t belong to the C standard library (they are compiler extensions). Clang and GCC both support them. If you require information about which functions are available, you must either read the documentation for your compiler of choice, or look through the header file. On Linux, these headers will not belong under /usr/include, but rather, somewhere like /usr/lib/gcc/.../include.

Most standard types have atomic variants. For example, atomic\_bool, atomic\_char, atomic\_short, atomic\_int, atomic\_long, etc. There are also some other weird types, such as atomic\_uint\_fast64\_t, atomic\_ptrdiff\_t, atomic\_intmax\_t, etc. You’ll have to look at the header file or do your research to find out what these are.

Here are a few atomic operations used for arithmetic and logical operations:

- atomic\_fetch\_add(volatile A \*obj, M amount)

- atomic\_fetch\_sub(volatile A \*obj, M amount)

- atomic\_flag\_clear(volatile atomic\_flag \*obj)

- atomic\_flag\_test\_and\_set(volatile atomic\_flag \*obj)

- atomic\_fetch\_xor(volatile A \*obj, M arg)

System:

Since Linux is written in C, most of the Linux utilities are just programs that call some C library API call. Therefore, many of the Linux utilities are also functions that we can call in C. And because of this tight association between the two, it is often-times useful for us to be able to send some command to the terminal from our program. Enter system(). This function allows us to enter commands into the command line from our C program. This function is part of stdlib.h and takes in a single parameter: A string with the command that you’d like to run. In short, this function returns the exit status of the command, but there is actually a lot more edge cases to this one, so I recommend that you consult the man pages for more information about the return value. The most important thing to know for error handling is that system() returns -1 if it fails to execute the command at all. In my opinion, it is generally better to use the built in libc API calls for executing tasks wherever possible, but system() is very convenient for a lot of system-related tasks such as creating logs or removing files. Use it at your own discretion.

Signals:

If you’ve ever used a Unix-like or Unix-based OS (OS X, Linux, BSD, etc.) then you’ve probably used signals even if you never realized it. When you wish to terminate a program in Linux, you typically press Ctrl + C to terminate the program. Under the hood, Linux is actually sending a signal called SIGINT to the process to request that it terminate. Ctrl + z stops the program by sending SIGTSTP (signal tty stop). You may think Ctrl + d also sends a signal, but actually it sends EOF to the program, which is not a signal. It is possible to intercept these signals in C by using a function called signal(). It’s really quite simple, all you have to do is call the function at some point before the signal is called. The first argument is the signal that you want to intercept, and the second parameter is a function pointer to the handler function that you provide. The only signal that can’t be handled is SIGKILL which always terminates the program no matter what. For example, if I wanted to intercept SIGINT, I could do something like the following:

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

void handle\_sigint(int);

void handle\_sigint(int sig) {

printf("\nNot exiting >:)\n");

}

int main(int argc, char \*argv[]) {

signal(SIGINT, &handle\_sigint);

while(1) {

printf("Infinite loop..\n");

sleep(1);

}

/\* Unreachable code \*/

return 0;

}

Since Ctrl + c is not sufficient to kill this program, you will need to send SIGTSTP using Ctrl + z on Linux. Stopping a process freezes its state, but does not kill the program, so in order to free the process resources, run pgrep <name\_of\_program> which will return the PID of that process, followed by kill -9 <PID> which will send SIGKILL to the process (or combine them into one command e.g. kill -9 $(pgrep <name\_of\_program>).

Clocks:

Your system uses different kinds of clocks to calculate elapsed time depending on the circumstance. There are 2 types of clock:

**CLOCK\_MONOTONIC:** A monotonic clock is a real-time clock defined by POSIX. It cannot be set via clock\_settime() and cannot have negative clock jumps (i.e. it is purely an incremental timer, not a decremental timer).

**CLOCK\_REALTIME:** This type of clock is based on the system clock-time. This sort of clock is relative to the current time.

sleep(): unsigned int sleep(unsigned in seconds)

Any time that we need to create some sort of downtime in our process, we can use sleep() to pause the execution of a thread. sleep() takes in the number of seconds that you would like to pause for and suspends the calling thread for that amount of time. Other threads are still free to execute while sleep() blocks its calling thread.

nanosleep(): int nanosleep(const struct timespec \*rqtp, struct timespec \*rmtp);

If the granularity of milliseconds is not enough, we can also use nanosleep() which is defined in time.h. This function takes two timespec structs: rqtp and rmtp.The timspec struct has two important members: time\_t tv\_sec, and long tv\_nsec which are used for seconds and nanoseconds respectively. This function causes the calling thread to sleep for the requested number of seconds + nanoseconds as defined by the rqtp timespec struct. If nanosleep returns due to being interrupted by a signal, it returns -1 and sets errno appropriately. The argument for rmtp can either be a timespec struct or NULL. If rmtp is not NULL, it stores the remaining time (i.e. the expiry time - the time elapsed). It is recommended that you use nanosleep() over usleep() if possible since usleep() has been somewhat deprecated.

strftime(): size\_t strftime(char \*restrict s, size\_t max,

const char \*restrict format,

const struct tm \*restrict tm);

size\_t strftime\_l(char \*restrict s, size\_t max,

const char \*restrict format,

const struct tm \*restrict tm,

locale\_t locale);

If we require the current date-time string, we can use the strftime and strftime\_l functions. These two functions convert the time and date to a string using a specified format.

The strftime function returns a string representing the date time in the format specified by format, which is a format string, similar to that of printf. The maxsize parameter is the maximum number of bytes to be copied to the string s. The tm struct contains members such as tm\_sec, tm\_min, tm\_hour, tm\_mday, tm\_mon, tm\_year, tm\_wday, tm\_yday, and tm\_isdst (more info in the time.h man page).

The format string specified by format has a few special characters. The character 0 will be interpreted as a padding character (whitespace). The plus sign ‘+’ specifies the maximum number of padding bytes, and the minus sign ‘-’ specifies the minimum. For example, %+4Y would indicate that the year can be at most 4 bytes long. There are predefined format specifiers in the strftime man pages which are easier to use than creating your own.

The strftime\_l function generally works the same as strftime, but adds an additional parameter (locale). The locale\_t struct (defined in locale.h) contains things like the encoding scheme, language tag (e.g. en\_US), etc. Within the format specifier for strftime\_l, you can use the E or O characters to tell the function that the following format specifier should be interpreted according to the locale, rather than the default interpretation. For instance, normally %c would be equivallent to %a %b %e %T %Y, but %Ec means use the local’s date and time representation.

POSIX Events:

POSIX uses events as a mechanism for notifying other processes or threads of some action that occurred. An example of this are POSIX timers, which expire, then deliver an event via signal to notify the listeners. Events can either send a signal, or associate a callback to be triggered when the action occurs.

The sigevent struct:

A sigevent is used to describe the *method* of handling the POSIX event (either sending a signal or spawning a new thread to run a callback function). The actual method is determined by setting the sigev\_notify field to be one of: SIGEV\_NONE (do nothing), SIGEV\_SIGNAL (send a signal), or SIGEV\_THREAD (spawn a new thread to execute the callback function). In either case, we’re able to relay some data to the recipient via the sigval union. This union can either be some integer data, or some raw buffer of data which is referenced by void pointer. All of the information for the sigval union, sigevent struct, signal numbers, and more, can be found by reading the man page for signal.h. The defintions for sigval and sigevent are shown below:

union sigval { /\* Data passed with notification \*/

int sival\_int; /\* Integer value \*/

void \*sival\_ptr; /\* Pointer value \*/

};

struct sigevent {

int sigev\_notify; /\* Notification method \*/

int sigev\_signo; /\* Notification signal \*/

union sigval sigev\_value; /\* Data passed with notification \*/

/\* Function used for thread notification \*/

void (\*sigev\_notify\_function)(union sigval);

/\* Attributes for notification thread \*/

void \*sigev\_notify\_attributes;

/\* ID of thread to signal (Linux-specific) \*/

pid\_t sigev\_notify\_thread\_id;

};

sigaction(): int sigaction(int sig,

const struct sigaction \*restrict act,

const struct sigaction \*restrict oact);

The sigaction() API associates any of the signals defined in signal.h with some callback function. Unlike the sigevent struct, which defined the method of delivering an event, sigaction() only specifies what to do when a signal is received. Signals can be received for all sorts of reasons – not necessarily due to an event being transmitted. Being able to modify the default behaviour for these signals can be quite useful. In order to associate a signal with an action, we use a sigaction struct, which has the following definition:

struct sigaction {

void (\*sa\_handler)(int signal); /\* Handler callback \*/

sigset\_t sa\_mask; /\* Signal filter/mask \*/

int sa\_flags; /\* Additional flags \*/

void (\*sa\_sigaction)(int signal, signinfo\_t \*info, void \*arg); /\* Alternative callback \*/

}

The sigaction struct has a function pointer called sa\_handler which points to the callback function that should be invoked when the signal is received. We will discuss sa\_mask in the next section, so bear with me for a moment. The sa\_flags field can accept a number of additional flags that change the behaviour of how the signal is caught/handled. For brevity I’ll let you search these in the sigaction man page. One such flag, however, SA\_SIGINFO, if set will cause the sa\_sigaction callback function to be used instead of sa\_handler. The sa\_sigaction callback provides additional info via the siginfo\_t struct and can accept additional data if required.

In order to actually associate a signal with a sigaction struct, we need to use the sigaction() API (yes, it’s a bit confusing that the actual function and struct are named the same). The sigaction() function accepts a signal number as its first parameter, which is the signal that we are associating the action with. It then accepts two sigaction structs. If act is not NULL, then it is used as the mapping for the signal specified. If oact is not NULL, then the action previously mapped to the signal is stored in it.

Filtering Signals with sigset\_t:

The sigaction struct has a field called sa\_mask, which is of type sigset\_t. This field is used to filter or mask the types of signals that we either want to act upon or ignore. The sigset\_t alias is really just an integer that we perform bitwise operations on, much like how a normal mask works. A couple helper functions make setting the sigset\_t object easier. These include sigemptyset(), which clears the set to 0, sigaddset() to add a signal to the set, sigdelset() to remove a signal from the set, and sigfillset(), to indicate that we want to whitelist all available signals.

Timers:

Timers are slightly different than calls to sleep or nanosleep. A timer is created on a per-process basis. Multiple threads may share the same timer, whereas sleep or nanosleep cause a single thread to remain inactive for some duration of time. Timers use a specified clock (clock\_id) as the timing basis (this is how we select what type of clock the timer shall use e.g. CLOCK\_MONOTONIC or CLOCK\_REALTIME). There are a few structs for timers that we should be familiar with. These are timerspec, itimerspec, and tm. Here is the declaration of each:

**timespec:**

struct timespec {

time\_t tv\_nsec; /\* Seconds \*/

long tv\_nsec; /\* Nanoseconds \*/

};

Note here that time\_t is a typedef for the number of seconds since the epoch (Jan 1st 1970).

**itimerspec:**

struct itimerspec {

struct timespec it\_interval; /\* Timer interval \*/

struct timespec it\_value; /\* Initial expiration \*/

};

**tm:**

struct tm {

int tm\_sec /\* Seconds [0-60] (1 leap second) \*/

int tm\_min; /\* Minutes [0-59] \*/

int tm\_hour; /\* Hours [0-23] \*/

int tm\_mday; /\* Day [1-31] \*/

int tm\_mon; /\* Month [0-11] \*/

int tm\_wday; /\* Days of week [0-6] \*/

int tm\_yday; /\* Days in year [0-365] \*/

int tm\_isdst; /\* DST [-1/0/1] \*/

};

timer\_create: int timer\_create(clock\_id clockid,

struct sigevent \*restrict evp,

timer\_t \*restrict timerid)

The timer\_create() API call is how we create a new POSIX timer. The first argument is the ID of the clock type that we want to use. This can be one of CLOCK\_MONOTONIC or CLOCK\_REALTIME. The latter option represents the system’s best estimate at the current wall-time clock. It is subject to change as it gets updated via an NTP server to keep up with the actual time. A monotonic clock, on the other hand, begins ticking when the system boots, and keeps a more accurate pacing, but does not represent the wall-time clock accurately. Which you use will depend upon what you’re attempting to do. Do you need the timer to expire at an actual real world time, or at some time relative to now? A third Linux specific option, CLOCK\_BOOTTIME works like CLOCK\_MONOTONIC, but operates even during suspend mode. The timer\_t alias (not to be confused with time\_t) is just an integer to which the created timer will be bound. This id can be referenced for further operations on the timer such as when destroying it with timer\_delete().

timer\_settime(): int timer\_settime(timer\_t timerid, int flags,

const struct itimerspec \*restrict value,

struct itimerspec \*restrict ovalue);

Hot-Reloading:

If you read through my C programming notes, you may have reached the portion where I discussed the dlfcn.h library, which allowed us to open or close DLLs at runtime and load symbols from within said DLLs. I mentioned that it was possible to perform hot reloading using these functions, but never provided a practical example since I felt it would add unnecessary padding to an already lengthy document. Here, I’ll provide an overview of the steps by which we can achieve hot reloading, followed by some sample code.

In order for hot-reloading to work, both the state and behavior must be co-located within the DLL itself. This is because the entire premise depends upon our ability to make edits to the code of the DLL and then reload the DLL from within our application. On a more technical level, what we need are a context object, such as a struct, as well as function pointers in our main application which point to some APIs exposed from the DLL. Our application calls into the hooks at runtime, which invoke the behavior present in the DLL. Doing things this way allows us to dynamically update both state and behaviour. We use some mechanism to reload the DLL once the edits have been made. The example I’m about to show you was heavily inspired by the Youtube programmer Tsoding, who has some excellent content on low level programming. The example code I’m providing is split into 3 files. shared.c and shared.h makeup the libshared.so DLL, and main.c makes up our main application.

/\* shared.h \*/

#ifndef SHARED\_H

#define SHARED\_H

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

typedef struct {

int count;

} State\_t;

static State\_t \*state = NULL;

void init();

void update();

void \*pre\_reload();

void post\_reload(void \*new\_state);

#endif /\* SHARED\_H \*/

/\* shared.c \*/

#include "shared.h"

void init() {

state = calloc(1, sizeof(\*state));

assert(state);

}

void \*pre\_reload() {

return state;

}

void post\_reload(void \*new\_state) {

state = (State\_t\*)new\_state;

}

void update() {

state->count = 0;

printf("Count is: %d\n", state->count);

}

/\* main.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <dlfcn.h>

#define DLL\_NAME "libshared.so"

#define ERROR() \

fprintf(stderr, "%s\n", dlerror()); \

exit(EXIT\_FAILURE)

static void \*dll\_hndl = NULL;

void (\*init)() = NULL;

void \*(\*pre\_reload)() = NULL;

void (\*post\_reload)(void\*) = NULL;

void (\*update)() = NULL;

void reload() {

int status;

/\* Close DLL if open \*/

if (dll\_hndl != NULL) {

status = dlclose(dll\_hndl);

if (status != 0) {

ERROR();

}

}

/\* Open DLL \*/

dll\_hndl = dlopen(DLL\_NAME, RTLD\_NOW);

if (dll\_hndl == NULL) {

ERROR();

}

/\* Assign function pointers \*/

init = dlsym(dll\_hndl, "init");

if (init == NULL) {

ERROR();

}

pre\_reload = dlsym(dll\_hndl, "pre\_reload");

if (pre\_reload == NULL) {

ERROR();

}

post\_reload = dlsym(dll\_hndl, "post\_reload");

if (post\_reload == NULL) {

ERROR();

}

update = dlsym(dll\_hndl, "update");

if (update == NULL) {

ERROR();

}

}

int main(void) {

char c = '\0';

reload();

init();

while (c != 'q') {

c = getchar();

if (c != '\n') {

while (getchar() != '\n') ;;

}

if (c == 'r') {

void \*state = pre\_reload();

reload();

post\_reload(state);

printf("Reloaded DLL\n");

} else {

update();

}

}

if (dll\_hndl != NULL) {

dlclose(dll\_hndl);

}

return 0;

}

In the example provided, we create a context object called state which is defined in shared.h. The DLL contains the functions init(), update(), pre\_reload(), and post\_reload(). The init() function is responsible for initializing the context object. The pre\_reload() function returns a pointer to the context object. The post\_reload() function sets the state to whatever data is passed into it. Together, the pre\_reload() and post\_reload() functions allow us to make a backup copy in the main program, then close and reopen the DLL, and then restore the state back to what it was before we closed it. Finally, the update() function contains our modular behaviour/logic (in this case, we just print out whatever count is).

In the main program, we create function pointers for each of the functions defined in our DLL. When the program starts, we call reload(). This function will check to see if we have a valid handle to the DLL, which we will if it’s currently open. If so, we first close the DLL. In either case, we then open the DLL and retrieve a new handle. We use the dlsym() function to retrieve the addresses of each function loaded in the DLL’s memory mapping and map them to our function pointers. In our program’s runtime loop we call update() each time the user presses enter. At first this will print out 0 since this is the initial value of count in the state object. We can edit the shared.c file while main is running and change the assignment of state->count in the update() function from 0 to something else such as 1. We recompile the DLL and then enter the character r into our main program which reloads the DLL. At this point update will start printing out our new value of 1!

Note that in order to compile this example, we first compile the shared.c/h translation unit like so: gcc -fPIC -shared shared.c -o libshared.so. We compile main.c as normal: gcc main.c -o main. Also make sure that you use either LD\_LIBRARY\_PATH or -Wl,rpath to contain the current directory (./) so that main.c knows where to look when invoking dlopen().

If you want to go even further with this example, rather than using user input as the conditional for reloading the DLL, we can actually reload it when shared.c is updated. In order to do this, we take advantage of a couple things we’ve learned so far. Here is the updated main.c:

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <dlfcn.h>

#include <fcntl.h>

#include <unistd.h>

#include <pthread.h>

#include <sys/stat.h>

#include <sys/wait.h>

#define DLL\_NAME "libshared.so"

#define ERROR() \

fprintf(stderr, "%s\n", dlerror()); \

exit(EXIT\_FAILURE)

static void \*dll\_hndl = NULL;

static pthread\_mutex\_t mutex;

static pthread\_cond\_t cond;

void (\*init)() = NULL;

void \*(\*pre\_reload)() = NULL;

void (\*post\_reload)(void\*) = NULL;

void (\*update)() = NULL;

void reload() {

int status;

/\* Close DLL if open \*/

if (dll\_hndl != NULL) {

status = dlclose(dll\_hndl);

if (status != 0) {

ERROR();

}

}

/\* Open DLL \*/

dll\_hndl = dlopen(DLL\_NAME, RTLD\_NOW);

if (dll\_hndl == NULL) {

ERROR();

}

/\* Assign function pointers \*/

init = dlsym(dll\_hndl, "init");

if (init == NULL) {

ERROR();

}

pre\_reload = dlsym(dll\_hndl, "pre\_reload");

if (pre\_reload == NULL) {

ERROR();

}

post\_reload = dlsym(dll\_hndl, "post\_reload");

if (post\_reload == NULL) {

ERROR();

}

update = dlsym(dll\_hndl, "update");

if (update == NULL) {

ERROR();

}

}

long long time\_to\_nano(struct timespec ts) {

return (ts.tv\_sec \* 1000L \* 1000L \* 1000L) + ts.tv\_nsec;

}

void \*file\_listener(void \*args) {

int fd = -1;

long long nanos = 0L;

struct stat s = { 0 };

struct timespec last\_access = { 0 };

while (true) {

fd = open("shared.c", O\_RDONLY);

fstat(fd, &s);

last\_access = s.st\_mtim;

close(fd);

if (time\_to\_nano(last\_access) > nanos) {

printf("shared.c was updated\n");

nanos = time\_to\_nano(last\_access);

system("gcc -fPIC -shared shared.c -o libshared.so");

pthread\_mutex\_lock(&mutex);

pthread\_cond\_broadcast(&cond);

pthread\_mutex\_unlock(&mutex);

}

}

return NULL;

}

int main(void) {

pthread\_t t;

pthread\_mutex\_init(&mutex, NULL);

pthread\_cond\_init(&cond, NULL);

pthread\_create(&t, NULL, file\_listener, NULL);

reload();

init();

while (true) {

/\* Block until file update \*/

pthread\_mutex\_lock(&mutex);

pthread\_cond\_wait(&cond, &mutex);

pthread\_mutex\_unlock(&mutex);

void \*state = pre\_reload();

reload();

post\_reload(state);

printf("Reloaded DLL\n");

update();

}

pthread\_join(t, NULL);

pthread\_mutex\_destroy(&mutex);

pthread\_cond\_destroy(&cond);

if (dll\_hndl != NULL) {

dlclose(dll\_hndl);

}

return 0;

}

In this updated version we now use a conditional variable to signal when the file is modified. We do this by using the stat struct on the shared.c file and checking the st\_mtim timespec struct which tells us the last data modification timestamp. This is accomplished on a separate thread dedicated to listening for file updates. The time\_to\_nano() function converts this timespec struct into nano seconds. If the current timestamp in nanoseconds is greater than the previous time, this means the file has been updated sometime in between. We do something a bit hacky, which is to use the system() API to recompile the file. We then broadcast the update to all threads blocked on the condvar (in our case only the main thread is blocked). Once unblocked, we reload the DLL and output the count value of the state object.

Locale:

Socket Programming:

Unix Domain Sockets vs. Internet Sockets vs. Berkley Sockets:

UDS a.k.a. IPC sockets are a data communications endpoint for exchanging data between processes executing on the same operating system. These sockets are also sometimes referred to by their address family (AF\_UNIX). Valid socket types in the UNIX domain are SOCK\_STREAM (TCP), SOCK\_DGRAM (UDP), or SOCK\_SEQPACKET (SCTP). This is in contrast to Internet sockets, which UDSs closely mirror in terms of their API, but that allow for network communications between devices across the internet. Finally, we have Berkley sockets, which are named after the Berkley Software Distribution (BSD), who also created the BSD operating system. Berkley sockets are also sometimes called BSD sockets or POSIX sockets, as they were integrated into the POSIX standard early on. Berkley sockets are yet another API which builds upon both UDSs and Internet sockets. With Berkley sockets, there are API calls for both IPC and network communications. The API calls for Berkley sockets are included in the <sys/socket.h> header file. We also require the <arpa/inet.h> header file, which includes additional APIs for internet sockets. If you’re unaware, ARPANET (Advanced Research Projects Agency Network) was the forerunner of the internet. It was initially used by the U.S. Defense Department to link computers at Pentagon-funded research institutions over telephone lines, and was the first wide-area packet switched network to use the TCP/IP protocol suite.

Writing a Client-Side Program:

The following code is credited entirely to Jacob Sorber on YouTube because I think that his example code was excellent for explaining the core idea of a client-side socket program. I highly recommend that you watch the video here: <https://www.youtube.com/watch?v=bdIiTxtMaKA&list=PL9IEJIKnBJjH_zM5LnovnoaKlXML5qh17>. I’ve taken the liberty of copying his code so that you may copy it into your IDE or vim or emacs or whatever you use. Here it is:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <limits.h>

#include <stdarg.h>

#include <errno.h>

#include <arpa/inet.h> /\* Internet socket APIs \*/

#include <sys/socket.h> /\* Berkley socket APIs \*/

#define SERVER\_PORT 80 /\* Standard HTTP port \*/

#define CRLF "\r\n"

typedef struct sockaddr sockaddr\_t;

typedef struct sockaddr\_in sockaddr\_in\_t;

int main (int argc, char \*\*argv) {

char sendline[LINE\_MAX];

char recvline[LINE\_MAX];

sockaddr\_in\_t servaddr = { 0 };

int sock\_fd, nb\_sent, nb\_read;

if (argc != 2) {

fprintf(stderr, "Usage: %s <server address>\n", argv[0]);

exit(EXIT\_FAILURE);

}

/\*

Create a socket:

- AF\_INET means this is an internet socket (as opposed to AF\_UNIX)

- SOCK\_STREAM specifies that this is a TCP socket

\*/

if ((sock\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) == -1) {

fprintf(stderr, "Error while creating the socket\n");

exit(EXIT\_FAILURE);

}

/\* Create the network address \*/

servaddr.sin\_family = AF\_INET; /\* Match the socket \*/

servaddr.sin\_port = htons(SERVER\_PORT); /\* Set the port number \*/

/\* Perform input validation on argv[1] to ensure inet address is in proper format \*/

if (inet\_pton(AF\_INET, argv[1], &servaddr.sin\_addr) <= 0) {

fprintf(stderr, "The inet address %s is not formatted correctly\n", argv[1]);

exit(EXIT\_FAILURE);

}

/\* Attempt a connection \*/

if (connect(sock\_fd, (sockaddr\_t \*)(&servaddr), sizeof(servaddr)) < 0) {

fprintf(stderr, "Connection failed\n");

exit(EXIT\_FAILURE);

}

/\*

Prepare HTTP request:

- GET = GET request

- / = Root of the server

- HTTP/1.1 = Protocol is HTTP 1.1

\*/

sprintf(sendline, "GET / HTTP/1.1" CRLF CRLF);

nb\_sent = strlen(sendline);

/\* Write our request to the socket \*/

if (write(sock\_fd, sendline, nb\_sent) != nb\_sent) {

fprintf(stderr, "Write error\n");

}

memset(recvline, 0, sizeof(recvline));

/\* Read response from the socket \*/

while ((nb\_read = read(sock\_fd, recvline, (sizeof(recvline) - 1))) > 0) {

printf("Response: %s\n", recvline);

}

if (nb\_read == -1) {

fprintf(stderr, "Read error\n");

exit(EXIT\_FAILURE);

}

return EXIT\_SUCCESS;

}

Now again, this looks super scary in Microsoft Word because I haven’t changed the font size, but in an editor, this is much more manageable. We define a few macros at the top, most notably SERVER\_PORT which is set to 80. 80 is the standard port number for HTTP requests, and we will be trying to scrape the contents of the root folder of whichever website we request with this protocol. Next, we declare a basic function called err\_hdlr which simply takes in a format string in similar fashion to printf() or scanf(), followed by the optional variable arguments corresponding to any format specifiers in the format string. If errno was set by any of the functions that may have failed prior, perror() will print out an appropriate string that corresponds to its value. In main() we first check if the argument count is set correctly, and if not print a usage string. In the next if statement, we create a file descriptor (which remember, is just an int) which is returned from invoking the socket() function. We pass in AF\_INET to specify that the socket will be communicating over the internet. If we wanted to make a Unix-Domain Socket instead, we’d use AF\_UNIX here. The other parameter is set to SOCK\_STREAM, which makes it a TCP/IP socket rather than a UDP socket. In order to make it a UDP socket, you’d pass in SOCK\_DGRAM. The final parameter is the protocol number. 0 means use the default protocol for the requested socket type. The sin\_family member just matches what we passed to socket(). sin\_port takes in our port number, 80, but we must take care to convert it into network-byte order. In order to do this, we pass SERVER\_PORT to a function called htons(), which stands for host to network (short int). There are other variants of hton\*(), such as htonl(), htohl() and htohs(). htons() will essentially just convert the number 80 to the correct endianness for us, since almost all computers speak in little-endian, but the internet speaks in big-endian (a.k.a. network-byte order). Next, we call inet\_pton() and pass in the ip address string that the user entered from the command line so that it can be validated i.e., ensure it’s in the proper format and is a valid IP. This gets put into the sin\_addr member of the servaddr struct. Next, we attempt to connect to the actual server using the struct we setup and the socket (file descriptor) that we’ve established. Next, we construct a string to send to the server. This string is an HTTP GET request. We request the root of the webpage that we would like with the ‘/’ character, and we specify that it’s over HTTP 1.1. It is critical that the two return carriages and newline characters (\r\n\r\n) are placed at the end. This is the format that the web expects, and the return carriages are necessary since the server may be running on Windows (if it was a Linux server, we wouldn’t need them). Finally, we continue to invoke read() on the socket to receive the response until it has no more data to write.

The response that you get will depend upon the website that you’re attempting to access. In some cases, you may receive an HTTP 400 response, indicating a “bad request”. In Jacob’s video, he uses Google’s IP address as the example IP and it returns the HTTP headers, as well as the minified CSS + JS code for the web page.

Writing a Server-Side Application: