Section 2: Threads vs. Processes, File APIs

CS 162

June 29, 2020

Contents

1 Vocabulary

- **fork** - A C function that calls the fork syscall that creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process (except for a few details, read more in the man page). Both the newly created process and the parent process return from the call to fork. On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created.

- **wait** - A class of C functions that call syscalls that are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal.

- **pipe** - A system call that can be used for interprocess communication.

  More specifically, the `pipe()` syscall creates two file descriptors, which the process can `write()` to and `read()` from. Since these file descriptors are preserved across `fork()` calls, they can be used to implement inter-process communication.

  ```
  /* On error, pipe() returns -1. On success, it returns 0
   * and populates the given array with two file descriptors:
   *  - fildes[0] will be used to read from the data queue.
   *  - fildes[1] will be used to write to the data queue.
   *
   * Note that whether you can write to fildes[0] or read from
   * fildes[1] is undefined. */
  int pipe(int fildes[2]);
  ```

- **exit code** - The exit status or return code of a process is a 1 byte number passed from a child process (or callee) to a parent process (or caller) when it has finished executing a specific procedure or delegated task

- **exec** - The exec() family of functions replaces the current process image with a new process image. The initial argument for these functions is the name of a file that is to be executed.

- **pthreads** - A POSIX-compliant (standard specified by IEEE) implementation of threads. A `pthread_t` is usually just an alias for "`unsigned long int`".

- **pthread_create** - Creates and immediately starts a child thread running in the same address space of the thread that spawned it. The child executes starting from the function specified. Internally, this is implemented by calling the clone syscall.

  ```
  /* On success, pthread_create() returns 0; on error, it returns an error
   * number, and the contents of *thread are undefined. */
  int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                     void *(*start_routine) (void *), void *arg);
  ```

- **pthread_join** - Waits for a specific thread to terminate, similar to `waitpid(3)`.

```
/* On success, pthread_join() returns 0; on error, it returns an error number. */
int pthread_join(pthread_t thread, void **retval);
```

- **pthread_yield** - Equivalent to thread_yield() in Pintos. Causes the calling thread to vacate the CPU and go back into the ready queue without blocking. The calling thread is able to be scheduled again immediately. This is not the same as an interrupt and will succeed in Pintos even if interrupts are disabled.

```
/* On success, pthread_yield() returns 0; on error, it returns an error number. */
int pthread_yield(void);
```

- **critical section** - A section of code that accesses a shared resource and must not be concurrently run by more than a single thread.
- **race condition** - A situation whose outcome is dependent on the sequence of execution of multiple threads running simultaneously.
- **lock** - Synchronization primitives that provide mutual exclusion. Threads may acquire or release a lock. Only one thread may hold a lock at a time. If a thread attempts to acquire a lock that is held by some other thread, it will block at that line of code until the lock is released and it successfully acquires it. Implementations can vary.
- **semaphore** - Synchronization primitives that are used to control access to a shared variable in a more general way than locks. A semaphore is simply an integer with restrictions on how it can be modified:
  - When a semaphore is initialized, the integer is set to a specified starting value.
  - A thread can call `down()` (also known as **P**) to attempt to decrement the integer. If the integer is zero, the thread will block until it is positive, and then unblock and decrement the integer.
  - A thread can call `up()` (also known as **V**) to increment the integer, which will always succeed.

  Unlike locks, semaphores have no concept of ”ownership”, and any thread can call `down()` or `up()` on any semaphore at any time.
- **system call** - In computing, a system call is how a program requests a service from an operating system's kernel. This may include hardware-related services, creation and execution of new processes, and communication with integral kernel services such as process scheduling.
- **file descriptors** - File descriptors are an index into a file-descriptor table stored by the kernel. The kernel creates a file-descriptor in response to an open call and associates the file-descriptor with some abstraction of an underlying file-like object; be that an actual hardware device, or a file-system or something else entirely. Using file descriptors, a process's read or write calls are routed to the correct place by the kernel. When your program starts you have 3 file descriptors.

File Descriptor	File
0	stdin
1	stdout
2	stderr

- **int open(const char *path, int flags)** - open is a system call that is used to open a new file and obtain its file descriptor. Initially the offset is 0.

- **size_t read(int fd, void *buf, size_t count)** - read is a system call used to read `count` bytes of data into a buffer starting from the file offset. The file offset is incremented by the number of bytes read.
- **size_t write(int fd, const void *buf, size_t count)** - write is a system call that is used to write up to `count` bytes of data from a buffer to the file offset position. The file offset is incremented by the number of bytes written.
- **size_t lseek(int fd, off_t offset, int whence)** - lseek is a system call that allows you to move the offset of a file. There are three options for whence
  - SEEK_SET - The offset is set to `offset`.
  - SEEK_CUR - The offset is set to current_offset + `offset`
  - SEEK_END - The offset is set to the size of the file + `offset`
- **int dup(int oldfd)** - creates an alias for the provided file descriptor and returns the new fd value. dup always uses the smallest available file descriptor. Thus, if we called dup first thing in our program, it would use file descriptor 3 (0, 1, and 2 are already signed to stdin, stdout, stderr). The old and new file descriptors refer to the same open file description and may be used interchangeably.
- **int dup2(int oldfd, int newfd)** - dup2 is a system call similar to dup. It duplicates the `oldfd` file descriptor, this time using `newfd` instead of the lowest available number. If newfd was open, it closed before being reused. This becomes very useful when attempting to redirect output, as it automatically takes care of closing the file descriptor, performing the redirection in one elegant command. For example, if you wanted to redirect standard output to a file, then you woul
- **signals** - A signal is a software interrupt, a way to communicate information to a process about the state of other processes, the operating system, and the hardware. A signal is an interrupt in the sense that it can change the flow of the program —when a signal is delivered to a process, the process will stop what its doing, either handle or ignore the signal, or in some cases terminate, depending on the signal.
- **int signal(int signum, void (*handler)(int))** - signal() is the primary system call for signal handling, which given a signal and function, will execute the function whenever the signal is delivered. This function is called the signal handler because it handles the signal.
- **SIG_IGN, SIG_DFL** Usually the second argument to signal takes a user defined handler for the signal. However, if you'd like your process to drop the signal you can use SIG_IGN. If you'd like your process to do the default behavior for the signal use SIG_DFL.

2 Threads

2.1 Join

What does C print in the following code?
(Hint: There may be zero, one, or multiple answers.)

```
void *helper(void *arg) {
    printf("HELPER\n");
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    printf("MAIN\n");
    return 0;
}
```

```
The output of this program could be "MAIN\nHELPER\n", "HELPER\nMAIN\n" or "MAIN\n".
The actual order could be different each time the program is run.
First, the pthread_yield() does not change the answer, because it provides no
guarantee about what order the print statements execute in.
Second, the helper thread may be preempted at any point (e.g., before or after running
printf()).
Last, the main() function can return without giving enough time
for the helper thread to run, killing the process and all associated threads.
```

How can we modify the code above to always print out "HELPER" followed by "MAIN"?

```
Change pthread_yield to pthread_join.

void *helper(void *arg) {
    printf("HELPER\n");
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_join(thread, NULL);
    printf("MAIN\n");
    return 0;
}
```

2.2 Thread Stack Allocation

What does C print in the following code?

```
void *helper(void *arg) {
    int *num = (int*) arg;
    *num = 2;
    return NULL;
}

int main() {
    int i = 0;
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, &i);
    pthread_join(thread, NULL);
    printf("i is %d\n", i);
    return 0;
}
```

```
The spawned thread shares the address space with the main thread and has a
pointer to the same memory location, so i is set to 2. "i is 2"
```

2.3 Thread Heap Allocation

What does C print in the following code?

```
void *helper(void *arg) {
    char *message = (char *) arg;
    strcpy(message, "I am the child");
    return NULL;
}

int main() {
    char *message = malloc(100);
    strcpy(message, "I am the parent");
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, message);
    pthread_join(thread, NULL);
    printf("%s\n", message);
    return 0;
}
```

```
"I am the child"
```

3 Processes

3.1 Forks

How many new processes are created in the below program assuming calls to fork succeeds?

```
int main(void)
{
  for (int i = 0; i < 3; i++)
      pid_t pid = fork();
}
```

7 (8 including the original process). Newly forked processes will continue to execute the loop from wherever their parent process left off.

3.2 Process Stack Allocation

What can C print?

```
int main(void)
{
    int stuff = 5;
    pid_t pid = fork();
    printf("The last digit of pi is %d\n", stuff);
    if (pid == 0)
        stuff = 6;
}
```

The last digit of pi is 5
The last digit of pi is 5
(Since the entire address space is copied, stuff will still remain the same. There is also the fork() failure case where only one line is printed)

3.3 Process Heap Allocation

What can C print?

```
int main(void)
{
    int* stuff = malloc(sizeof(int)*1);
    *stuff = 5;
    pid_t pid = fork();
    printf("The last digit of pi is %d\n", *stuff);
    if (pid == 0)
        *stuff = 6
}
```

The last digit of pi is 5
The last digit of pi is 5
(Since the entire address space is copied, stuff will still remain the same. There is also the fork() failure case where only one line is printed)

3.4 Simple Wait

What can C print? Assume the child PID is 90210.

```
int main(void)
{
    pid_t pid = fork();
    int exit;
    if (pid != 0) {
        wait(&exit);
    }
    printf("Hello World\n: %d\n", pid);
}
```

Hello World 0
Hello World 90210 (or the failure case)

3.5 Simple Pipe

Explain how the following code snippet uses `pipe()` to communicate between processes.

```
int main(void)
{
    char msg[16];

    int fildes[2];
    if (pipe(fildes) == -1)
            return -1;

    pid_t pid = fork();
    if (pid != 0) {
        strcpy(msg, "Go Bears!");
        write(fildes[1], msg, 16);
    } else {
        read(fildes[0], msg, 16);
        printf("%s\n", msg);
    }

    return 0;
}
```

At the beginning of the program, a pipe is opened up for communication. The parent creates a message, and writes it to `fildes[1]`. The child reads from `fildes[0]` to obtain this message, and then prints it out.

4 Threads and Processes

What does C print in the following code?
(Hint: There may be zero, one, or multiple answers.)

```
void *worker(void *arg) {
    int *data = (int *) arg;
    *data = *data + 1;
    printf("Data is %d\n", *data);
    return (void *) 42;
}

int data;
int main() {
    int status;
    data = 0;
    pthread_t thread;

    pid_t pid = fork();
    if (pid == 0) {
        pthread_create(&thread, NULL, &worker, &data);
        pthread_join(thread, NULL);
    } else {
        pthread_create(&thread, NULL, &worker, &data);
        pthread_join(thread, NULL);
        pthread_create(&thread, NULL, &worker, &data);
        pthread_join(thread, NULL);
        wait(&status);
    }
    return 0;
}
```

```
One of the following is printed out:

"Data is 1"
"Data is 1"
"Data is 2"

"Data is 1"
"Data is 2"
"Data is 1"
```

How would you retrieve the return value of worker? (e.g. “42”)

```
You can use the 2nd argument of pthread_join. For example:

void *v_return_value;
pthread_join(thread, (void**)&v_return_value);
int return_value = (int)v_return_value;
```

5 Files

5.1 Files vs File Descriptor

What's the difference between `fopen` and `open`?

```
fopen is implemented in libc whereas open is a syscall. fopen will use open
in it's implementation. fopen will return a FILE * and open will return an int.
The FILE * object allows you to call utility methods from
stdio.h like fscanf. Also the FILE * object comes with some library
level buffering of writes.


--------------
|  libc      |
-------------
| syscall    |
--------------
```

5.2 Quick practice with write and seek

What will the test.txt file look like after I run this program? For simplicity assume read() and write() do not return short. (Hint: if you write at an offset past the end of file, the bytes inbetween the end of the file and the offset will be set to 0.)

```
int main() {
    char buffer[200];
    memset(buffer, 'a', 200);
    int fd = open("test.txt", O_CREAT|O_RDWR);
    write(fd, buffer, 200);
    lseek(fd, 0, SEEK_SET);
    read(fd, buffer, 100);
    lseek(fd, 500, SEEK_CUR);
    write(fd, buffer, 100);
}
```

```
The first write gives us 200 bytes of a. Then we seek to the offset 0
and read 100 bytes to get to offset 100. Then we seek to offset
100 + 500 to offset 600. Then we write 100 more bytes of a.

At then end we will have a from 0-200, 0 from 200-600, and a from 600-700
```

5.3 Reading and Writing with File Pointers vs. Descriptors

Write a utility function, **void copy(const char *src, const char *dest)**, that simply copies the file contents from src and places it in dest. You can assume both files are already created. Also assume that the src file is at most 100 bytes long. First, use the file pointer library to implement this. Fill in the code given below:

```
void copy(const char *src, const char *dest) {
  char buffer [100];
  FILE* read_file = fopen(________, ____);
  int buf_size = fread(__________, ____, __________, __________);
  fclose(read_file);

  FILE* write_file = fopen(________, ____);
  fwrite(__________, ____, __________, __________);
  fclose(write_file);
}
```

```
void copy(const char *src, const char *dest) {
  char buffer [100];
  FILE* read_file = fopen(src, "r");
  int buf_size = fread(buffer, 1, sizeof(buffer), read_file);
  fclose(read_file);

  FILE* write_file = fopen(dest, "w");
  fwrite(buffer, 1, buf_size, write_file);
  fclose(write_file);
}
```

Next, use file descriptors to implement the same thing.

```
void copy(const char *src, const char *dest) {
  char buffer [100];
  int read_fd = open(________, ________);
  int bytes_read = 0;
  int buf_size = 0;

  while ((bytes_read = read(__________, __________, __________)) > 0) {
    _____________________
  }
  close(read_fd);

  int bytes_written = 0;
  int write_fd = open(________, ________);
  while (_____________________) {
    __________ += write(__________, __________, __________);
  }
  close(write_fd);
}
```

```
void copy(const char *src, const char *dest) {
  char buffer [100];
  int read_fd = open(src, O_RDONLY);
  int bytes_read = 0;
  int buf_size = 0;

  while ((bytes_read = read(read_fd, &buffer[buf_size], sizeof(buffer) - buf_size)) > 0) {
    buf_size += bytes_read;
  }
  close(read_fd);

  int bytes_written = 0;
  int write_fd = open(dest, O_WRONLY);
  while (bytes_written < buf_size) {
    bytes_written += write(write_fd, &buffer[bytes_written], buf_size - bytes_written);
  }
  close(write_fd);
}
```

Compare the file pointer implementation to the file descriptor implementation. In the file descriptor implementation, why does **read** and **write** need to be called in a loop?

```
Read and write need to be called in a loop because there is no guarentee
that both functions will actually process the specified number of bytes
(they can return less bytes read / written). However, this functionality
is already handled in the file pointer library, so a single call to
fread and fwrite would suffice.
```