

Section 2: Synchronization, Files

CS 162

September 11, 2020

Contents

1	Vocabulary	2
2	Warmup	3
2.1	Threads and Processes	3
3	Synchronization	5
3.1	The Central Galactic Floopy Corporation	5
3.2	Crowded Video Games	7
4	Shared Data	8
5	Files	10
5.1	Files vs File Descriptor	10
5.2	Quick practice with write and seek	10
5.3	Reading and Writing with File Pointers vs. Descriptors	11

1 Vocabulary

- **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.

- **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**

2 Warmup

2.1 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;
```

3 Synchronization

3.1 The Central Galactic Floopy Corporation

It's the year 3162. Floopies are the widely recognized galactic currency. Floopies are represented in digital form only, at the Central Galactic Floopy Corporation (CGFC).

You receive some inside intel from the CGFC that they have a Galaxynet server running on some old OS called x86 Ubuntu 14.04 LTS. Anyone can send requests to it. Upon receiving a request, the server forks a POSIX thread to handle the request. In particular, you are told that sending a transfer request will create a thread that will run the following function immediately, for speedy service.

```
void transfer(account_t *donor, account_t *recipient, float amount) {
    assert (donor != recipient); // Thanks CS161

    if (donor->balance < amount) {
        printf("Insufficient funds.\n");
        return;
    }
    donor->balance -= amount;
    recipient->balance += amount;
}
```

Assume that there is some struct with a member `balance` that is typedef-ed as `account_t`. Describe how a malicious user might exploit some unintended behavior.

There are multiple race conditions here.

Suppose Alice and Bob have 5 floopies each. We send two quick requests: `transfer(&alice, &bob, 5)` and `transfer(&bob, &alice, 5)`. The first call decrements Alice's balance to 0, adds 5 to Bob's balance, but before storing 10 in Bob's balance, the next call comes in and executes to completion, decrementing Bob's balance to 0 and making Alice's balance 5. Finally we return to the first call, which just has to store 10 into Bob's balance. In the end, Alice has 5, but Bob now has 10. We have effectively duplicated 5 floopies.

Graphically:

Thread 1

```
temp1 = Alice's balance (== 5)
temp1 = temp1 - 5 (== 0)
Alice's balance = temp1 (== 0)
temp1 = Bob's balance (== 5)
temp1 = temp1 + 5 (== 10)
INTERRUPTED BY THREAD 2
```

Thread 2

```
temp2 = Bob's balance (== 5)
temp2 = temp2 - 5 (== 0)
Bob's balance = temp2 (== 0)
temp2 = Alice's balance (== 0)
temp2 = temp2 + 5 (== 5)
Alice's balance = temp2 (== 5)
THREAD 2 COMPLETE
```

RESUME THREAD 1

```
Bob's balance = temp1 (== 10)
```

THREAD 1 COMPLETE

It is also possible to achieve a negative balance. Suppose at the beginning of the function, the

donor has enough money to participate in the transfer, so we pass the conditional check for sufficient funds. Immediately after that, the donor's balance is reduced below the required amount by some other running thread. Then the transfer will go through, resulting in a negative balance for the donor.

Sending two identical `transfer(&alice, &bob, 2)` may also cause unintended behavior, since the increment/decrement operations are not atomic (though it is arguably harder to exploit for profit).

Since you're a good person who wouldn't steal floopies from a galactic corporation, what changes would you suggest to the CGFC to defend against this exploit?

The entire function must be made atomic. One could do this by disabling interrupts for that period of time (if there is a single processor), or by acquiring a lock beforehand and releasing the lock afterwards. Alternatively, you could have a lock for each account. In order to prevent deadlocks, you will have to acquire locks in some predetermined order, such as lowest account number first.

3.2 Crowded Video Games

A recent popular game is having issues with its servers lagging heavily due to too many players being connected at a time. Below is the code that a player runs to play on a server:

```
void play_session(struct server s) {  
    connect(s);  
    play();  
    disconnect(s);  
}
```

After testing, it turns out that the servers can run without lagging for a max of up to 1000 players concurrently connected.

How can you add semaphores to the above code to enforce a strict limit of 1000 players connected at a time? Assume that a game server can create semaphores and share them amongst the player threads.

Introduce a semaphore for each server, initialized to 1000, to control the ability to connect to the game. A player will `down()` the semaphore **before** connecting, and `up()` the semaphore **after** disconnecting.

The order here is important - downing the semaphore after connecting but before playing means that there is no block on the `connect()` call, and upping the semaphore before disconnecting could lead to "zombie" players, who were pre-empted before disconnecting. Both of these cases mean that the limit of 1000 could be violated.

4 Shared Data

This problem is designed to help you with implementing the wait syscall in your project. For this problem, we're going to implement a wait function that allows one thread (the main thread) to wait for another thread to finish writing some data.

We're going to assume we don't have access to `pthread_join` for this problem. Instead, we're going to use synchronization primitives (locks and semaphores).

We need to design a struct for sharing information between the two threads. We also need to implement three functions to initialize the shared struct and synchronize the 2 threads. `initialize_shared_data` will initialize our shared struct. `wait_for_data` (called by the main thread) will block until the data is available. `save_data` (called by the child thread) will write 162 to the struct. Another requirement is that the shared data needs to be freed once it is no longer in use.

Here we have already designed a possible struct for sharing information:

```
typedef struct shared_data {
    sem_t semaphore;
    pthread_mutex_t lock;
    int ref_cnt;
    int data;
} shared_data_t;
```

For each member in `shared_data` above, describe its purpose.

- The semaphore allows to perform the waiting itself. A waiting thread could down the semaphore where the thread that saves the data can up the semaphore.
- The lock allows to have mutual exclusion on members in the struct that can be modified.
- The first integer allows to perform **reference counting**. It is an indicator for how many threads still hold a reference to this struct. Once it reaches 0, we can safely deallocate it from memory.
- The second integer is the data itself that is shared.

For the following questions, refer to the following main function so that it prints "Parent: Data is 162"

```
int main() {
    void *shared_data = malloc(sizeof(shared_data_t));
    initialize_shared_data(shared_data);
    pthread_t tid;
    int error = pthread_create(&tid, NULL, &save_data, shared_data);
    int data = wait_for_data(shared_data);
    printf("Parent: Data is %d\n", data);
    return 0;
}
```


For `initialize_shared_data`, write out the pseudo-code needed to initialize all members of `shared_data`.

```
void initialize_shared_data(void *shared_pg) {
    shared_data_t *shared_data = (shared_data_t *) shared_pg;
    // init semaphore to 0
    // init lock
    // init ref_cnt to 2
    // init data to a "invalid" number (i.e. -1)
}
```

For `wait_for_data`, write out the pseudo-code needed for a thread to wait for another thread to write data to `shared_data`. Remember to deallocate the `shared_data` if it is not needed anymore.

```
int wait_for_data(void *shared_pg) {
    shared_data_t *shared_data = (shared_data_t *) shared_pg;
    // down the semaphore to wait
    // read the data
    // acquire the lock
    // decrement reference count
    // release the lock
    // free struct if ref_cnt is 0 (i.e. set a flag in the critical section)
}
```

For `save_data`, write out the pseudo-code needed for a thread to save data and signal to waiting threads. Remember to deallocate the `shared_data` if it is not needed anymore.

```
void *save_data(void *shared_pg) {
    shared_data_t *shared_data = (shared_data_t *) shared_pg;
    // set data to 162
    // up the semaphore to signal to a waiting thread
    // acquire the lock
    // decrement reference count
    // release the lock
    // free struct if ref_cnt is 0 (i.e. set a flag in the critical section)
}
```

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 fscanff. 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 guarantee 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.