Section 2: Synchronization, Wait and Exit, & Files

CS 162

February 7, 2019

Contents

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.
- **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.
- **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.
- **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.

2 Synchronization

2.1 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	**Thread 2**
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	
	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 lowerst account number first.

2.2 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.

3 Wait and Exit

This problem is designed to help you with implementing wait and exit in your project. Recall that wait suspends execution of the parent process until the child process specified by the parameter id exits, upon which it returns the exit code of the child process. In Pintos, there is a 1:1 mapping between processes and threads.

3.1 Brainstorming

"wait" requires communication between a process and its children, usually implemented through shared data. The shared data might be added to struct thread, but many solutions separate it into a separate structure. At least the following must be shared between a parent and each of its children:

- Child's exit status, so that "wait" can return it.
- Child's thread id, for "wait" to compare against its argument.
- A way for the parent to block until the child dies (usually a semaphore).
- A way for the parent and child to tell whether the other is already dead, in a race-free fashion (to ensure that their shared data can be freed).

3.2 Code

Data structures to add to thread.h for waiting logic:

```
Pseudocode:

process_wait (tid_t child_tid) {
    iterate through list of child processes
    if child process tid matches tid parameter, call sema_down
    on the semaphore associated with that child process (when
    child process exits, it will sema_up on that same semaphore,
    waking up the parent process)
    --- after waking ---
    set exit code to terminated child process's exit code
    decrease ref_cnt of child //why? need to free memory, "zombie processes"
    return exit_code
}

process_exit (void) {
    sema_up on semaphore that parent might be sleeping on
    remove all child processes from child process list
    decrement ref_cnt
}


Code:

struct wait_status
    {
      struct list_elem elem;      /* 'children' list element. */
      struct lock lock;           /* Protects ref_cnt. */
      int ref_cnt;                /* 2=child and parent both alive,
             1=either child or parent alive,
```

```
            0=child and parent both dead. */
    tid_t tid;                  /* Child thread id. */
    int exit_code;              /* Child exit code, if dead. */
    struct semaphore dead;      /* 0=child alive, 1=child dead. */
  };


struct wait_status *wait_status; /* This process's completion state. */
struct list children;            /* Completion status of children. */
```

Implement wait:

```
    - Find the child in the list of shared data structures.
      (If none is found, return -1.)

    - Wait for the child to die, by downing a semaphore in the
      shared data.

    - Obtain the child's exit code from the shared data.

    - Destroy the shared data structure and remove it from the
      list.

    - Return the exit code.
```

Implement exit:

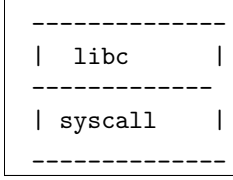
```
    - Save the exit code in the shared data.

    - Up the semaphore in the data shared with our parent
      process (if any).  In some kind of race-free way (such
      as using a lock and a reference count or pair of boolean
      variables in the shared data area), mark the shared data
      as unused by us and free it if the parent is also dead.

    - Iterate the list of children and, as in the previous
      step, mark them as no longer used by us and free them if
      the child is also dead.

      - Terminate the thread.
```

4 Files

4.1 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.



4.2 Quick practice with write and seek

What will the test.txt file look like after I run this program? (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

4.3 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.
```

4.4 Storing Ints

You are working for BigStore and your boss has tasked you with writing a function that takes an array of ints and writes it to a specified file for later use. He also informs you that a major bug has been found in the C file pointer library and wants you to use file descriptors. Fill in the following function:

```
void write_to_file(const char *file, int *a, int size) {
  int write_fd = open(________, ________);
  char *write_buf = _____________
  int buf_size = _____________
  int bytes_written = 0;
  // Write a to file.
  ___________________________
  ___________________________
  ___________________________
  close(write_fd);
}
```

```
void write_to_file(const char *file, int *a, int size) {
  int write_fd = open(file, O_WRONLY);

  char *write_buf = (char *) &a[0];
  int buf_size = size * sizeof(int);
  int bytes_written = 0;

  while (bytes_written < buf_size) {
    bytes_written += write(write_fd, &write_buf[bytes_written], buf_size - bytes_written);
  }
  close(write_fd);
}
```

Now, write the function that retrieves previously saved integers and places them in a int array.

```
void read_from_file(const char *file, int *a, int size) {
  int read_fd = open(________, ________);
  char *read_buf = _____________
  int buf_size = _____________
  // Read a from a file.
  ___________________________
  ___________________________
  ___________________________
  ___________________________
  ___________________________
  close(read_fd);
}
```

```
void read_from_file(const char *file, int *a, int size) {
  int read_fd = open(file, O_RDONLY);

  char *read_buf = (char *) &a[0];
  int buf_size = size * sizeof(int);

  int bytes_read = 0;
  int total_read = 0;
  while ((bytes_read = read(read_fd, &read_buf[total_read], buf_size - total_read)) > 0) {
    total_read += bytes_read;
  }
  close(read_fd);
}
```

Your coworker opens up one of the files that you used to store ints on his text editor and complains its full of junk! Explain to him why this might be the case.

```
Currently, we are reading and writing the contents of memory directly to disk.
This is convinient for us, because we do not have to any parsing of the input.
However, the memory representation of an int array is unlikely to be human readable.
```