Section 4: Synchronization, Locks

CS 162

July 6, 2020

Contents

1 Vocabulary

- **Lock** - Synchronization variables 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.

- **test_and_set** - An atomic operation implemented in hardware. Often used to implement locks and other synchronization primitives. In this handout, assume the following implementation.

```
int test_and_set(int *value) {
    int result = *value;
    *value = 1;
    return result;
}
```

  This is more expensive than most other instructions, and it is not preferable to repeatedly execute this instruction.

- **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 an be modified:

  - When a sempahore is initialized, the integer is set to a specified starting value.
  - A thread can call **down()** (also know 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.

- **Race Condition** - A state of execution that causes multiple threads to access the same shared variable (heap or global data segment) with at least one thread attempting to execute a write without enforcing mutual exclusion. The result is not necessarily garbage but is treated as being undefined since there is no guarantee as to what will actually happen. Note that multiple reads do not need to be mutexed.

- **Critical section** - A section of code that accesses a shared resource and must not be concurrently run by more than a single thread.

- **TCP** - Transmission Control Protocol (TCP) is a common L4 (transport layer) protocol that guarantees reliable in-order delivery. In-order delivery is accomplished through the use of sequence numbers attached to every data packet, and reliable delivery is accomplished through the use of ACKs (acknowledgements).

- **Device Driver** - Device-specific code in the kernel that interacts directly with the device hardware. They support a standard, internal interface so the same kernel I/O system can interact easily with different hardware. The top half of a device driver is used by the kernel to start I/O operations. The bottom half of a device driver services interrupts produced by the device. You should know that Linux has different definitions for “top half” and “bottom half”, which are essentially the reverse of these definitions (top half in Linux is the interrupt service routine, whereas the bottom half is the kernel-level bookkeeping).

2 Synchronization

2.1 Locking via Disabling Interrupts

Consider the following implementation of Locks:

```
Lock::Acquire() {
    disable_interrupts();
}
```

```
Lock::Release() {
    enable_interrupts();
}
```

1. For a single-processor system state whether this implementation is incorrect.

2. For a multiprocessor system, explain what additional reason(s) might make this implementation incorrect?

2.2 Counting Semaphores

We have a limited number of resources such that only N threads can use them at any given time. Explain how we can use a "counting semaphore" to control access to these resources.

2.3 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.

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?

2.4 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.

3 Additonal Questions

These are questions we may not get to in discussion but you may find useful when studying for the exam.

3.1 Signals

The following is a list of standard Linux signals:

```
Signal     Value      Action       Comment
----------------------------------------------------------------------
SIGHUP        1       Terminate    Hangup detected on controlling terminal
                                   or death of controlling process
SIGINT        2       Terminate    Interrupt from keyboard (Ctrl - c)
SIGQUIT       3       Core Dump    Quit from keyboard (Ctrl - \)
SIGILL        4       Core Dump    Illegal Instruction
SIGABRT       6       Core Dump    Abort signal from abort(3)
SIGFPE        8       Core Dump    Floating point exception
SIGKILL       9       Terminate    Kill signal
SIGSEGV      11       Core Dump    Invalid memory reference
SIGPIPE      13       Terminate    Broken pipe: write to pipe with no
                                   readers
SIGALRM      14       Terminate    Timer signal from alarm(2)
SIGTERM      15       Terminate    Termination signal
SIGUSR1   30,10,16    Terminate    User-defined signal 1
SIGUSR2   31,12,17    Terminate    User-defined signal 2
SIGCHLD   20,17,18    Ignore       Child stopped or terminated
SIGCONT   19,18,25    Continue     Continue if stopped
SIGSTOP   17,19,23    Stop         Stop process
SIGTSTP   18,20,24    Stop         Stop typed at tty
SIGTTIN   21,21,26    Stop         tty input for background process
SIGTTOU   22,22,27    Stop         tty output for background process
```

3.2 Signal Handlers

Fill in the blanks for the following function using syscalls such that when we type Ctrl-C, the user is prompted with a message: "Do you really want to quit [y/n]? ", and if "y" is typed, the program quits. Otherwise, it continues along.

```
void sigint_handler(int sig)
{
    char c;
    printf(\Ouch, you just hit Ctrl-C?. Do you really want to quit [y/n]?");
    c = getchar();
    if (c == 'y' || c == 'Y')
        ______________________________;
}

int main() {
    ______________________________;
    ...
```

```
}
```

3.3 Exec

What will C print?

```
int main(void)
{
    char** argv = (char**) malloc(3*sizeof(char*));
    argv[0] = "/bin/ls";
    argv[1] = ".";
    argv[2] = NULL;
    for (int i = 0; i < 10; i++) {
        printf("%d\n", i);
        if (i == 3)
            execv("/bin/ls", argv);
    }
}
```

3.4 Exec + Fork

How would I modify the above program using fork so it both prints the output of `ls` and all the numbers from 0 to 9 (order does not matter)? You may not remove lines from the original program; only add statements (and use fork!).

3.5 RPC Consistency

In our RPC example (Section 3) we opted to use the type `uint32_t`. What is a potentially issue that could result if we opted to use an `unsigned int` instead? If a value can only take either 0 or 1 is could your previous answer still be an issue?

3.6 Socket Choices

RPC calls are implemented with socket implementations. In lecture you saw running a server with 1 thread and last week in discussion you looked at sockets with multiple threads. Below is the code for running a server where each request is its own process.

```
struct addrinfo *setup_address (char *port) {
    struct addrinfo *server;
    struct addrinfo hints;
    memset (&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    iF (getaddrinfo(NULL, port, &hints, &server) != 0) {
        perror ("Address");
        return NULL;
    }
    return server;
}
```

```
int setup_server_socket (struct addrinfo *server) {
    // Setup the addr socket
    bool connected = false;
    int sock = -1;
    for (struct addrinfo *addr = server; addr != NULL && !connected; addr = addr->ai_next) {
        int sock = socket (addr->ai_family, addr->ai_socktype, addr->ai_protocol);
        if (sock != -1) {
            if (bind(sock, addr->ai_addr, addr->ai_addrlen) == -1)
                close (sock);
                sock = -1;
            } else {
                return sock;
            }
        }
    }
    return sock;
}

void run_server (char *port) {
    struct addrinfo *server = setup_address(port);
    int server_fd = setup_server_socket (server);
    if (server_fd == -1) {
        perror ("socket");
    } else {
        int rv = listen (server_fd, BACKLOG);
        if (rv < 0) {
            perror ("socket");
        } else {
            while (1) {
                int connection_socket = accept (server_fd, NULL, NULL);
                int pid = fork ();
                if (pid < 0) {
                    perror ("fork");
                } else if (pid == 0) {
                    receive_rpc (connection_socket);
                    exit (0);
                } else {
                    close (connection_socket);
                }
            }
        }
    }
}
```

What's a possible advantage of playing each RPC request in a separate process instead of a separate thread?