Section 3: Threads and Locks

CS162

July 3, 2019

Contents

1 Vocabulary 2

2 Problems 3
  2.1 Hello World ......................................................... 3
  2.2 Join ................................................................. 3
  2.3 Stack Allocation .................................................... 4
  2.4 Heap Allocation .................................................... 5
  2.5 Threads and Processes ............................................. 5
  2.6 The Central Galactic Floopy Corporation ..................... 7
  2.7 All Threads Must Die .............................................. 8
1 Vocabulary

- **Thread** - a thread of execution is the smallest unit of sequential instructions that can be scheduled for execution by the operating system. Multiple threads can share the same address space, but each thread independently operates using its own program counter.

- **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.

  ```c
  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).

  ```c
  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.

  ```c
  int pthread_yield(void);
  ```

- **Atomic operation** - An operation that appears to be indivisible to observers. Atomic operations must execute to completion or not at all.

- **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 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.

- **Scheduler** - Routine in the kernel that picks which thread to run next given a vacant CPU and a ready queue of unblocked threads. See next_thread_to_run() in Pintos.

- **Priority Inversion** - If a higher priority thread is blocking on a resource (a lock, as far as you’re concerned but it could be the Disk or other I/O device in practice) that a lower priority thread holds exclusive access to, the priorities are said to be inverted. The higher priority thread cannot continue until the lower priority thread releases the resource. This can be amendsed by implementing priority donation.
• **Priority Donation** - If a thread attempts to acquire a resource (lock) that is currently being held, it donates its effective priority to the holder of that resource. This must be done recursively until a thread holding no locks is found, even if the current thread has a lower priority than the current resource holder. (Think about what would happen if you didn’t do this and a third thread with higher priority than either of the two current ones donates to the original donor.) Each thread’s effective priority becomes the max of all donated priorities and its original priority.

2 Problems

2.1 Hello World

What does C print in the following code?

```c
void* identify(void* arg) {
    pid_t pid = getpid();
    printf("My pid is \%d\n", pid);
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &identify, NULL);
    identify(NULL);
    return 0;
}
```

*My pid is 2617*

Either twice (context switch between create and return) or once (no context switch; program exits before second thread is run)

2.2 Join

What does C print in the following code?

(Hint: There may be zero, one, or multiple answers.)

```c
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.

```c
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;
}
```

2.3 Stack Allocation

What does C print in the following code?

```c
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"
2.4 Heap Allocation

What does C print in the following code?

```c
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"

2.5 Threads and Processes

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

```c
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:

```c
void *v_return_value;
pthread_join(thread, (void**)&v_return_value);
int return_value = (int)v_return_value;
```
2.6 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.

```c
void transfer(account_t *donor, account_t *recipient, float amount) {
    assert (donor != recipient);
    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 Alices balance to 0, adds 5 to Bobs balance, but before storing 10 in Bobs balance, the next call comes in and executes to completion, decrementing Bobs balance to 0 and making Alices balance 5. Finally we return to the first call, which just has to store 10 into Bobs balance. In the end, Alice has 5, but Bob now has 10. We have effectively duplicated 5 floopies.

Graphically:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>temp1 = Alice's balance (== 5)</code></td>
<td><code>temp2 = Bob's balance (== 5)</code></td>
</tr>
<tr>
<td><code>temp1 = temp1 - 5 (== 0)</code></td>
<td><code>temp2 = temp2 - 5 (== 0)</code></td>
</tr>
<tr>
<td>Alice's balance = temp1 (== 0)</td>
<td>Bob's balance = temp2 (== 0)</td>
</tr>
<tr>
<td><code>temp1 = Bob's balance (== 5)</code></td>
<td><code>temp2 = Alice's balance (== 0)</code></td>
</tr>
<tr>
<td><code>temp1 = temp1 + 5 (== 10)</code></td>
<td><code>temp2 = temp2 + 5 (== 5)</code></td>
</tr>
<tr>
<td>INTERRUPTED BY THREAD 2</td>
<td>THREAD 2 COMPLETE</td>
</tr>
</tbody>
</table>

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 donors 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
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.

2.7 All Threads Must Die

You have three Pintos threads with the associated priorities shown below. They each run the functions with their respective names.

Tyrion: 4  
Ned: 5  
Gandalf: 11

Assume upon execution that all threads are unblocked and begin at the top of their code blocks. The operating system runs with a preemptive priority scheduler. You may assume that set_priority commands are atomic. (Note: The following uses references to Pintos locks and data structures.)

```c
struct list brace_yourself; // pintos list. Assume it's already initialized and populated.
struct lock midterm; // pintos lock. Already initialized.
struct lock is_coming;

void tyrion(){
    thread_set_priority(12);
    lock_acquire(&midterm);
    lock_release(&midterm);
    thread_exit();
}

void ned(){
    lock_acquire(&midterm);
    lock_acquire(&is_coming);
    list_remove(list_head(brace_yourself));
    lock_release(&midterm);
    lock_release(&is_coming);
    thread_exit();
}

void gandalf(){
    lock_acquire(&is_coming);
    thread_set_priority(3);
    while (thread_get_priority() < 11) {
        printf("YOU .. SHALL NOT .. PAAASS!!!!!!\n");
        timer_sleep(20);
    
```
What is the output of this program when there is no priority donation? Trace the program execution and number the lines in the order in which they are executed.

```c
void tyrion(){
    5    thread_set_priority(12);
    6    lock_acquire(&midterm); //blocks
    7    lock_release(&midterm);
    8    thread_exit();
}

void ned(){
    3    lock_acquire(&midterm);
    4    lock_acquire(&is_coming); //blocks
    5    list_remove(list_head(brace_yourself));
    6    lock_release(&midterm);
    7    lock_release(&is_coming);
    8    thread_exit();
}

void gandalf(){
    1    lock_acquire(&is_coming);
    2    thread_set_priority(3);
    3    while (thread_get_priority() < 11) {
    4        printf("YOU .. SHALL NOT .. PAAASS!!!!!!); //repeat till infinity
    5        timer_sleep(20);
    6    }
    7    lock_release(&is_coming);
    8    thread_exit();
}
```

Gandalf, as you might expect, endlessly prints "YOU SHALL NOT PASS!!" every 20 clock ticks or so.\*

What is the output and order of line execution if priority donation was implemented? Draw a diagram of the three threads and two locks that shows how you would use data structures and struct members (variables and pointers, etc) to implement priority donation for this example.

```c
void tyrion(){
    8    thread_set_priority(12);
    9    lock_acquire(&midterm); //blocks
    10    lock_release(&midterm);
    11    thread_exit();
}

void ned(){
    3    lock_acquire(&midterm);
    4    lock_acquire(&is_coming); //blocks
```
It turns out that Gandalf generally does mean well. Donations will make Gandalf allow you to pass. At some point Gandalf will sleep on a timer and leave Tyrion alone in the ready queue. Tyrion will run even though he has a lower priority (Gandalf has a 5 donated to him) Tyrion then sets his priority to 12 and chain-donates to Gandalf. Gandalf breaks his loop. Ned unblocks after Gandalf releases the is_coming lock. However, allowing Ned to remove the head of a list will trigger an ASSERT failure in lib/kernel/list.c.

Gandalf will print YOU SHALL NOT PASS at least once. Then Ned will get beheaded and cause a kernel panic that crashes Pintos.