Section 4: Threads

September 19, 2015

Contents

1 Warmup

1.1 Hello World

What does C print in the following code?

```
void print_hello_world() {
    pid_t pid = getpid();
    printf("Hello world %d\n", pid);
    pthread_exit(0);
}

void main() {
    pthread_t thread;
    pthread_create(&thread, NULL, (void *) &print_hello_world, NULL);
    print_hello_world();
}
```

```
Hello world 2617
Hello World 2617
```

2 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, and each thread independently operates using its own program counter.
- **pthreads** - A POSIX-compliant (standard specified by IEEE) implementation of threads.
- **pthread_join** - Waits for a specific thread to terminate, similar to `waitpid(3)`. (Hint: `man pthread_join`)
- **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. (Hint: `man pthread_create`)
- **atomic** - An operation is deemed to be atomic if it can be executed without interruption or interference from other threads/processes).
- **critical section** - A section of code that accesses a shared resource that must not be concurrently accessed by more than a single thread.
- **semaphore** - A synchronization primitive that can be used to protect a shared resource. Semaphores contain an integer value and support two operations:
  1) Increment: atomically increments the value of the semaphore. (Hint: `man sem_post`)
  2) Decrement: waits for the value of the semaphore to become positive, and then atomically decrements the value of the semaphore. (Hint: `man sem_wait`)
- **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.

- **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.

3 Problems

3.1 Join

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

```
void main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    printf("Hello World! 2\n");
    exit(0);
}

void *helper(void* arg) {
    printf("Hello World! 1\n");
    pthread_exit(0);
}
```

```
The output of this program is undefined because the order in which lines
are executed can be different each time the program is run.
Thus, C could print out any combination of "Hello World! 1" and "Hello World! 2".
```

How can we modify the code above to always print out `"Hello World! 1"` followed by `"Hello World! 2"`?

```
Add a pthread_join to coordinate execution.

void main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_join(thread, NULL);
    printf("Hello World! 2\n");
    exit(0);
}

void *helper(void* arg) {
    printf("Hello World! 1\n");
    pthread_exit(0);
}
```

3.2 Stack Allocation

What does C print in the following code?

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

void *helper(void *arg) {
    int *num = (int*) arg;
    *num = 2;
    pthread_exit(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"
```

3.3 Heap Allocation

What does C print in the following code?

```
void 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);
}

void *helper(void *arg) {
    char *message = (char *) arg;
    strcpy(message, "I am the child");
    pthread_exit(0);
}
```

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

3.4 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);
    pthread_exit(0);
}

int data;
void 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);
    }
}
```

```
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"
```

3.5 Atomicity

Given:

```
int x = 0;
```

Circle all the atomic operations below:

```
printf("x is %d\n", x);

x = malloc(sizeof(int));
```

```
int y = x;

x++;
```

None of the operations listed are atomic. 1) Printing to stdout is not atomic because it involves many operations, including writing to a file, some of which are blocking. 2) Allocating memory is not atomic. 3) Assigning x to y is not atomic because it involves multiple loads/stores to registers. 4) Incrementing x is not atomic because it involves a load, store, and add operation.

3.6 Synchronization

What does C print in the following code?

```
void main() {
    pthread_t threads[5];
    int i, j = 0;
    for (i = 0; i < 5; i++) {
        pthread_create(&threads[i], NULL, &helper, &j);
    }
    for (i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("j is %d\n", j);
}

void *helper(void *arg) {
    int *num = (int *) arg;
    *num = (*num) + 1;
    pthread_exit(0);
}
```

```
Five threads are trying to increment j in parallel,
and since incrementing isn't an atomic operation, anything could be printed out.
```

How can we modify the code in the previous page to always print out `"i is 5"` without modifying/using `pthread_join`?

```
Protect access to j using a semaphore.

sem_t sem;
void main() {
    pthread_t threads[5];
    int i, j = 0;
    sem_init(&sem, 0, 0);
    for (i = 0; i < 5; i++) {
        pthread_create(&threads[i], NULL, &helper, &j);
    }
    sem_post(&sem);
    for (i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("j is %d\n", j);
}

void *helper(void *arg) {
    sem_wait(&sem);
    int *num = (int *) arg;
    *num = (*num) + 1;
    sem_post(&sem);
    pthread_exit(0);
}
```

3.7 Who's Your Daddy

What does C print in the following code? Assume the new thread's ID is 0x4c554b45. Assume Round Robin Scheduling and no other threads/processes are running.

```
pthread_t thread_id = 0x0;

void print_hello_world() {
    thread_id = pthread_self();  //like getpid but for threads
    printf("You deallocated my father!\n");
    pthread_exit(0);
}

void main() {
    pthread_t thread;
    pthread_create(&thread, NULL, (void *) &print_hello_world, NULL);
    pthread_yield();
    printf("No, thread %#010x, I AM your father!!"\n, thread_id);
}
```

```
No, thread 0x4c554b45, I AM your father!!
You deallocated my father!

OR

You deallocated my father!
No, thread 0x4c554b45, I AM your father!!

OR

No, thread 0x0, I AM your father!!
You deallocated my father!

OR

garbage / other combinations possible if store on thread_id is interrupted at
assembly language level (x86).

Round robin wait queues are FIFO so you'll at least force the new thread
to start the global variable store before the parent prints. But without
```

```
synchronization or any notion of the time quantum the store is not guaranteed
to finish. It could be forced to yield the CPU at any time again, which may
create a race condition on thread_id.

Note that the following is INCORRECT:
You deallocated my father!
No, thread 0x0, I AM your father!
(impossible for parent to print 0x0 after child reached print statement
because execution in the same thread is still serial)
```

3.8 Simple HTTP Server with Threads

Implement a crude HTTP server by filling in the blanks with the necessary operations. This HTTP server spawns worker threads that are responsible for printing out the first 256 bytes from a socket. For this question, you can assume the following structures and functions are implemented appropriately.

```
// a FIFO queue consisting of nodes.
struct list;
// A single element in the FIFO queue, containing an integer.
struct node {
    int client_sock;
};

// This function sets up a socket, binds the socket to a specific port,
// and listens for connections.  It returns the file descriptor
// corresponding to the socket setup.
int setupSocket();

// initializes a FIFO queue
void list_init(struct list *l);
 // Pops and returns the earliest enqueued node from a list
struct node *list_pop(struct list *l);
// Appends a new file descriptor as a struct node in the list.
struct node *list_append(struct list *l, int client_sock);
```

```
// Global variables
struct          list clients;
sem_t           worker_sem;
sem_t           list_sem;

// Prints out the first 256 bytes from a socket, and then closes it.
void accept_request(void* unused) {
    char buf[256];
    (void) unused;

    while(1) {
        sem_wait(&worker_sem);
        sem_wait(&list_sem);
        int sockfd = list_pop(&clients)->client_sock;
        sem_post(&list_sem);
        if (read(sockfd, buf, 256) != 256) {
            fprintf(stderr, "Failed to read from client %d\n", sockfd);
        } else {
            fprintf(stdout, "Received from client %d: %s\n", sockfd, buf);
        }
        close(sockfd);
    }
}

void main() {
    pthread_t workers[10];
    int client_sock;
    struct sockaddr_in client_name;
    socklen_t size;
    int sockfd;

    list_init(&clients);
    sem_init(&worker_sem, 0, 0);
    sem_init(&list_sem, 0, 1);
    sockfd = setupSocket();

    // create worker threads
    for (int i = 0; i < 10; i++) {
        pthread_create(&workers[i], NULL, &accept_request, NULL);
    }

    while (1) {
        client_sock = accept(sockfd, (struct sockaddr *) &client_name, &size);
        sem_wait(&list_sem);
        list_append(&clients, client_sock);
        sem_post(&list_sem);
        sem_post(&worker_sem);
    }
}
```