COMS3010A: Operating Systems
Tutorial 2: Threads
Solutions

September 14, 2019

Contents

1 Questions **2**

1.1 Threads . . . 2
1.2 Join . . . 3
1.3 Stack Allocation . . . 4
1.4 Heap Allocation . . . 4
1.5 Atomic Operations . . . 5
1.6 Synchronization . . . 6
1.6.1 Semaphores . . . 7
1.6.2 Mutex Locks . . . 7

1 Questions

1.1 Threads

What output is produced by the following code? Assume the PID of the parent process is 26713.

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

int main( void ) {
    pthread_t thread;
    pthread_create(&thread, NULL, hello, NULL);
    pthread_join(thread, NULL);
    hello();
}
```

Answer:
"Hello world 26713"

The new thread is still part of the same process, and so will have the same PID.

1.2 Join

```
void*
helper() {
    printf( "1\n" );
    pthread_exit( NULL );
}

int
main( void ) {
    pthread_t thread;
    pthread_create( &thread, NULL, helper, NULL );
    printf( "2\n" );
}
```

1. List all possible outputs of the code.
   (Hint: The program may exit before the new thread is run.)

   Answer:
   1
   2
   OR
   2
   1
   OR
   2

   The order in which the print statements are executed depends on how the scheduler decides to schedule the two threads in the program. The third output is possible since the main program could complete and exit before the new thread is scheduled to run.

2. Alter the program so that it always outputs '1' then '2'.

```
void*
helper() {
    printf( "1\n" );
    pthread_exit( NULL );
}

int
main( void ) {
    pthread_t thread;
    pthread_create( &thread, NULL, helper, NULL );
    pthread_join(thread, NULL);
    printf( "2\n" );
}
```

1.3 Stack Allocation

What is the output of the following code?

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

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

Answer:
"i is 2"

Both threads share the same address space. The spawned thread can therefore access and alter the stack of the main thread through a pointer.

1.4 Heap Allocation

What is the output of the following code?

```
void*
helper( void *arg ) {
    char *message = (char *) arg;
    strcpy( message, "I know how to use threads" );
    pthread_exit( NULL );
}

int
main( void ) {
    char *message = malloc(26);
    strcpy(message, "I can't use threads");
    pthread_t thread;
    pthread_create( &thread, NULL, helper, message );
    pthread_join( thread, NULL );
    printf( "%s\n", message );
}
```

Answer:
"I know how to use threads"

Both threads share the same address space. The spawned thread can therefore access and alter the heap of the main thread.

Why would spawning processes rather than threads produce different output in the previous two questions?

Answer:
Using fork() to spawn new processes creates a copy of the the parent's address space. Child processes therefore cannot alter anything in the parent process' stack or heap; any changes made to variables by the child would not affect the parent. Threads share the same address space, so any changes made to the global heap or another thread's stack can be seen by all threads, including the main/parent thread.

1.5 Atomic Operations

Given the declarations:

```
int x = 0;
void* data;
```

Identify all atomic operations in the following lines of code.

```
++x;
int y = x;
printf("x is %d\n", x);
data = malloc(8);
x = 10;
```

Answer:
None of the listed operations are guaranteed to be atomic by the C language standard. This is why explicit synchronisation of shared data structures is needed when working with threads.

1. Incrementing x involves a load, an add, and a store operation. Multiple assembly level instructions could be interrupted by a context switch.
2. Assigning x to y is not atomic since it involves multiple loads and stores to registers.
3. Printing to stdout involves many operations, including writing to a file.
4. Allocating memory is not atomic.
5. Atomicity of simple integer assignment still depends on the architecture and implementation. While it may be atomic on most platforms, the C standard does not guarantee it.

1.6 Synchronization

Explain why the following code can produce incorrect results.

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

int
main( void ) {
    pthread_t threads[5];
    int data = 0;

    int i = 0;
    for(i = 0; i < 5; i++)
        pthread_create( &threads[i], NULL, helper, &data );
    for(i = 0; i < 5; i++)
        pthread_join( threads[i], NULL );

    printf("Data is %d\n", data);
}
```

Answer:
The output of the code is undefined, as multiple threads attempt to increment 'data' in parallel, and increment is not an atomic operation. Since no synchronisation is done to protect the shared variable 'data', lost updates can occur when multiple threads attempt to increment at the same time. This is a data race, a type of race condition.

1.6.1 Semaphores

Fill in the specified lines of code to use a semaphore for synchronisation. If you want to run the code, you need to include the semaphore.h header.

```
sem_t lock;

void*
helper( void *arg ) {
    sem_wait(&lock);
    int *num = (int*) arg;
    *num = (*num) + 1;
    sem_post(&lock);
    pthread_exit(NULL);
}

int
main( void ) {
    pthread_t threads[5];
    int data = 0;
    int i = 0;

    sem_init(&lock,0,1);
    for(i = 0; i < 5; i++)
        pthread_create( &threads[i], NULL, helper, &data );

    for(i = 0; i < 5; i++)
        pthread_join( threads[i], NULL );

    sem_destroy(&lock);

    printf("Data is %d\n", data);
}
```

1.6.2 Mutex Locks

Fill in the specified lines of code to use a mutex lock for synchronisation.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void*
helper( void *arg ) {
    pthread_mutex_lock(&lock);
    int *num = (int*) arg;
    *num = (*num) + 1;
    pthread_mutex_unlock(&lock);
    pthread_exit(NULL);
}
```