ECE 254 S16 Midterm Solutions

J. Zarnett

June 16, 2016

(1A)

If it fails, the output will be `-1`. If it succeeds, one of the two processes will print `0` and the other will print the process id of the child (e.g., `24601`).

(1B) Not the only answers but anything like this will be adequate:

1. There are protections between processes and the OS needs to know it is okay for process 1 to access memory of process 2.

2. If process 2 allocates the memory and it's shared with process 1, when process 2 exits, the shared memory cannot be deallocated like normal, so the OS must be aware that another process is using the memory.

(1C) When a joinable thread is finished, it cannot be cleaned up (its resources deallocated) until it is joined and the return value collected. A detached thread, when it finishes, is cleaned up automatically (because no other thread is going to collect the result).

(1D)

No, there is no data race. Although two threads are created, the first thread is created and then immediately joined; the main thread awaits for thread 1 to run and collects the result before creating the second thread. Then it waits for the second thread before it goes on to the print statement.

(1E)

```
void *sum( void *void_arg ) {
    int *start = (int *) void_arg;
    int *sum = malloc( sizeof( int ) );
    *sum = 0; // Default value

    for ( int i = *start; i < ARRAY_LENGTH; i += NUM_CPUS) {
     *sum = *sum + array[i];
    }
    free( void_arg );
    pthread_exit(sum);
}
```

Potential pitfalls:

- Forgetting to free the passed arguments
- Freeing both the passed argument and an alias of it (sum)
- Freeing the sum we should be returning
- Pointer problems (not dereferencing correctly)
- Not using malloc for the sum and instead returning a local variable
- Forgetting to initialize sum
- Going off the end of the array or not using the skip strategy (for loop wrong)

(1F)

```
int main( int argc, char** argv ) {
    pthread_t threads[NUM_CPUS];
    int global_sum = 0;

    for ( int i = 0; i < NUM_CPUS; ++i ) {
        int* start = malloc( sizeof( int ) );
        *start = i;
        pthread_create(&threads[i], NULL, sum, start);
    }

    void* temp_sum;
    for ( int i = 0; i < NUM_CPUS; ++i ) {
        pthread_join( threads[i], &temp_sum );
        int *thread_sum = (int *) temp_sum;
        global_sum += *thread_sum;
        free( thread_sum );
    }
    printf("The sum is %d", global_sum);

    pthread_exit(0);
}
```

Potential pitfalls:

- Forgetting to free the returned sum when finished with it
- Double free on the returned item
- Freeing the start parameter given to the sum function
- Pointer problems (not dereferencing correctly)
- Not using malloc for the start value and instead using local variable i
- Forgetting to initialize the global sum or other variables
- Misuse of pthread functions
- Create followed immediately by Join

(2A)

The key problem is that `sem` is a binary semaphore and if it gets signalled twice without a wait in between, the second signal is "lost" because the value cannot be incremented beyond 1. Most of the marks are awarded for understanding this is the problem.

A full scenario to cause the problem:

- Start with four processes, P1 to P4 , `s = 0, delay = 0`
- P1 and P2 get past `signal(sem)` in `semWait`; `s =-2`
- P3 completes `semSignal`
- `s = -1, delay = 1`
- P4 completes `semSignal`
- `s = 0, delay = 1`
- But P4's `signal(delay)` is lost; `delay` is a *binary* semaphore and its value cannot exceed 1!
- If P1 proceeds, P2 will be stuck at `wait(delay)`, until `semSignal` is called again

If you found a different problem, e.g., starvation or deadlock not as described, and explained it well, then you get full marks.

(2B)

The key to the solution is that if anything is blocked in `semWait` then `sem` will not be signalled in `semSignal`. If `s` is less than zero, some thread is (or will soon be) waiting on `delay`. But what we don't want is two signals on `delay` taking place with no wait in between, which could happen if we signal `sem`.

In a practical sense it means the `signal(sem)` in `semWait` is unconditional, and in `semSignal` it becomes conditional.

```
void semWait(semaphore s) {
    wait(sem);
    s--;
    if (s < 0) {
        signal(sem);
        wait(delay);
    }
    signal(sem);
}
```

```
void semSignal(semaphore s) {
    wait(sem);
    s++;
    if (s <= 0) {
        signal(delay);
    } else {
        signal(sem);
    }
}
```

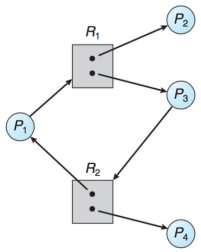
(The solution is not very optimal but it at least avoids the problem in part A!)

If your solution ameliorates the problem you identified in A, even if that was a different problem, full marks.

(2C)

Assume there is a deadlock. This means every philosopher is waiting, thus, all chopsticks are held by someone. There is one left-handed philosopher sitting next to a right-handed philosopher (the table is a circle). The chopstick between them will be the second chopstick for both parties. Thus, if the left-handed philosopher got it, it's his second chopstick and and he can eat – no deadlock. If the right handed-philosopher got it, it's his second chopstick and he can eat – no deadlock. If neither is the case, then the chopstick is not held and there is no deadlock. In each of those three scenarios, the assumption of deadlock is contradicted.

(3A) This is an example from the lecture notes:



Any valid graph with a cycle but no deadlock is acceptable.

(3B) The banker's algorithm requires knowledge of the resource requests that a process will make in the future. For this reason, it will not work in a general purpose operating system where users can run arbitrary programs at any time (that don't announce their needs).

(3C) Deadlock detection is expensive (computationally); if the expected costs of detecting deadlock exceed the expected costs of having deadlocked processes, then it is sensible to just ignore deadlock. Choosing not to implement it is also easier from the perspective of the OS designers.

(3D) In lectures we discussed: Robbery (Preemption), Mass Murder (kill all processes in the deadlock), Murder (kill selected processes in the deadlock), Time Travel (rollback), Armageddon (reboot system). (Saying "do nothing" is not okay here though.)

Choose any two of those and give one pro and one con for each of those two.