[1] 1. Two threads run through this code: What will be the final contents of the array? Also, **Assume that get_counter() has its own internal locking (not shown), and will return 1 when first called, and 2 when called next, etc.**

```
mutex_t m;
int slot = -1;
int array[2] = { 0, 0 }; // initialize to 0, 0

function1() {
    mutex_acquire(&m);
    slot++;
    int tid = get_counter();
    array[slot] = tid;
    mutex_release(&m);
}
```

CRITICAL SECTION

Solution: When the first thread (say, T1) passes through function1(), it acquires the lock, and sets array[0] = 1. Then, the second thread (say, T2) can only access the critical section (marked above) once T1 releases the mutex lock. Finally, when T2 acquires the lock, it sets array[1] = 2. Then, T2 releases the lock.

Hence, final contents of the array are: {1, 2}

[2] 2. *Student X knows that he can use semaphores as locks, and locks as locks. So he thinks why not use both? His implementation is as follows:* **There are two threads. One thread uses a lock, the other a semaphore, and they run concurrently. What will be the final value of the variable *count ? Justify your answer***

```
int count = 0;
mutex_t m;
sem_t s; // the semaphore s is initialized to 1
//thread 1:
mutex_acquire(&m);
count++;
mutex_release(&m);
```

```
//thread 2:
sem_wait(&s);
count++;
sem_post(&s);
```

Final Answer: In worst case, if the below description occurs, then. count = 1

Solution: considering the worst case scenario, thread 1 is in the middle of the assembly code segment corresponding to 'count++', when thread 2 gets the CPU, does its operations, sets count to 1, before the 'count++' of thread 1 also writes to count, (as '1'). This happens because 's' and 'm' are defined separately here, and T2 doesn't check m, while T1 doesn't check s. Essentially, both threads can enter the critical section (count++) at the same time. So, no actual lock functionality occurs, since 'count' is still accessible to one thread even when the other has "locked" the code segment.

2

[1] **3. What will be the output of the following code: Justify your answer**

```
1  void child(void *arg) {
2      int *p = (int *) arg;
3      printf("%d\n", *p);
4  }
5  int main(int argc, char *argv[]) {
6      int x = 3;
7      thread_t p1;
8      thread_create(&p1, child, &x);
9      thread_join(p1);
10     return 0;
11 }
```

NULL, x, 3

If the whole assembly code segment corresponding to thread 1's (count++) finishes before thread 2 begins the (count++), Then, (count = 2)

Solution: In Line 6, x is set to 3. Then, thread p1 is declared (Line 7). Next, thread is created, and child() function is invoked with argument '3', as part of the thread p1. This does a call by reference, sending address of x to child(), which ~~then gives the value of~~ sets pointer 'p' to the same address as 'arg'. Then, 3 would be printed. Hence, output is 3.

~~However, child() takes argument of type (void*)~~

1

as long as line 1 doesn't manipulate arg

[1] **4. What are the possible outputs that will get printed ? Justify your answer**

```
1  cond_t c;
2  mutex_t m;
3  void child(void *arg) {
4      printf("child\n");
5      cond_signal(&c);
6  }
7  int main() {
8      thread_t p1;
9      thread_create(&p1, child, NULL);
10     cond_wait(&c, &m);
11     printf("done\n");
12 return 0;
13 }
```

Also note (IMPORTANT)
Since child takes argument of type (void *), it takes pointer to NULL as input. So, if child() typecasts and makes arg a null pointer, then output would be unpredictable (junk value) or NULL, since '3' is no longer there

Note that, thread_join() is used to make the parent wait for child thread to complete.

(Sorry, not enough space provided!)

assume mutex_t 'm' is unlocked initially

Solution: Initially, a thread p1 is created, and child() is invoked as part of this thread, with NULL augument. conditional wait is used to wait for child to finish executing, and gives it the lock 'm'. child locks 'm', then prints [child\n] and signals to parent to continue. Then [done\n] is printed

-0.5

output:

```
child
done
```

what if child gets executed immediately?

1 5. **When the below code runs, and result is printed, what value will be printed? Justify the reason for your answer**

```
1
2  void mythread(void *arg) {
3      int result = 0; //local variable
4      result = result + 200;
5      printf("result %d\n", result);
6  }
7  int main(){
8      thread_t p1, p2;
9      thread_create(&p1, mythread, NULL);
10     thread_create(&p2, mythread, NULL);
11     thread_join(p1);
12     thread_join(p2);
13     return 0;
14 }
```

Solution: ~~No locks or synchronization measures have been used, so there are 2 possibilities:~~

~~1.~~ ~~200~~ result 200: This occurs iff ~~one of the threads~~ finishes entirely before the other, without ~~accessing~~

-1

Note: result is not a shared variable, it is a local variable. So, irrespective of the order in ~~which~~ the threads execute, they don't clash, and final output is:

```
result 200
result 200
```

Note: threads can't access each other's execution space

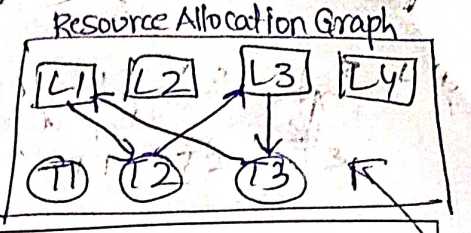
2 6. *One way to avoid deadlock is to schedule threads carefully. Assume the following characteristics of threads T1, T2, and T3:*

- *T1 (at some point) acquires and releases locks L1, L2*
- *T2 (at some point) acquires and releases locks L1, L3*
- *T3 (at some point) acquires and releases locks L3, L1, and L4*

For which schedules below is deadlock possible (Fill POSSIBLE/NOT POSSIBLE)? Justify the reason for your answer for each of the case given below

(T1: L1, L2) (T2: L4, L3) (T3: L3, L1, L4)

a) T1 runs to completion, then T2 to completion, then T3 runs
b) T1 and T2 run concurrently to completion, then T3 runs
c) T1, T2, and T3 run concurrently
d) T3 runs to completion, then T1 and T2 run concurrently



Solution:

a) NOT POSSIBLE. since T2 only occurs after T1 runs to completion and T3 also similarly occurs only after T2 runs to completion

b) NOT POSSIBLE, since cyclic wait does not occur, it is possible to schedule access of L1 for T1, T2.

c) POSSIBLE. say, T2 has L1, T3 has L3. If T2 requests L3 and T3 requests L1, deadlock is created.

d) NOT possible, this is same as (b), since T1, T2 can schedule access of L1, and make progress.

-2

Satisfies: hold-and-wait, cyclic wait, preemption

2 7. Can we use the test-and-set instruction to implement a lock on a multiprocessor environment? Explain why or why not.

Solution:

Test-and-set instruction was used to make locking and unlocking via setting a variable, to be atomic nature. It is a hardware solution, and would be implemented per-processor. So, NO, we can't directly use test-and-set instruction by itself to implement a lock in a multiprocessor environment.

If we ensure that test-and-set instructions made from separate cores are handled serially (with mutual exclusion), then it could be done.

-6