Q: Name at least two important differences between a POSIX thread

created through a pthread_create() call and a child process

created through a fork() call.

A:

1) Threads share data space and file descriptors with the other

threads and the parent process while forked process doesn't

Q: Given

```
int X[10];

myfun();

pthread_t tid;
```

Write the call to pthread_create(), which executes MyFun, and

accepts array X as an argument

A: pthread_create(&tid, NULL, myfun, X)

Q: What is the required function prototype for the function MyFun()?

A: void* myfun(void* X)

Q: True False A thread may share data with its main function or other threads.

A: TRUE

Q: True False A child runs the same code as its parent.

A: TRUE

Q: True False A child process may share a program counter with its parent.

A: FALSE

Q: True False A thread may share a File Descriptor with the main or other threads

A: TRUE

Q: How does a thread acquire RAM that is NOT shared with other threads in the task?

A: Define local variables and use them only in this thread scope.

Q: Assume that you have globally defined

```
struct Shared
{ char Name[10];
  int Value;
} S1, S2;
thread_t tid;
void * RtnValue;
void * FooBar(void * arg);   // For this problem, arg will point to an
```

instance of struct Shared

Show the correct way to:

1. Store "THREAD 1" in S1's Name field:
   - o ____
2. Pass S1 in the following call:
   - o pthread_create(&tid,NULL,FooBar,_______________________);
3. From within the function FooBar, print the Name that was passed inside the struct:
   - o printf("The Name Received = %s\n",

____;

4. store 2 times the thread id into the Value field of arg:

o ______________________ = ___________________________

5. return the struct via a thr_exit:

o pthread_exit(_______________________________);

6. Have main retrieve the returned value: Use the variable RtnValue since

you don't know which thread has exit-ed.

o pthread_join(_______, ________________);

7. Have main print which thread exit-ed and the integer returned:

o printf("Thread %d exited with value = %d\n",

______________________, _____________________);

A:

1. strcpy(S1.Name, "THREAD 1");

2. pthread_create(&tid,NULL,FooBar, void *S1);

3. printf("The Name Received = %s\n", ((struct Shared*)arg)->Name);

4. ((struct Shared*)arg)->Value = 2 * pthread_self();

5. pthread_exit(arg);

6-7. Even though pthread_join() can be used to wait for a specific thread;

there is no way to wait for "any" thread by using the Pthreads library in Linux.

Q: How can the code segment "if(i < y) cout << foobar(i,y);" be made to

behave as if it were atomic? Answer the question by recoding the segment.

pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;

```
  pthread_mutex_lock( &m );
  if( i < y )
    cout << foobar( i, y );
  pthread_mutex_unlock( &m );
```

Q: True or False

In the specification of pthread_join() in Linux, there is no way to wait for "any" thread (i.e. the call should specify a particular thread_id to join)

A: TRUE

Q: What is the effect of executing pthread_yield()?

A: pthread_yield(): stop executing the caller thread and yield the CPU to another thread

Q: What is a critical section i.e. what makes a section 'critical'?

A: In an asynchronous procedure of a computer program, a part that can not be executed simultaneously with an associated critical section of another asynchronous procedure.

In general, a code segment in which shared variables, shared file descriptors (shared resources) are accessed is considered a critical section.

Q: The word 'mutex' is short for ____.

A: mutual exclusion

Q: What is an atomic operation?

A: An operation that can not have it's execution suspended until

it is fully completed. Generally, it is a single operation that can

not be interrupted or divided into smaller operations.

Q: What does it mean to say that "pthread_mutex_lock(...) is a blocking call"?

A: If the mutex lock requested by this call is held by another

process, the called will be blocked until the lock is released by the

current owner (via executing a pthread_mutex_unlock() call)

Q: What would you expect to see (what instructions) surrounding the 'critical

section' code?

A: pthread_mutex_t mp=PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&mp);

<CRITICAL SECTION>

pthread_mutex_unlock(&mp);

Q: What does it mean to say that a library or a module is MT-Safe?

A: MT-Safe : MultiThread-Safe which means that the behavior of the function is

stable when two threads try to use it at the same time.In other

words, a library/module protects its global and static data with locks and can provide a reasonable amount of concurrency.

Q: What does it mean to say "rand() call is not (multi) thread-safe (MT-safe)?

This means that the behavior of the function rand() is not stable when two threads try to use it at the same time. To get around this problem, we use MT-safe interfaces (such as rand_r()). However, rand_r() doesn't exist on some multithreaded operating systems (e.g. LINUX), therefore, you need to execute such calls inside a Critical Section to make sure that only one thread can call the function at any one time.

Example:

```
pthread_mutex_t rand_mutex=PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock( &rand_mutex );
        int a = rand() % 5;
pthread_mutex_unlock( &rand_mutex );
```

Q: What is a spinlock?

A: When a process is in its critical section, any other process that tries to enter the same CS must loop continuously in the entry code of the CS until the first one gets out. The spinlock is the most common technique used for protecting a CS in Linux. It is easy to implement but has the disadvantage that locked-out threads continue to execute in a busy-waiting mode. Thus spinlocks are most effective in situations where the wait-time is expected to be very short.

```
spin_lock(&lock)
  /* Critical Section */
spin_unlock(&lock)
```

Q: What is an alternative to spinlocking?

A: Put the process in a wait queue, so it doesn't waste CPU cycles and allow it to sleep until the block is released.

Q: Explain under what circumstances a spinlock might be less efficient than the alternative.

A: In uniprocessor systems, processes can use CPU one at a time. If a process is "busy waiting" for a long time, then a spinlock might be less efficient.

Q: Explain under what circumstances a spinlock might be more efficient than the alternative.

A:

1. If you have more processor power than you need (e.g. in multiprocessor systems). Spinlock do not require context switch when a process must wait on a lock, and therefore it is more efficient.

2. when the locks are expected to be held for a short time, a spinlock

might be more efficient (saves the context switch time, therefore, preferred).

Q: In a uniprocessor environment, threads waiting for a lock

always sleep rather than spin. Why is this true (and reasonable)?

A: In a uniprocessor system, only one thread can run at a time. If a thread

is spinning for a lock, it is doing nothing but wasting CPU cycles while

waiting for the lock to become available. Instead, it is better that they

sleep (in the blocked queue) while waiting so that others can use the CPU.

When the lock becomes available, OS will wake them up.

Q: True or False?

Even if mutual exclusion is not enforced on a critical section,

results of multiple execution is deterministic (i.e. it produces

the same results each time it runs).

A: False

Q: True or False?

If mutual exclusion is not enforced in accessing a critical section,

a deadlock is guaranteed to occur.

A: False

Q: True or False?

An acceptable solution for implementing mutual exclusion is to let the users disable interrupts before entering a critical section and enable them after leaving the critical section.

A: False

Q: Give two reasons why we should not give the users the power to disable/enable interrupts in a multiprogrammed computer system.

A: (i) they may (unintentionally) forget to enable them which means all (hardware & software) interrupts will be ignored.

(ii) they may intentionally abuse this power and let their processes dominate the system resources.

Q: True or False?

One of the solutions proposed for handling the mutual exclusion problem relies on the knowledge of relative speeds of processes/processors.

A: False. No solution should rely on such assumptions

Q: What is the difference between starvation and deadlock?

A:

STARVATION: A condition in which a process is indefinitely delayed because other processes are always given preference.

DEADLOCK: An impasse that occurs when multiple processes are waiting for the availability of a resource that will not become available

because it is being held by another process that is in a similar state.

Q: What is the difference between pthread_mutex_lock and pthread_mutex_trylock?

A: pthread_mutex_lock is a blocking call. If we try to lock a mutex that is already locked by some other thread, pthread_mutex_lock blocks until the mutex is unlocked. But pthread_mutex_trylock is a nonblocking function that returns if the mutex is already locked.

Q: True or False?

If the shared resources are numbered 1 through N and a process can only ask for resources that are numbered higher than that of any resource that it currently holds, then deadlock can never happen.

A: TRUE