Participation: 5pts
Each question: 0.5 pts

1. The ____ multithreading model multiplexes many user-level threads to a smaller or equal number of kernel threads.
A) many-to-one model
B) one-to-one model
C) many-to-many model
D) many-to-some model

2. A _____ uses an existing thread — rather than creating a new one — to complete a task.
A) lightweight process
B) thread pool
C) scheduler activation
D) asynchronous procedure call

3. According to Amdahl's Law, what is the speedup gain for an application that is 60% parallel and we run it on a machine with 4 processing cores?
A) 1.82
B) .7
C) .55
D) 1.43

True/False

4. A traditional (or heavyweight) process has a single thread of control.

5. A thread is composed of a thread ID, program counter, register set, and heap.

6. Each thread has its own register set and stack.

7. The single benefit of a thread pool is to control the number of threads.

8. It is possible to create a thread library without any kernel-level support.

9. Virtually all contemporary operating systems support kernel threads.

10. It is possible to have concurrency without parallelism.

Extra Point (2 pts)

In Figure 1, below you can see an implementation of the producer-consumer problem (that we talked about in class) using two indexes *in* and *out*. There is an inefficiency problem with this implementation (resource wastage) what was the inefficiency?

```
item next_consumed;

while (true) {
     while (in == out)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;

     /* consume the item in next_consumed */
}
```

```
while (true) {
     /* produce an item in next_produced */

     while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
}
```

Figure 1: Producer-Consumer implementation with two shared variables *in* and *out*

In Figure 2 you can see an implementation of the producer and consumer problem using a *counter* variable which avoids the inefficiency explained in Figure 1 implementation.

In a multithreaded process where one thread is producing items and one other thread is consuming items what can go wrong in this implementation? (Hint: Think about data inconsistency)

```
while (true) {
     while (counter == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;

     /* consume the item in next_consumed */
}
```

```
while (true) {
     /* produce an item in next_produced */

     while (counter == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

Figure 2: New Producer-Consumer implementation with shared variable *counter*