

Quiz 1

1. If we have a line A in thread 1 and line B in thread 2. Which term refers to a situation when the 2 lines may be executed in any order? Fill in the blank: “Line A and B are executed _____”.
2. Standard definition of a mutex imposes 3 properties. List them:

3. This is incorrect implementation of mutex. Identify all problems. Be specific – specify line numbers to show sequence that leads to a problem(s).

Shared variable:

```
threadAisin = 0
threadBisin = 0
```

```
Thread A
while ( 1 ) do {
A1  while ( threadBisin == 1 ) do {}
A2  threadAisin = 1
A3  critical section code
A4  threadAisin = 0
A5  non critical code
}
```

```
Thread B
while ( 1 ) do {
B1  while ( threadAisin == 1 ) do {}
B2  threadBisin = 1
B3  critical section code
B4  threadBisin = 0
B5  non critical code
}
```

4. Consider the following design: **main** starts 100 identical threads each given its own input, only 6 threads are allowed to execute code at the same time. Do not modify **main**. Provide (psedo) code for thread. Make sure to state all shared variables. Your thread code should have a call to **DoWork(arg)** – assume this is where actual calculation is performed.

```
main() {  
    args[100]; // array of arguments  
    for i 0..99 {  
        start thread( args[i] );  
    }  
}
```

```
//shared vars
```

```
// thread  
thread( arg ) {
```

```
}
```

5. State Amdahl's Law.

Program speedup r is calculated as

$$r =$$

where p is the fraction of the time that is effected by speedup and s is the speedup. When

applied to a parallel algorithm p is the fraction of the time that is “parallized” and s is the number of cores available to the algorithm.

6. The code below attempts to implement a barrier: a point in a thread with the requirement that thread can only continue beyond it if all other threads have completed their *rendezvous* code. There is a problem – identify it.

```
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)

Thread:
rendezvous

mutex.wait
count = count +1
mutex.signal()

if ( count == n ) barrier.signal()

barrier.wait()

critical point
```

7. We want to create an efficient shuffling algorithm. Idea - perform random swaps of elements of the array using multiple threads.

Proposed solution 0 – discuss this solution. Assume read and write are atomic, i.e. if 2 threads asynchronously write-write, the data will not be garbage, one of the threads will write first and then the second will overwrite the result.

Shared variable:

```
int a[n] = {1,...,n}
```

```
for( int i=0; i<1000; ++i ) {  
    int i = rand()%n; // random index  
    int j = rand()%n; // random index  
    t = a[i]  
    a[i] = a[j]  
    a[j] = t  
}
```

Can array be {1,1,1,...,1} after this solution?

Proposed solution 1 – discuss this solution.

Shared variable:

```
int a[n] = {1,...,n}  
Semaphore sem[n] = { Semaphore(1),...,Semaphore(1)}
```

```
for( int i=0; i<1000; ++i ) {  
    int i = rand()%n; // random index  
    int j = rand()%n; // random index  
    sem[i].wait();  
    sem[j].wait();  
    t = a[i]  
    a[i] = a[j]  
    a[j] = t  
    sem[j].signal();  
    sem[i].signal();  
}
```

Proposed solution 2 – discuss this solution.

Shared variable:

```
int a[n] = {1,...,n}  
Semaphore sem = Semaphore(1)
```

```
for( int i=0; i<1000; ++i ) {  
    int i = rand()%n; // random index  
    int j = rand()%n; // random index  
    sem.wait();  
    t = a[i]  
    a[i] = a[j]  
    a[j] = t  
    sem.signal();  
}
```

Solve the problem efficiently and correctly.

8. Three kinds of threads share access to a singly-linked list: searchers, inserters and deleters. Searchers merely examine the list; hence they **can execute concurrently** with each other. Inserters add new items to the **end** of the list; **insertions must be mutually exclusive** to preclude two inserters from inserting new items at about the same time. However, one insert can proceed in parallel with any number of searches. Finally, deleters remove items from **anywhere** in the list. At most one deleter process can access the list at a time, and **deletion must also be mutually exclusive with searches and insertions**.

To summarize:

- searchers can execute concurrently with other searches.
- insertion must be mutually exclusive with other insertions
- deleters must be mutually exclusive other deletions as well as with searches and insertions

Shared variable:

```
insertMutex = Semaphore(1)
noSearcher = Semaphore(1)
noInserter = Semaphore(1)
searchSwitch = Lightswitch() // to be used with noSearcher mutex
insertSwitch = Lightswitch() // to be used with noInserter mutex
```

Reminder: Lightswitch object has a counter in it `insertSwitch.wait(noInserter)` increments the counter and keeps `noInserter` locked, `insertSwitch.signal(noInserter)` decrements the counter and if counter drops to 0 signals `noInserter` semaphores.

Write code for all 3 threads. All 3 threads can be written with 5 or less lines (including line "critical section" that executes the actual linked list code (you do not implement any of the linked list related code)).