Assignment 2.1 MESI-Protocol.

We reconsider the example from the lecture.

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 1; // A.1</td>
<td>while (b == 0) {} // B.1</td>
</tr>
<tr>
<td>b = 1; // A.2</td>
<td>assert(a == 1) ; // B.2</td>
</tr>
</tbody>
</table>

Draw a happened-before diagram for the execution B.1 A.1 A.2 B.1 B.2. Assume the underlying machine model to have caches and to be sequentially consistent using the MESI-protocol. Start with the cache state, where CPU B exclusively has a and b in its cache. Annotate each event of a cache line with the new state of the cache line.

Assignment 2.2 Happened-Before Diagram for Dekker.

Draw a happened-before diagram for the Dekker algorithm describing an interaction of two threads for a case where one of the threads succeeds to enter the critical section. Assume the underlying machine model to have caches and to be sequentially consistent using the MESI-protocol. In the beginning, all variables have a value of zero and are in shared state.

Assignment 2.3 Store Buffer and Invalidate Queues

Consider the following example program with Threads A and B executing a() and b(), respectively:

```c
struct G {
    int b=0;
    int a=0;
};
```

```c
void a() {
    G.b=1;
    int rega=G.a;
    // *
}
```

```c
void b() {
    G.a=1;
    int regb=G.b;
    // *
}
```

Given a machine model with a MESI-compliant cache and store buffers or invalidate queues. Specify an execution of the program such that reaching the respective program points * both the variables rega and regb contain value 0. Draw a happened-before diagram for this execution.
Assignment 2.4 Dekker Implementation.

1. Implement Dekker’s algorithm without memory barriers.

   To implement Posix threads in C, you might want to look for `pthread_create()` in `pthread.h` and compile with the `-pthread` compiler flag!

2. Demonstrate that out-of-order execution actually breaks Dekker’s algorithm when implemented without memory barriers.

   Hint: Clever instrumentation makes the difference!

3. Introduce memory barriers in your Dekker’s implementation; Test whether you can still observe broken behaviour.

   The statements to introduce memory barriers are compiler dependent.
   
   - Clang or GNU C++ as in MingW/Orwell-Dev-C++ or Linux systems use `__sync_synchronize(void),`
   - MacOS’ Xcode uses `OSMemoryBarrier(void) defined in libkern/OSAtomic.h`
   - MS’ Visual C++ uses `_mm_mfence(void) defined in intrin.h`

As an environment for threads, you may use Posix threads, e.g.

```c
// gcc -pthread dekker.c -o dekker
#include <pthread.h>// pthread_create, pthread_exit
#include <stdio.h> // printf
#include <stdlib.h> // exit

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    flag[0] = false;
    flag[1] = false;
    for (t = 0; t < NUM_THREADS; t++) {
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, dekker, (void *)t);
        if (rc) {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    /* last thing that main() should do */
    pthread_exit(NULL);
}
```