10. The following piece of code shows the implementation of the spin_lock synchronization function, which works in the following way: the thread execution it tries to acquire a lock a the memory address lock; if the lock is already acquired, it waits for a while (x time units) and then tries again; the process is repeated until the thread succeeds acquiring the lock.

```
void spin_lock (int *lock, int x) {
    int ret;
    do {
            ret = test_and_set (lock, 1);
        if (ret==1)
            pause (x); /* Pause the thread for x time units */
    } while (ret == 1);
}
```

a) Re-implement spin_lock function using load_linked/store_conditional
b) Write an optimized version for the test_and_set and load_liked/store_conditional versions with the objective of reducing coherence traffic.

```
int test_and_set (int *lock); // Set *lock to 1 and returns previous value of *lock
int load_linked (int *addr); // returns value of *addr
int store_conditional (int *addr, int value); // store value to addr
                                              // returns 0 if store fails, 1 otherwise
```

```
void spin_lock (int *lock, int x) {
    int ret;
    do {
            ret = test_and_set (lock, 1);
        if (ret==1)
            pause (x); /* Pause the thread for x time units */
    } while (ret == 1);
}
```

a) Re-implement `spin_lock` function using **`load_linked/store_conditional`**

```
int test_and_set (int *lock); // Set *lock to 1 and returns previous value of *lock
int load_linked (int *addr); // returns value of *addr
int store_conditional (int *addr, int value); // store value to addr
                                              // returns 0 if store fails, 1 otherwise
```

```
void spin_lock (int *lock, int x) {
    int …
    do {





    } while (…);
}
```

```
void spin_lock (int *lock, int x) {
    int ret;
    do {
            ret = test_and_set (lock, 1);
        if (ret==1)
            pause (x); /* Pause the thread for x time units */
    } while (ret == 1);
}
```

a) Re-implement `spin_lock` function using **`load_linked/store_conditional`**

```
int test_and_set (int *lock); // Set *lock to 1 and returns previous value of *lock
int load_linked (int *addr); // returns value of *addr
int store_conditional (int *addr, int value); // store value to addr
                                              // returns 0 if store fails, 1 otherwise
```

```
void spin_lock (int *lock, int x) {
    int value;
    do {




    } while (…);
}
```

```
void spin_lock (int *lock, int x) {
    int ret;
    do {
            ret = test_and_set (lock, 1);
        if (ret==1)
            pause (x); /* Pause the thread for x time units */
    } while (ret == 1);
}
```

a) Re-implement `spin_lock` function using **`load_linked/store_conditional`**

```
int test_and_set (int *lock); // Set *lock to 1 and returns previous value of *lock
int load_linked (int *addr); // returns value of *addr
int store_conditional (int *addr, int value); // store value to addr
                                              // returns 0 if store fails, 1 otherwise
```

```
void spin_lock (int *lock, int x) {
    int value;
    do {
        value = load_linked (lock);




    } while (…);
}
```

```
void spin_lock (int *lock, int x) {
    int ret;
    do {
            ret = test_and_set (lock, 1);
        if (ret==1)
            pause (x); /* Pause the thread for x time units */
    } while (ret == 1);
}
```

a) Re-implement `spin_lock` function using **`load_linked/store_conditional`**

```
int test_and_set (int *lock); // Set *lock to 1 and returns previous value of *lock
int load_linked (int *addr); // returns value of *addr
int store_conditional (int *addr, int value); // store value to addr
                                              // returns 0 if store fails, 1 otherwise
```

```
void spin_lock (int *lock, int x) {
    int value, ret;
    do {
        value = load_linked (lock);
        ret = store_conditional (lock, 1);


    } while (…);
}
```

```
void spin_lock (int *lock, int x) {
    int ret;
    do {
            ret = test_and_set (lock, 1);
        if (ret==1)
            pause (x); /* Pause the thread for x time units */
    } while (ret == 1);
}
```

a) Re-implement `spin_lock` function using **`load_linked/store_conditional`**

```
int test_and_set (int *lock); // Set *lock to 1 and returns previous value of *lock
int load_linked (int *addr); // returns value of *addr
int store_conditional (int *addr, int value); // store value to addr
                                              // returns 0 if store fails, 1 otherwise
```

```
void spin_lock (int *lock, int x) {
    int value, ret;
    do {
        value = load_linked (lock);
        ret = store_conditional (lock, 1);


    } while (…);
}
```

```
void spin_lock (int *lock, int x) {
    int ret;
    do {
            ret = test_and_set (lock, 1);
        if (ret==1)
            pause (x); /* Pause the thread for x time units */
    } while (ret == 1);
}
```

a) Re-implement `spin_lock` function using **`load_linked/store_conditional`**

```
int test_and_set (int *lock); // Set *lock to 1 and returns previous value of *lock
int load_linked (int *addr); // returns value of *addr
int store_conditional (int *addr, int value); // store value to addr
                                              // returns 0 if store fails, 1 otherwise
```

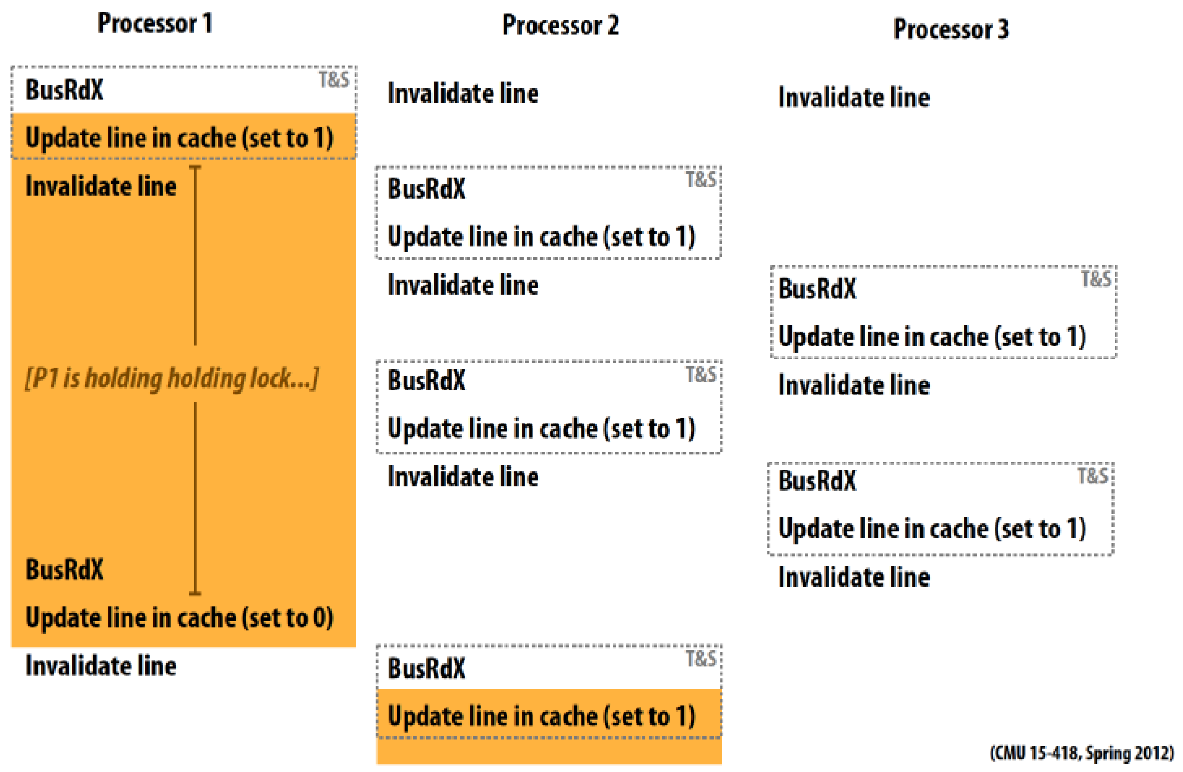
```
void spin_lock (int *lock, int x) {
    int value, ret;
    do {
        value = load_linked (lock);
        ret = store_conditional (lock, 1);
        if (value == 1 || ret == 0)
            pause (x);
    } while (value ==1 || ret == 0);
}
```

```
void spin_lock (int *lock, int x) {
    int ret;
    do {
            ret = test_and_set (lock, 1);
        if (ret==1)
            pause (x); /* Pause the thread for x time units */
    } while (ret == 1);
}
```

b) Optimized version for the `test_and_set` version with the objective of reducing coherence traffic.

```
void spin_lock (int *lock, int x) {
    int ret;
    do {
            ret = test_and_set (lock, 1);
        if (ret==1)
            pause (x); /* Pause the thread for x time units */
    } while (ret == 1);
}
```

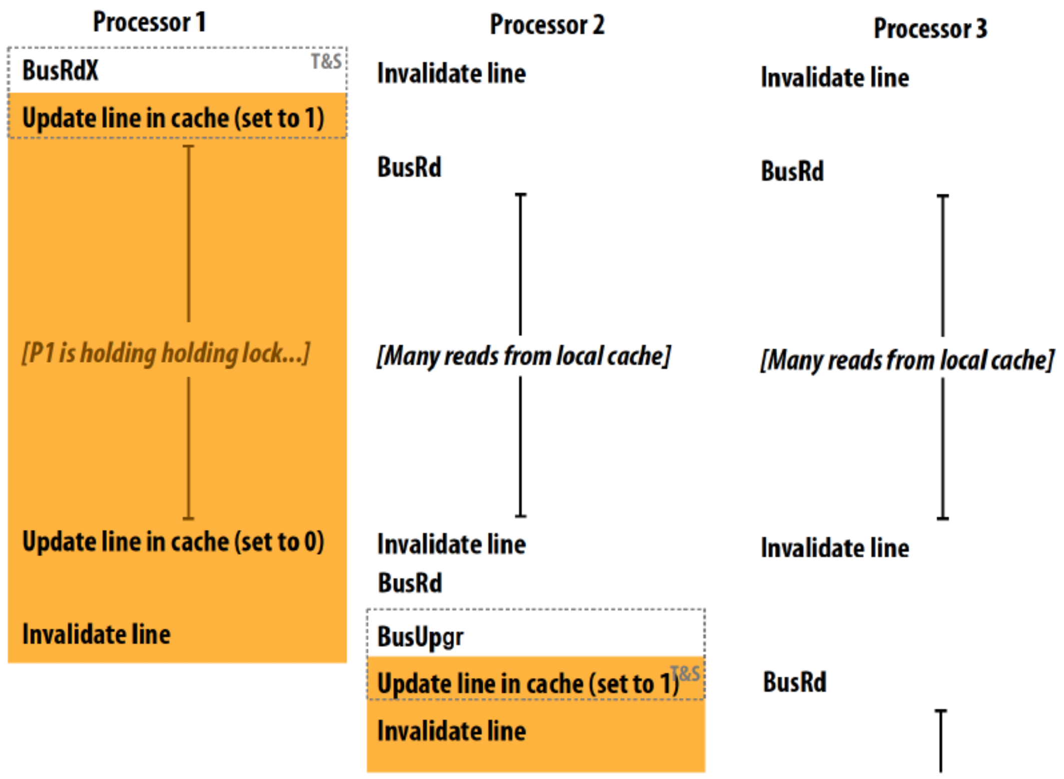
b) Optimized version for the `test_and_set` version with the objective of reducing coherence traffic.


Processor 1
Processor 2
Processor 3
BusRdX
T&S
Update line in cache (set to 1)
Invalidate line
[P1 is holding holding lock...]
BusRdX
Update line in cache (set to 0)
Invalidate line
Invalidate line
BusRdX
T&S
Update line in cache (set to 1)
Invalidate line
BusRdX
T&S
Update line in cache (set to 1)
Invalidate line
BusRdX
T&S
Update line in cache (set to 1)
Invalidate line
BusRdX
T&S
Update line in cache (set to 1)
Invalidate line
BusRdX
T&S
Update line in cache (set to 1)
Invalidate line
(CMU 15-418, Spring 2012)

For every *test-and-set* there are bus transactions invalidating any other ***lock*** copy in others processors caches …

```
void spin_lock (int *lock, int x) {
    int ret;
    do {
            ret = test_and_set (lock, 1);
        if (ret==1)
            pause (x); /* Pause the thread for x time units */
    } while (ret == 1);
}
```

b) Optimized version for the `test_and_set` version with the objective of reducing coherence traffic.



Processor 1
Processor 2
Processor 3
BusRdX
T&S
Update line in cache (set to 1)
Invalidate line
Invalidate line
BusRd
BusRd
[P1 is holding holding lock...]
[Many reads from local cache]
[Many reads from local cache]
Update line in cache (set to 0)
Invalidate line
Invalidate line
BusRd
Invalidate line
BusUpgr
Update line in cache (set to 1)
T&S
BusRd
Invalidate line

test-test-and-set:

Invalidation occurs only when potentially any processor can acquire the lock

```
void spin_lock (int *lock, int x) {
    int ret;
    do {
            ret = test_and_set (lock, 1);
        if (ret==1)
            pause (x); /* Pause the thread for x time units */
    } while (ret == 1);
}
```

b) Optimized version for the `load_linked / store_conditional` version with the objective of reducing coherence traffic.

```
void spin_lock (int *lock, int x) {
    int value, ret;
    do {
        /* test */
        while (*lock==1) pause(x);

        /* test-and-set */
        value = load_linked (lock);
        ret = store_conditional (lock, 1);
        if (value == 1 || ret == 0)
            pause (x);
    } while (value ==1 || ret == 0);
}
```

```
void spin_lock (int *lock, int x) {
    int ret;
    do {
            ret = test_and_set (lock, 1);
        if (ret==1)
            pause (x); /* Pause the thread for x time units */
    } while (ret == 1);
}
```

b) Optimized version for the `load_linked / store_conditional` version with the objective of reducing coherence traffic.

```
void spin_lock (int *lock, int x) {
    int value, ret;
    do {
        /* test */
        while (load_linked(lock)==1) pause(x);

        /* test-and-set */
        // value = load_linked (lock);
        ret = store_conditional (lock, 1);
        if (ret == 0)
            pause (x);
    } while (ret == 0);
}
```

```
void spin_lock (int *lock, int x) {
    int ret;
    do {
            ret = test_and_set (lock, 1);
        if (ret==1)
            pause (x); /* Pause the thread for x time units */
    } while (ret == 1);
}
```

b) Optimized version for the `test_and_set` version with the objective of reducing coherence traffic.

```
void spin_lock (int *lock, int x) {
    int ret;
    do {
        /* test */
        while (*lock == 1)
            pause (x);

        /* test-and-set */
            ret = test_and_set (lock, 1);
        if (ret==1)
            pause (x);
    } while (ret == 1);
}
```