9. The following piece of code shows the implementation of a barrier synchronisation, for a UMA multiprocessor system with MSI coherency, making use of the atomic instruction t&s:

```
lock:   t&s r2, barr.lock              // acquire lock
        bnez r2, lock
        if (barr.counter == 0)
            barr.flag = 0              // reset flag if first
        mycount = barr.counter++;
        if (mycount == P) {            // last to arrive?
            barr.counter =0            // reset for next barrier
            barr.flag = 1              // release waiting processors
        } else
            while (barr.flag == 0);    // busy wait for release
        barr.lock = 0                  // release lock
```

(a) Identify a concurrency problem that exists in this code and propose a solution to solve it.
(b) Based on the solution proposed for the concurrency problem identified in the first question, implement a new version that reduces the synchronization overhead to acquire lock.

9. The following piece of code shows the implementation of a barrier synchronisation, for a UMA multiprocessor system with MSI coherency, making use of the atomic instruction t&s:

```
lock:   t&s r2, barr.lock              // acquire lock
        bnez r2, lock
        if (barr.counter == 0)
            barr.flag = 0              // reset flag if first
        mycount = barr.counter++;
        if (mycount == P) {            // last to arrive?
            barr.counter =0            // reset for next barrier
            barr.flag = 1              // release waiting processors
        } else
            while (barr.flag == 0);    // busy wait for release
        barr.lock = 0                  // release lock
```

(a) Identify a concurrency problem that exists in this code and propose a solution to solve it.

When is lock released? … are you sure that threads will release it?

9. The following piece of code shows the implementation of a barrier synchronisation, for a UMA multiprocessor system with MSI coherency, making use of the atomic instruction t&s:

```
lock:   t&s r2, barr.lock              // acquire lock
        bnez r2, lock
        if (barr.counter == 0)
            barr.flag = 0              // reset flag if first
        mycount = barr.counter++;
        if (mycount == P) {            // last to arrive?
            barr.counter =0            // reset for next barrier
            barr.flag = 1              // release waiting processors
        } else
            while (barr.flag == 0);    // busy wait for release (WHEN???)
        barr.lock = 0                  // release lock
```

(a) Identify a concurrency problem that exists in this code and propose a solution to solve it.

When is lock released? … are you sure that threads will release it?

9. The following piece of code shows the implementation of a barrier synchronisation, for a UMA multiprocessor system with MSI coherency, making use of the atomic instruction t&s:

```
lock:   t&s r2, barr.lock              // acquire lock
        bnez r2, lock
        if (barr.counter == 0)
            barr.flag = 0              // reset flag if first
        mycount = barr.counter++;
        if (mycount == P) {            // last to arrive?
            barr.counter =0            // reset for next barrier
            barr.flag = 1              // release waiting processors
        } else
            while (barr.flag == 0);    // busy wait for release (WHEN???)
        barr.lock = 0                  // release lock
```

(a) Identify a concurrency problem that exists in this code and propose a solution to solve it.

When is lock released? … are you sure that threads will release it?
When should lock be released?

9. The following piece of code shows the implementation of a barrier synchronisation, for a UMA multiprocessor system with MSI coherency, making use of the atomic instruction t&s:

```
lock:   t&s r2, barr.lock              // acquire lock
        bnez r2, lock
        if (barr.counter == 0)
            barr.flag = 0              // reset flag if first
        mycount = barr.counter++;
        barr.lock = 0                  // release lock
        if (mycount == P) {            // last to arrive?
            barr.counter =0            // reset for next barrier
            barr.flag = 1              // release waiting processors
        } else
            while (barr.flag == 0);    // busy wait for release
```

(a) Identify a concurrency problem that exists in this code and propose a solution to solve it.

When should lock be released? Once barr.counter has been updated we CAN and SHOULD release lock!!! otherwise we will be not unlock and anybody else will increase lock... to be free!!!

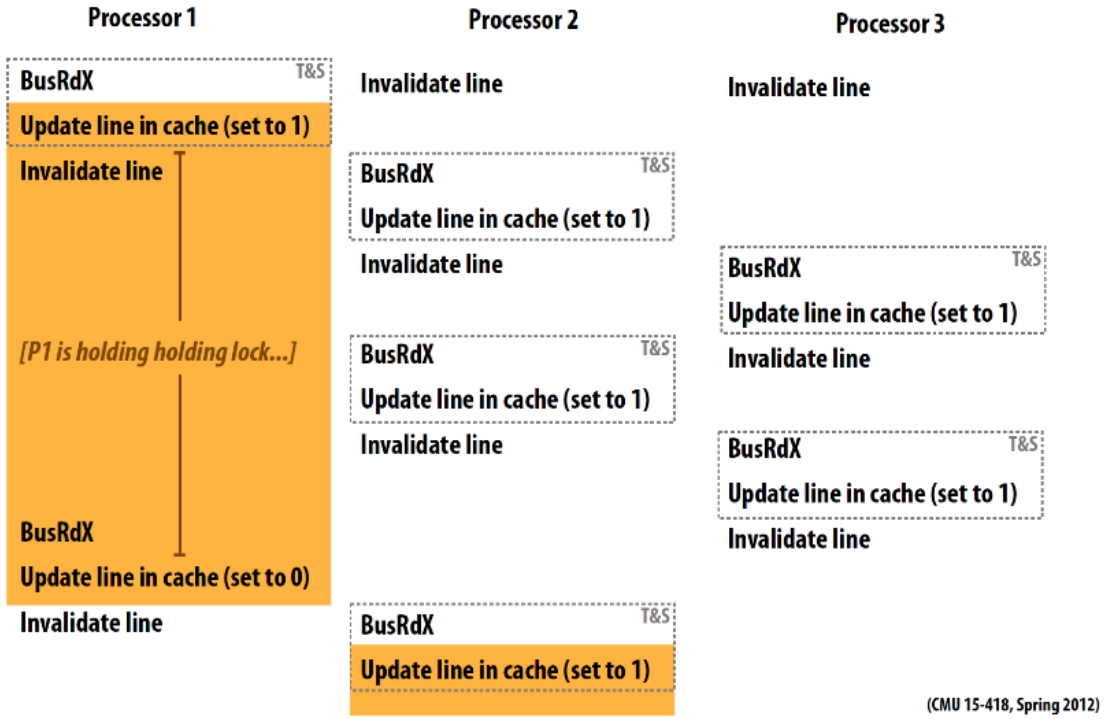
9. The following piece of code shows the implementation of a barrier synchronisation, for a UMA multiprocessor system with MSI coherency, making use of the atomic instruction t&s:

```
lock:   t&s r2, barr.lock           // acquire lock
        bnez r2, lock
        if (barr.counter == 0)
            barr.flag = 0           // reset flag if first
        mycount = barr.counter++;
        barr.lock = 0               // release lock
        if (mycount == P) {         // last to arrive?
            barr.counter =0         // reset for next barrier
            barr.flag = 1           // release waiting processors
        } else
            while (barr.flag == 0); // busy wait for release
```

(b) Based on the solution proposed for the concurrency problem identified in the first question, implement a new version that reduces the synchronization overhead to acquire lock.

9. The following piece of code shows the implementation of a barrier synchronisation, for a UMA multiprocessor system with MSI coherency, making use of the atomic instruction t&s:

```
lock:   t&s r2, barr.lock          // acquire lock
        bnez r2, lock
        if (barr.counter == 0)
            barr.flag = 0          // reset flag if first
        mycount = barr.counter++;
        barr.lock = 0              // release lock
        if (mycount == P) {        // last to arrive?
            barr.counter =0        // reset for next barrier
            barr.flag = 1          // release waiting proce
        } else
            while (barr.flag == 0); // busy wait for release
```



(b) Based on the solution proposed for the concurrency problem identified in the first question, implement a new version that reduces the synchronization overhead to acquire lock.

9. The following piece of code shows the implementation of a barrier synchronisation, for a UMA multiprocessor system with MSI coherency, making use of the atomic instruction t&s:

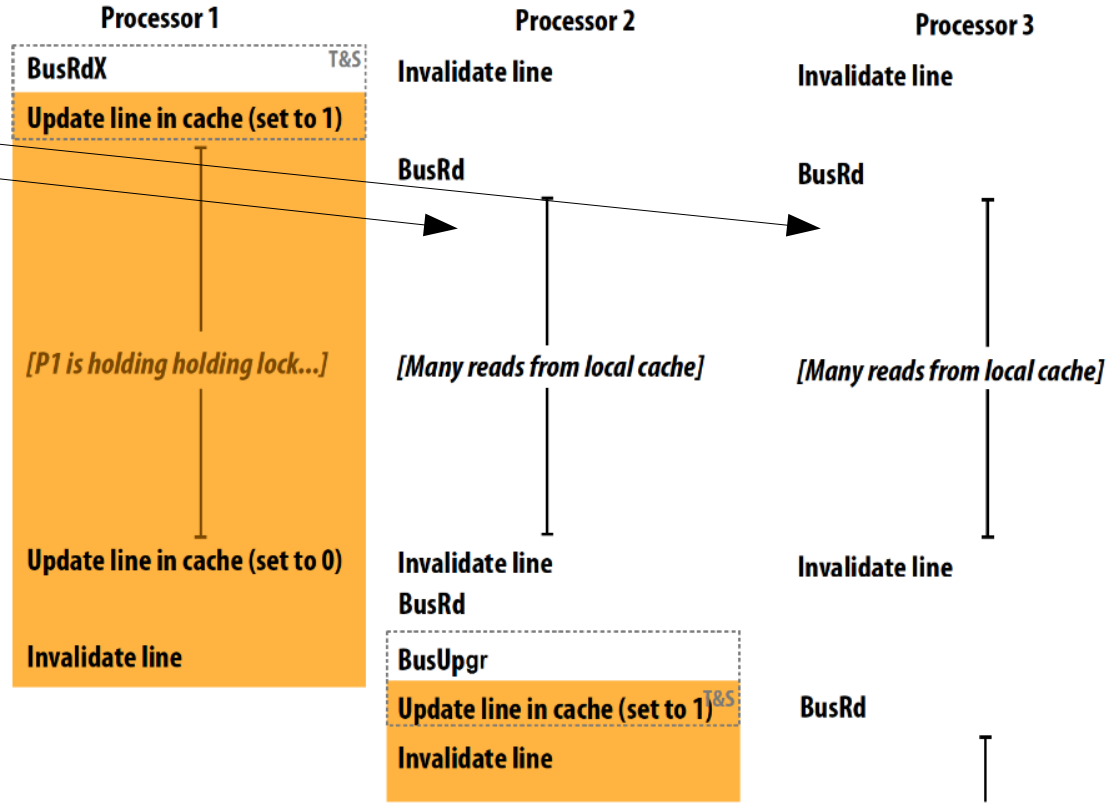
```
TEST        {   lock:   ld r2, lock                    // read lock
                        bnez r2, lock                  // is there any opportunity?
TEST &      {           t&s r2, barr.lock              // acquire lock
SET                     bnez r2, lock
                        if (barr.counter == 0)
                              barr.flag = 0            // reset flag if first
                        mycount = barr.counter++;
                        barr.lock = 0                  // release lock
                        if (mycount == P) {            // last to arrive?
                              barr.counter =0          // reset for next barrier
                              barr.flag = 1            // release waiting processors
                        } else
                              while (barr.flag == 0);  // busy wait for release
```

(b) Based on the solution proposed for the concurrency problem identified in the first question, implement a new version that reduces the synchronization overhead to acquire lock.

Using test-test-and-set technique we can reduce the amount of coherence cache protocol overhead.
Version using t&s

9. The following piece of code shows the implementation of a barrier synchronisation, for a UMA multiprocessor system with MSI coherency, making use of the atomic instruction t&s:

```
lock:   ld r2, lock                 // read lock
        bnez r2, lock               // is it free?
        t&s r2, barr.lock           // acquire lock
        bnez r2, lock
        if (barr.counter == 0)
              barr.flag = 0         // reset flag if first
        mycount = barr.counter++;
        barr.lock = 0               // release lock
        if (mycount == P) {         // last to arrive?
              barr.counter =0       // reset for next barrier
              barr.flag = 1         // release waiting proc
        } else
              while (barr.flag == 0);       // busy wait for release
```



(b) Based on the solution proposed for the concurrency problem identified in the first question, implement a new version that reduces the synchronization overhead to acquire lock.

Using test-test-and-set technique we can reduce the amount of coherence cache protocol overhead.
Version using t&s

9. The following piece of code shows the implementation of a barrier synchronisation, for a UMA multiprocessor system with MSI coherency, making use of the atomic instruction t&s:

TEST { (TEST) & SET {

```
lock:   ll r2, barr.lock        // Load linked read lock
        bnez r2, lock           // Is there any opportunity?
        mov r2, #1              // Prepare r2 to set 1
        sc r2, barr.lock        // Try to store 1, if sc return 0...
        beqz r2, lock           // Repeat if anybody else did it before
if (barr.counter == 0)
        barr.flag = 0                   // reset flag if first
mycount = barr.counter++;
barr.lock = 0                           // release lock
if (mycount == P) {                     // last to arrive?
        barr.counter =0                 // reset for next barrier
        barr.flag = 1                   // release waiting processors
} else
        while (barr.flag == 0);         // busy wait for release
```

Second test (read) is not needed since sc will return 0 if between ll and sc occurred an store

(b) Based on the solution proposed for the concurrency problem identified in the first question, implement a new version that reduces the synchronization overhead to acquire lock.

Using test-test-and-set technique we can reduce the amount of coherence cache protocol overhead.
Version using load linked and store conditional