ⓘ Students have either already taken or started taking this quiz, so take care when editing it. If you change any quiz questions in a significant way, you might want to consider re-grading students' quizzes who took the old version of the quiz.

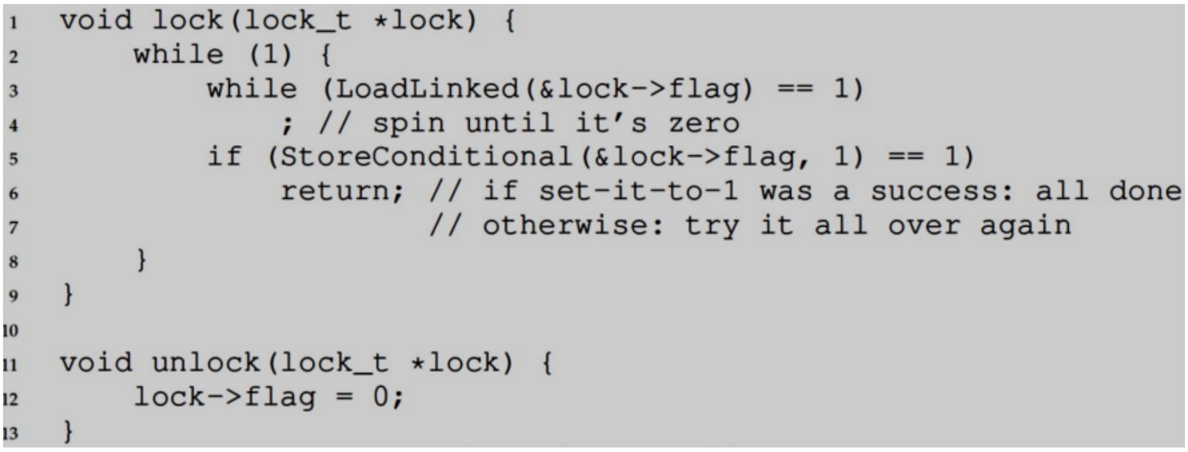
Points 8 ✓ Published ⋮

Details Questions

☑ Show question details

Question 1 pts

The following code is the implementation of a lock using load-linked/store-conditional instructions:



```
1   void lock(lock_t *lock) {
2       while (1) {
3           while (LoadLinked(&lock->flag) == 1)
4               ; // spin until it's zero
5           if (StoreConditional(&lock->flag, 1) == 1)
6               return; // if set-it-to-1 was a success: all done
7                       // otherwise: try it all over again
8       }
9   }
10
11  void unlock(lock_t *lock) {
12      lock->flag = 0;
13  }
```

Which of the following is most correct about this lock?

ıswer

○ "If" condition (on line 5) will be TRUE if no other thread called store-conditional on this lock since this thread's "while" loop (on line 3) ended

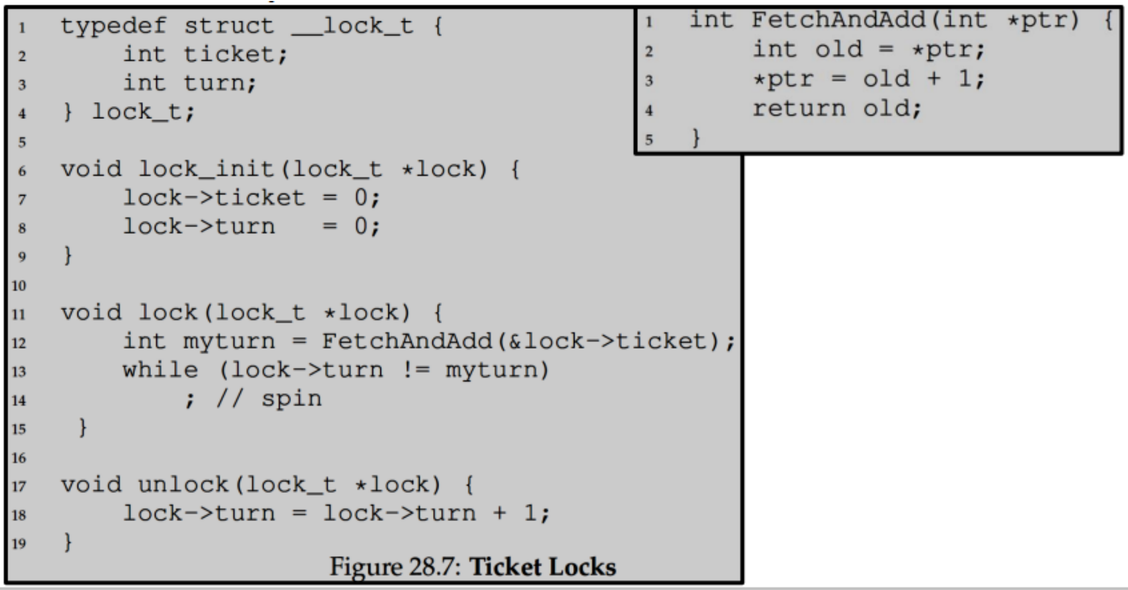
○ "If" condition (on line 5) will be FALSE if no other thread called store-conditional on this lock since this thread's "while" loop (on line 3) ended

○ "If" condition (on line 5) will be TRUE if no other thread called load-linked on this lock since this thread's "while" loop (on line 3) ended

○ "If" condition (on line 5) will be FALSE if no other thread called load-linked on this lock since this thread's "while" loop (on line 3) ended

Question 1 pts

The following is the implementation of a lock using fetch-and-add instructions:



```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn   = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
 }

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

Figure 28.7: **Ticket Locks**

Suppose the current value of ticket = 2 and the current value of turn = 1.

A thread calls lock(). Then:

ıswer

○ This thread will spin until another thread calls unlock()

○ This thread will spin until two other threads call unlock()

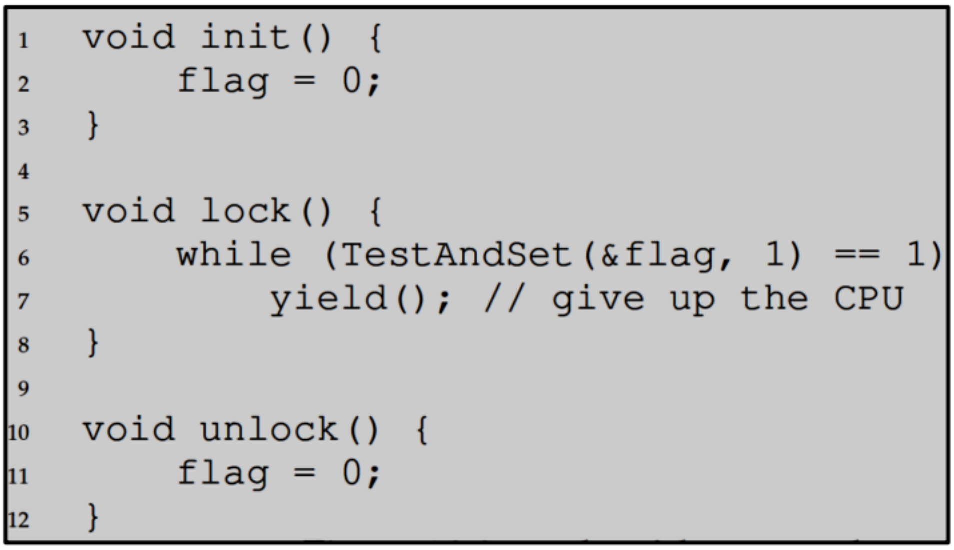
○ This thread will spin until three other threads call unlock()

○ This thread will not spin at all

Question

1 pts

Here is the implementation of a lock using yeild() function:



```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Suppose the lock was acquired by another thread when lock() is called by a thread. Then

ıswer

○ This thread keeps getting scheduled by the OS, but the thread keeps giving up the CPU

○ This thread goes to sleep is woken up only when the lock is released

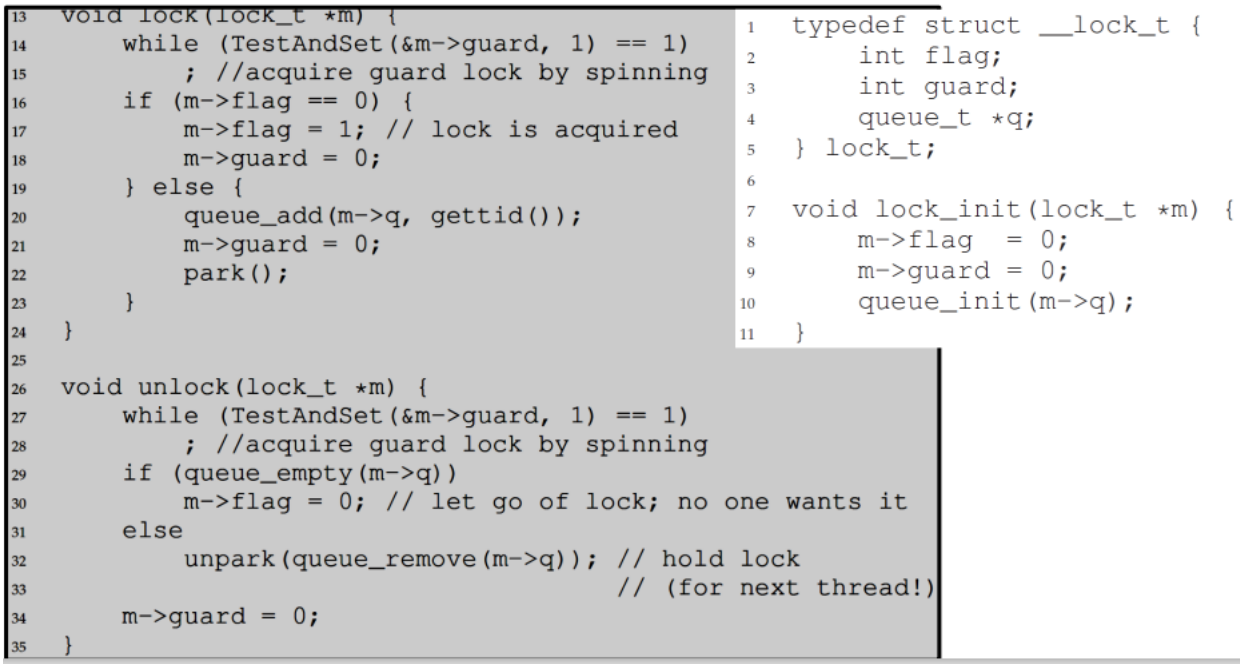
○ This thread keeps spinning until the lock is released by the other thread

○ This thread gets scheduled by the OS once, and it gives up the CPU. The thread is scheduled again only after lock is released by the other thread.

Question

1 pts

Here is the code for a queue-based implementation of a lock:



```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag  = 0;
    m->guard = 0;
    queue_init(m->q);
}
```

```
void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock
                                    // (for next thread!)
    m->guard = 0;
}
```

Which of the following is most accurate about this implementation?

ıswer

○ "guard" is a spin lock around actual the actual locking mechanisms of "flag" and "queue"

○ "flag" is a spin lock around actual the actual locking mechanisms of "guard" and "queue"

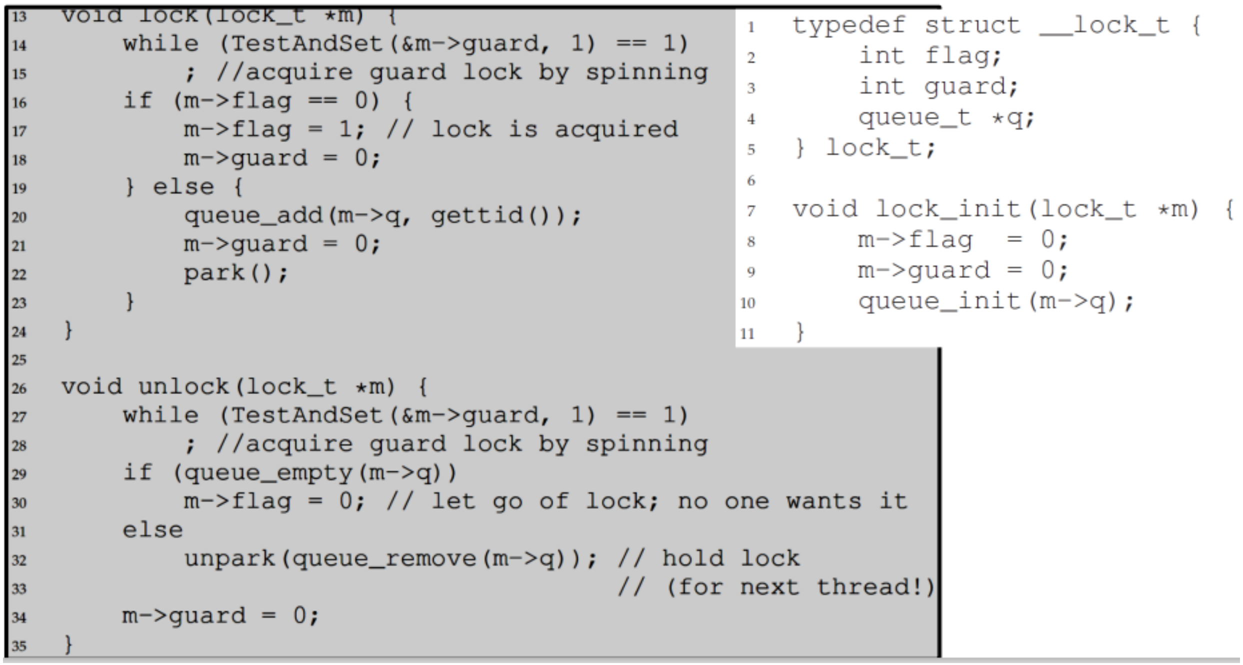
○ Both "flag" and "guard" serve as the main locking mechanism and there is no spinning

○ Both "flag" and "guard" are spin-locks

Question

1 pts

Here is the code for queue-based lock implementation:



```
void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock
                                    // (for next thread!)
    m->guard = 0;
}
```

```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag  = 0;
    m->guard = 0;
    queue_init(m->q);
}
```

Suppose a thread that had acquired the lock calls unlock(), while another thread was waiting in the queue.

Then which of the following is most accurate:

ıswer

○ "flag" was set to 1, and its value is unchanged as the waiting thread wakes up and acquires the lock.

○ "flag" was set to 0, and its value is unchanged as the waiting thread wakes up and acquires the lock.

○ "flag" was set to 1, and its value changes to 0 as the lock is released. It is then set to 1 after the waiting thread wakes up and acquires the lock.

○ "flag" was set to 0, and its value changes to 1 as the lock is released. It is then set to 0 after the waiting thread wakes up and acquires the lock.

Question

1 pts

Which of the following statements best describes concurrent approximate counters?

ıswer

○ Concurrent approximate counters trade off exact counting accuracy for improved performance in parallel and concurrent environments.

○ Concurrent approximate counters guarantee exact counting accuracy in multi-threaded environments.

○ Concurrent approximate counters are designed to provide precise counting results even when multiple threads access the counter simultaneously.

○ Concurrent approximate counters are exclusively used in single-threaded applications to avoid counting errors.

Question

1 pts

Which of the following best describes the purpose of the following C code snippet?

```
if (raise(SIGUSR1) != 0)
    perror("Failed to raise SIGUSR1");
```

ıswer

○ It causes a process to send the SIGUSR1 signal to itself

○ The code increases the value of SIGUSR1 in the current process.

○ The code is used to catch and handle the SIGUSR1 signal in the current process.

○ It creates a new process and sends the SIGUSR1 signal to it.

Question

1 pts

What is the purpose of the following C code?

```
#include <unistd.h>

int main(void) {
    alarm(10);
    for (;;) {
        // Infinite loop
    }
}
```

ıswer

○ The program sets an alarm to trigger after 10 seconds and enters an infinite loop.

○ The code sets an alarm for 10 seconds and then terminates the program.

○ It creates a background process that runs for 10 seconds and then terminates.

○ It waits for 10 seconds and then prints a message to the console.

+ New question + New question group Find questions

☐ Notify users this quiz has changed

Cancel Save