1) Recall that mutex can be used to update shared data structures as:

```
#include <pthread.h>

pthread_mutex_t my_mutex;
pthread_mutex_init(&my_mutex, 0);

/* All threads needing to access the critical section */
    pthread_mutex_lock(&my_mutex);
    /* critical section needing mutual exclusion */
    pthread_mutex_unlock(&my_mutex);

pthread_mutex_destroy(&my_mutex);
```

?

Alternative to the blocking `pthread_mutex_lock(&my_mutex)` are:

- `pthread_mutex_trylock(&my_mutex)` which attempts to lock the mutex, but will immediately return with an error code if it is busy/locked
- `pthread_mutex_timedlock(&my_mutex, &my_timespec)` which attempts to lock the mutex. If it is busy, then blocks for at most the specified time (#include <time.h>)

a) When might these alternatives be useful?

b) What might be a common programming error when using mutexs?

2) Monitors are a (slightly) higher-level mutual exclusion interface based on ADTs (abstract data types/objects):

- encapsulated state variables accessible only by its public interface (methods)
- monitor object has a mutex
- every method starts by locking and ends by unlocking the mutex so only one thread is access the state variables at any time

Java "synchronized" keyword can be used to achieve monitor-like behavior.

```
public class TicketVendor {
  private int seatsRemaining, cashOnHand;
  private static final int PRICE = 1000;

  public synchronized void sellTicket(){
    if(seatsRemaining > 0){
      dispenseTicket();
      seatsRemaining = seatsRemaining - 1;
      cashOnHand = cashOnHand + PRICE;
    } else
      displaySorrySoldOut();
  }

  public synchronized void audit(){
    // check seatsRemaining, cashOnHand
  }

  private void dispenseTicket(){
    // ...
  }

  private void displaySorrySoldOut(){
    // ...
  }

  public TicketVendor(){
    // ...
  }
}
```

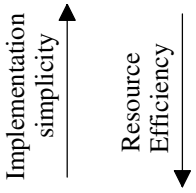
Figure 4.6: Outline of a monitor-style class in Java

a) Should the `dispenceTicket` and `displaySorrySoldOut` methods have be `synchronized`? (Explain your answer)

b) Why should the `seatsRemaining` and `cashOnHand` be private?

c) Why would it be a bad idea to add a `synchronized` method to interactively collect ticket sales information from from a patron?

3) Three (all PTHREAD_MUTEX_NORMAL - deadlock if thread holding mutex tries to lock it) mutex mechanisms are discussed:

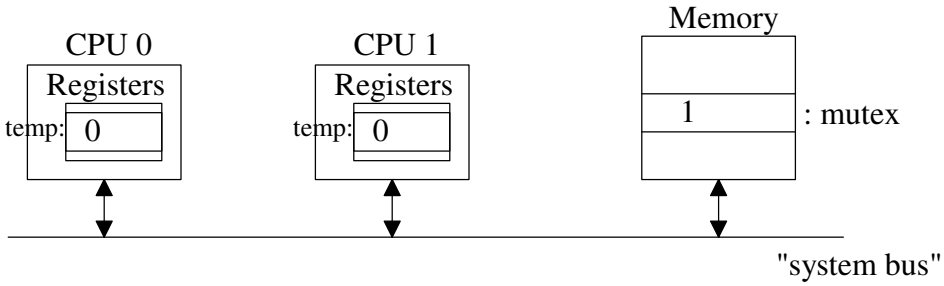


spinlock - simple busy-wait constantly performing "exchange operation"

cache-conscious spinlock - busy-wait, but mostly only reading lock value

queuing mutex - waiting in mutex queue and not busy-waiting

Modern architectures have an "*exchange operation*" which atomically (not interruptable) swaps contents of a register and a memory location.



a) What would be the results of the following **sequence** of operations?
CPU 0: exchange temp, mutex
CPU 1: exchange temp, mutex

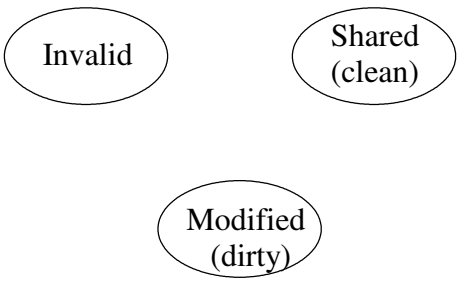
Spinlock: (mutex values: 0 ≡ locked, 1 ≡ unlocked)

Code to Lock mutex:	**Code to Unlock mutex:**
let register temp = 0 repeat atomic exchange temp, mutex until temp == 1	mutex = 1

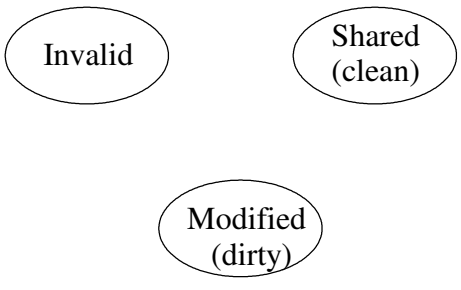
b) What disadvances does this simple spinlock implementation have?

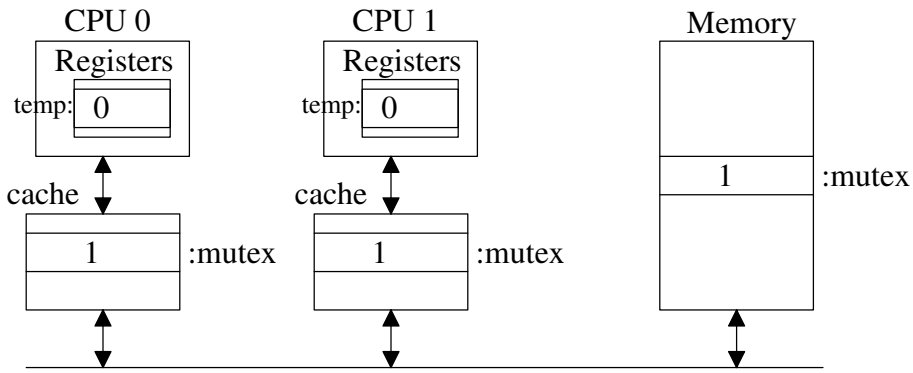
4) **Cache-Conscious Spinlock:** cache-coherence protocol typically "write-invalidates" (e.g. MESI)
Bus watching with write through / Snoopy caches - caches eavesdrop on the bus for other caches write requests. If the cache contains a block written by another cache, it take some action such as invalidating it's cache copy.
a) complete transition diagrams

cache-block state transistions due to local CPU requests



cache-block state transistions due to nonlocal CPUs' requests





Cache-Conscious Spinlock: (mutex values: 0 ≡ locked, 1 ≡ unlocked)

Code to Lock mutex:	**Code to Unlock mutex:**

let register temp = 0
 	
repeat
  atomic exchange temp, mutex
  if temp == 0 then
  while mutex == 0 do
  do nothing
  end while
  end if
until temp == 1
	mutex = 1

b) How does this help reduce the memory contention?

c) What disadvances does this cache-consious spinlock implementation have?

5) Queuing mutex - solves the busy-wait problem by having the thread notify the OS and moves it to a wait queue if the mutex is locked. What is the tradeoff for this?