CSL373/CSL633 Minor2 Exam
Operating Systems
Sem II, 2014-15

Answer all 6 questions Max. Marks: 34

Name:

GCL ID:

Marks table:
1.
2.
3.
4. a. b. c. d.
5.
6.

1. Give an example of a program where coarse-grained locking is likely to have better performance than fine-grained locking. [4]

Answer -

Check for this kind of concept -

Using coarse grained locks in a program usually decreases the locking overhead as compared to using fine grained locks. Fine grained locks can give better performance in the case where multiple threads are accessing different parts of a data structure such that their access is independent. Coarse grained locks can provide better performance in the case where the threads have to modify significant parts of the data such that fine grained locks will ensure the acquiring and releasing overhead of multiple locks.

Make sure they provided at least one example specifically code is not required an example in english would suffice.

2. Is it possible for the CPU utilization of your system to be 100% if all the threads in the system are involved in a deadlock? Explain. [4]

Answer - Yes, It is possible for system to have a 100% CPU utilization even in the case of deadlock. Consider the case when the threads involved in the system use **spin locks,** now if the threads are even in a deadlock scenario they will constantly check for reacquiring the lock whenever they get a chance to run hence consuming the cpu constantly which can result in 100% CPU utilization.

3. Can blocking locks be implemented for user-level threads? If no, why not? If yes, how? [6]

Answer - Yes, blocking locks can be implemented for user level threads, this can be done by using the help of scheduler such that when a thread acquired a lock we put the scheduler in a blocking queue. The thread will get a chance to run when the resource on which the queue is blocked is freed by one of the threads.

4. Consider the following function ‘foo()’ that reads and writes global variables ‘a’ and ‘b’

```
void foo(void) {
  a = a + 1;
  b = a + b;
}
```

a. If two threads execute foo() concurrently, what are all the possible final values of ‘a’ and ‘b’ (in terms of the initial values a0 and b0)? [3.5]. Warning: this may be a lengthy question; attempt in the end.

HINT: Can you use the fact that addition is commutative and associative, to reduce the number of cases? What can the minimum final value of ‘a’, ‘b’? What can be the maximum final value of ‘a’, ‘b’?

Result can be obtained by various interleavings of following code:

Read a
Write a <- a + 1
Read a
Read b
Write b <- a + b

The possibilities are: (a0+1, a0+2) X (b0+a0+1, b0+a0+2, b0+2a0+1, b0+2a0+2, b0+2a0+3, b0+2a0+4)

Now consider the following function ‘foo_coarse()’ with coarse-grained locking using a global lock ‘lg’. Assume that ‘lg’ has been properly initialized.

```
void foo_coarse(void) {
    acquire(&lg);
    a = a + 1;
    b = a + b;
    release(&lg);
}
```

b. If two threads execute foo_coarse() concurrently, what are all the possible final values of ‘a’ and ‘b’ (in terms of the initial values a0 and b0)? [0.5]

a will be: a0+2
b will be: b0+2a0+3

Marks - 0.5 for correct else 0.

Now consider the following function ‘foo_fine()’ with fine-grained locking using two global locks ‘la’ and ‘lb’. Assume that ‘la’ and ‘lb’ have been initialized properly.

```
void foo_fine(void) {
    acquire(&la);
    a = a + 1;
    release(&la);
    acquire(&lb);
    b = a + b;
    release(&lb);
}
```

c. If two threads execute foo_fine() concurrently, what are all the possible final values of ‘a’ and ‘b’ (in terms of the initial values a0 and b0)? [1.5]

a will be: a0+2
b could be: b0+2a0+3, b0+2a0+4

Marks - 0.5 for a and 0.5,0.5 for b.

Now consider the following function ‘foo_fine2()’ with fine-grained locking using two global locks ‘la’ and ‘lb’. Assume that ‘la’ and ‘lb’ have been initialized properly.

```
void foo_fine(void) {
    acquire(&la);
    a = a + 1;
    acquire(&lb);
    b = a + b;
    release(&la);
    release(&lb);
}
```

d. If two threads execute foo_fine2() concurrently, what are all the possible final values of ‘a’ and ‘b’ (in terms of the initial values a0 and b0)? [1.5]

a will be: a0+2
b will be: b0+2a0+3

Marks - 0.5 for a, 1 for b.

5. Consider the producer-consumer code below:

```
char queue[MAX];  //global
int head = 0, tail = 0; //global
struct cv not_full, not_empty;
struct lock qlock;

void produce(char data) {
  acquire(&qlock);
  while ((head + 1) % MAX  ==  tail) {
    wait(&not_full, &qlock);
  }
  queue[head] = data;
  head = (head + 1) % MAX;
  notify(&not_empty);   ← [CAN THIS STATEMENT BE MOVED AFTER 'release(&qlock)'?]
  release(&qlock);
}

char consume(void) {
  acquire(&qlock);
  while (tail == head) {
    wait(&not_empty, &qlock);
  }
  e = queue[tail];
  tail = (tail + 1) % MAX;
  notify(&not_full);
  release(&qlock);
```

```
  return e;
}
```

Can we move the notify(&not_empty) condition after ‘release(&qlock)’? If yes, why? If no, what is the problem that can occur? Is it okay to do this under certain conditions? What is that condition? [6]

Similarly, can we move notify(&not_full) condition after ‘release(&qlock)’? If yes, why? If no, what is the problem that can occur? Is it okay to do this under certain conditions? What is that condition? [2]

Please describe your answer clearly and precisely. For example, if you think that there could be a problem, describe the exact schedule that will cause that problem.

Answer -
Yes, it can be moved. Require some explanation.

6. We have discussed how scheduling/ordering of thread computation can be achieved using semaphores, using the following example:

semaphore_t Sa, Sb;//both Sa and Sb are initialized to 0

Thread 1:

…..
a = compute_something();
V(Sa);
….

Thread 2:
….
b = compute_something();
V(Sb);
….

Thread 3:
….
P(Sa);
P(Sb);
c = function(a, b);

….

For correctness we need to ensure that ‘c’ is computed only after ‘a’ and ‘b’ have been computed. Can we achieve this using condition variables and locks? If not, why not? If yes, write the code required to achieve these scheduling guarantees (using only condition variables and locks)? You are not allowed to use semaphores. [5]

Yes,

```
cv_t done;
lock_t lock;
bool a_computed = false;
bool b_computed = false;
```

Thread 1:

```
acquire(&lock);
a = compute_something();
a_computed = true;
notify(&done);
release(&lock);
```

Thread 2:

```
acquire(&lock);
b = compute_something();
b_computed = true;
notify(&done);
release(&lock);
```

Thread 3:

```
acquire(&lock);
while(!a_computed || !b_computed) {
```

```
    wait(&done, &lock);
}
c = function(a, b);
release(&lock);
```

Marks - 0 if used anything else from cv and locks.
5 for completely correct answer.
deduct 2 for minor bug.
deduct 4 for conceptual bug.