Index No:								[CS4532]

UNIVERSITY OF MORATUWA

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

B.Sc. Engineering
2014 Intake Semester 8 Examination

CS4532 CONCURRENT PROGRAMMING

Time allowed: 2 Hours December 2018

ADDITIONAL MATERIAL: *None*

INSTRUCTIONS TO CANDIDATES:

1. This paper consists of **four (4)** questions in **fourteen (14)** pages including Annex.
2. Answer **All** questions.
3. Answer the questions on the paper itself and within the given space.
4. For MCQ and True/False questions, select the most appropriate answer. No penalty for wrong answers.
5. The maximum attainable mark for each question is given in brackets.
6. This examination accounts for 50% of the module assessment.
7. This is a closed book examination.

   NB: It is an offence to be in possession of unauthorized material during the examination.
8. Only calculators approved by the Faculty of Engineering are permitted.
9. Assume reasonable values for any data not given in or with the examination paper. Clearly state such assumptions made on the script.
10. In case of any doubt as to the interpretation of the wording of a question, make suitable assumptions and clearly state them on the script.
11. This paper should be answered only in English.

Q1	Q2	Q3	Q4	Total

Question 1 (25 marks)

Tick **TRUE** or **FALSE**. Give **one sentence justification**. [2 × 4 marks]

		True	False
(i)	Any mutual exclusion solution that satisfies Liveness Property also satisfy Safety Property.		
(ii)	The maximum speedup achievable with n processes while multiplying $m \times n$ and $n \times l$ matrices is ml.		
(iv)	Checkpoint and rollback always guarantee recovery from a deadlock.		
(v)	While a Volatile variable in Java simplifies concurrent programming (as it maintains only one copy in memory), it has lower write performance compared to a static variable.		

One of the biggest hurdles in scaling existing multi-threaded applications to benefit from larger number of computing elements provided by the GPUs and MPI is the need to re-write part of the code to fit those platforms. One alternative is to use a distributed Operating System (OS) that provides a single system view across multiple nodes. That way, an application would see a significant increase in the available number of cores/CPUs and just needs to create more threads to benefit from them. Suppose there is an idea to develop such a distributed OS to run existing multi-threaded applications without any code changes or recompilations.

(vi) While such a distributed OS could increase the speedup, additional overheads will be introduced due to message passing and coordination among nodes. Suppose this overhead is proportional to the sequential fraction of the application. Let the proportional factor be λ.

Propose a simple modification to the Amdahl's Law to accommodate such additional overheads introduced by the distributed OS. Justify your answer. [3 marks]

Hint: Amdahl's law in the context of concurrent programming is given as $1/(1 - p + p/n)$, where n is the number of processors and p is the fraction that can be parallelized.

(vii) Suppose a weather forecasting program that typically takes 18 hours to run is planning to use the proposed distributed OS. 30% of the weather forecasting program cannot be parallelized in terms of their contribution to the execution time. The remaining code is parallelized.
How much time will be required to generate a weather forecast, if the proposed distrusted system uses 8 nodes each with 8 cores? Use the equation derived in section (vi). Assume $\lambda = 0.1$. [3 marks]

(viii) How practical is to the achieve the speedup you obtained in question (vii)? Briefly discuss. [4 marks]

(ix) OSs provide various concurrency primitives to make the implementations of mutual exclusion and synchronization easy (e.g., locks and semaphores). As existing applications may already use such primitives, it is important that the new distributed OS support those. Suppose you are assigned the task of developing a solution to emulate semaphores in the new OS. Outline a solution to provide the basic semaphore functionality (i.e., *up*() and *down*()) for the new distributed OS. [7 marks]

Question 2 (25 marks)

Consider the following 2 threads:

```
int v1 = 10;
int v2 = 20;
```

Thread 1	Thread 2
`v1 = v1 - 5;` `v2 = v2 + 5;`	`v2 = v2 - 2;` `v1 = v1 + 2;`

(i) Can the final values be $v1 = 5$, $v2 = 23$? Justify your answer. [4 marks]

(ii) Can the final values be $v1 = 5$, $v2 = 25$? Justify your answer. [4 marks]

(iii) Following is an implementation of the producer-consumer problem with an unbounded buffer. Modify the program to work with a bounded buffer of size 10. Add additional statements and use arrows to indicate where they need to be added to the existing code. [3 marks]

```
mutex = Semaphore(1)

items = Semaphore(0)
```

Producer:	Consumer:
`item = produceItem()`	`items.wait()`
`mutex.wait()`	`mutex.wait()`
`buffer.add(item)`	`item = buffer.get()`
`mutex.signal()`	`mutex.signal()`
`items.signal()`	`consumeItem(item)`

A monitor implementation for handling a warehouse is given below. Warehouse keeps track of quantities of each item in a map named *items*. Items can be added to the warehouse using the *addItems*() method. Items can be taken from the warehouse using the *getItems*() method. If the required quantity from the specified item is available, requested quantity will be removed from the *items* map. If not, caller is blocked until the request is fulfilled.

Conditional variable (i.e., *newItems*) has a *signalAll*() method, which unblocks all currently blocking threads on that variable. Although all waiting threads are unblocked, only one of them (chosen randomly) can acquire the monitor and run inside monitor methods at a given time.

```
Monitor warehouse {

      condition newItems;

      Map<string, int> items = new HashMap();

      function getItems(string item, int quantity) {
            boolean fulfilled = false;
            while (!fulfilled) {
                  int count = items.get(item);
                  if (count >= quantity) {
                        items.put(item, (count - quantity));
                        fulfilled = true;
                  } else {
                        newItems.wait();
                  }
            }
      }

      function addItems(string item, int quantity) {
                        int count = items.get(item);
            items.put(item, (count + quantity));
            newItems.signalAll();
      }
}
```

(iv) Tick **TRUE** or **FALSE**. Give **one sentence justification**. [2 × 4 marks]

		True	False
(a)	The proposed monitor implementation can work with any number of threads who may call *addItem* and *getItem* functions.		
(b)	The proposed monitor implementation is based on busy waiting.		
(c)	The proposed monitor algorithm satisfies mutual exclusion.		
(d)	The proposed monitor algorithm can cause a deadlock.		

Threads T1, T2, and T3 run the code given below. R1, R2, and R3 are non-preemptible resources that require mutually exclusive access. Each thread has run up to the point indicated by `current`.

Thread T1:	Thread T2:	Thread T3:

`Acquire R1`
`---------- current position`
`Acquire R3`
`Release R1`
`Release R3`
	`Acquire R2`
	`Acquire R3`
	`---------- current position`
	`Acquire R1`
	`Release R2`
	`Release R3`
	`Release R1`
		`Acquire R3`
		`---------- current position`
		`Acquire R2`
		`Release R3`
		`Release R2`

(v) Model above situation using a resource allocation graph. [3 marks]

(vi) Propose a method to prevent the deadlock in this code. [3 marks]

Question 3 (25 marks)

Assume that you must write a program to check spellings in large documents. Program should perform the following steps:

- Read step: Read documents from the hard disk
- Spell check step: Underline misspelled words
- Upload step: Upload the processed document to a file server over a network.

On average reading a 100 MB document takes 10 seconds. Underlining misspelled words of a 1 MB document fragment takes 1 second. Uploading a 100 MB document takes 15 seconds.

Both reading and uploading documents to/from the file server are I/O intensive. Spell checking is CPU intensive. Assume that operating system and other background task overheads on CPU cores are negligible and memory is unlimited. Further assume that no hardware optimizations are available in the CPU and a CPU core runs one instruction at a time.

(i) If we have to process a large number of documents, propose a solution to efficiently run this program by combining concurrency patterns as appropriate. [3 marks]

(ii) If all 3 steps are performed on a single 100 MB document, calculate the total time required to complete the program. Briefly show calculation steps. [3 marks]

(iii) Calculate the minimum time required to complete all 3 steps on a single 100 MB document using a 4-core processor? Show calculation steps. [3 marks]

(iv) What is the best speedup you can gain based on the concurrency pattern proposed in section (i) for 2 files, compared to section (ii)? Discuss. [3 marks]

Intel Optane is a non-volatile memory technology introduced to further extend the memory hierarchy by bridging the speed gap between RAM and Hard Disk. However, specialized motherboards are needed to connect Optane chips. Optane shows the highest speedup in gaming applications.

Suppose Intel licenses the technology to company *OX* to build the Optane chips while license to build motherboards is given to companies *MY* and *MZ*. 3 OEM vendors HP, Dell, and Asus are planning to sell gaming PCs based on Optane technology. However, as the technology is new production by *OX*, *MY*, and *MZ* is limited. Suppose Optane chips are released to OEMs as a batch of 100 chips while *MY* and *MZ* release motherboards as batch of 50 each. Each batch is released once in every 3 days. Given all components it takes another day to release a batch of 50 gaming PCs. Alternatively, 3 OEMs are rushing to introduce Optane-based gaming PCs as the customers may lose the interest if products are not available in the market.

(v) Using an example, explain a scenario where none of the 3 OEMs will be able to release a new batch of gaming PCs at least before 7 days. [3 marks]

(vi) Outline a semaphore-based solution that guarantees at least one of the 3 OEMs will be able to release a batch of gaming PCs after 4 days. [10 marks]

OX:		My/MZ:
HP:	Dell:	Asus:

Question 4 (25 marks)

Exponential function is given as:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \cdots$$

(i) "*Regardless of whether we use PThreads, OpenMP, CUDA, or MPI balancing the load among multiple threads/nodes is a difficult problem*." Do you agree or disagree with this statement? Justify. [3 marks]

(ii) Outline CUDA-based solution to estimate e (i.e., $x = 1$) using first one million numbers in the sequence. [11 marks]

(iii) Outline an MPI program (using pseudo code) to estimate e (i.e., $x = 1$) using first one million numbers in the sequence. Indicate MPI functions and key parameters (it is not essential to follow exact function signature). Result need to be in node 0. [11 marks]

Appendix – MPI Functions

```
. . .
#include <mpi.h>
. . .
int main(int argc, char* argv[]) {
   . . .
   /* No MPI calls before this */
   MPI_Init(&argc, &argv);
   . . .
   MPI_Finalize();
   /* No MPI calls after this */
   . . .
   return 0;
}
```

```
int MPI_Init(int *argc, char **argv)
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Finalize()
int MPI_Send (void *buf,int count, MPI_Datatype datatype, int dest, int
            tag, MPI_Comm comm)
int MPI_Recv (void *buf,int count, MPI_Datatype datatype, int source, int
            tag, MPI_Comm comm, MPI_Status *status)
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype
            datatype, MPI_Op op, int root, MPI_Comm comm)
int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void
            *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Allreduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype
            datatype, MPI_Op op, MPI_Comm comm)
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root,
            MPI_Comm comm)
int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void
            *recvbuf, int recvcnt, MPI_Datatype recvtype, int root,
            MPI_Comm comm)
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void
            *recvbuf, int recvcnt, MPI_Datatype recvtype, int root,
            MPI_Comm comm)
```

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

---------- END OF THE PAPER ----------