**Hello, all. I tried putting together some rough answers for the 2014 exam. I'd love some help with collab in this a3**

**s well. 2014 +1**

1. (a) Briefly explain the *critical section problem*, describe the four properties which are normally associated with a good solution, and explain the roles of the *entry protocol* and *exit protocol* (you do not need to discuss details of any specific solution). [*8 marks* ]

   (b) Consider the standard Bakery Algorithm (BA) and Spin Lock (SL) solutions to the critical section problem.

      i. *Without writing pseudocode*, explain the work undertaken by the BA in its entry and exit protocols. [*2 marks* ]

      ii. Contrast BA and SL solutions in terms of both the requirements they make of the system (for example, memory model, atomicity) and their adherence to the four properties of part (a). [*4 marks* ]

      iii. Briefly discuss the suitability of the BA as the basis of an implementation of locks. [*2 marks* ]

   (c) You are given access to an atomic machine instruction SFAA (Saturated-Fetch-And-Add), whose behaviour can be summarised by the following pseudocode:

```
int SFAA (int x) {
 <
   int temp = x;
   if (x<100) x++;
   return temp;
 >
}
```

   Write and explain pseudocode for an SL style solution to the critical section problem which exploits SFAA. Try to make your solution reasonably cache-efficient. [*9 marks* ]

---

1. (a)

4 properties associated with a good solution to a critical section problem:

- Mutual Exclusion
    - A maximum of one thread at a time is allowed to enter its critical section.
- Absence of livelocks and deadlocks
    - If two or more threads attempt to access a critical section, at least one succeeds.
- Absence of unnecessary delay
    - If there are no other threads currently in a critical section and there is a thread requiring access to enter a critical section, this thread is not prevented to enter its critical section.
- Eventual Entry (No starvation)
    - A thread attempting to enter a critical section will eventually succeed.

The first 3 are necessary for a good solution to a critical section problem, but the 4th one is desirable. This is because we may be making progress in other threads.

Entry Protocol:

An entry protocol should be able to block processes that are requesting to enter their critical section, but cannot due to other processes currently in their critical section. In other words, enforce mutual exclusion (single process in critical section in any given time).

Exit Protocol:

An exit protocol must be aware of when a process that is currently in its critical section completes its critical section, and make aware of waiting processes that it can enter its critical section.

1. (b) (i)

Bakery Algorithm Entry Protocol:

**The entry protocol calculates the threads turn by looking at other threads' turns and taking the current maximum thread turn value and incrementing by one. After calculating its own thread turn value, the protocol will loop until all other threads that are requesting access to their critical section have greater NON Zero turn values than itself.**

Bakery Algorithm Exit Protocol:

A thread turn value of 0 indicates that it is currently not requesting access to a critical section. When the thread exits its own critical section, it will set its turn value to 0 to indicate it is no longer requesting access to its critical section.

1. (b) (ii)

- Both BA and SL guarantee mutual exclusion, so only a maximum of a single thread may enter its critical section.
- Both BA and SL guarantee absence of livelocks, deadlocks and delays to entry of critical section.
- BA does not need atomic read and atomic write instructions, but can be
- implemented using them. On the other hand, SL **requires** atomic instructions to achieve mutual exclusion.
    - But BA requires a sequentially consistent (SC) memory model in both cases (with or without atomic instructions) in order to work, which is expensive due to the severe restrictions on the buffering and pipelining of 0memory accesses and is not commonly implemented in real life systems.
    - SL also requires a sequentially consistent (SC) memory model, although it may also be implemented in weaker models if memory fences are explicitly introduced by the programmer.
- BA guarantees eventual entry (no starvation) which is desirable in a critical section solution. SL does not guarantee this.
- Naive implementations of SL make poor use of cache coherency and can cause cache contention. A pragmatically better solution is known as Test-and-Test-and-Set (though it still uses Test-and-Set).

1. (b) (iii)

***Not sure of this solution, perhaps we are overthinking for 2 marks?***

Not sure if this is it but I was thinking it'd be that it supports at most N threads as we need to preallocate an array of size N for the number of threads in use?

BA needs atomic instructions which are expensive to implement and requires a

sequentially consistent memory model which is not common in the real world.

If SC was enforced on the other hand, BA could be used as the basis of a SL style system. A global TURNS[] array that is visible to all threads can be used to indicate which threads are requesting access to its critical section.

LOCK() will assign a turn value to the thread and update TURNS[] with its turn value.

UNLOCK() will set its turn value to 0 and update TURNS[] with its turn value, allowing the next thread to enter its critical section.

1. (c)

**Not sure what the significance of the SFAA instruction is**

```
int SFFA(int x){
 <
   int temp = x;
   if(x<100) x++;
   return temp;
 >
}

int lock = 99;
co[i = 0 to n-1] {

   await(SFFA(lock)<100);
   critical section...
   lock = lock-1;
   non-critical section...
}
```

The above pseudo-code provides an SL style solution for the critical section problem by exploiting SFFA. Since the entirety of SFFA's contents are considered one atomic instruction the write and return of the variable lock occur at the same time, which means whenever a process returns a value lower than 100 we can be certain that it has incremented the variable. This provides a natural lock on the system, as processes awaiting to enter the critical section must wait until a process has finished its run of the critical section and decremented the lock. The added benefit of this solution is that we can run multiple threads at the same time by decrementing the initial value of lock by

the total number of concurrent threads we want to allow.

-- Nicholas

Ashley

```
int l = 99;
while(l == 100  || SFAA(l) == 100); // spin
        // enter cs
        l = 99;
        // leave cs
```

Checks that l == 100 (i.e other process in cs) and will not check second condition if it is (hence cache efficient). When x = 99, checks second condition which will also return false and will enter CS. l is now = 100 so no other process will enter cs.

```
}
```

+1

2. (a) Consider the following attempt to implement a reusable counter barrier for `P>1` processes.

```
shared int count = 0;
co [myID = 0 to P-1] {
  while (something) {
     do some work;
     ## now the barrier
     <count = count + 1;>
     <await (count == P);>
     < if (myID == P-1) count = 0;>
  }
}
```

The implementation is broken.

i. Describe the behaviour this broken implementation would have in a typical execution. [*3 marks* ]

ii. Explain whether there are any circumstances in which, by luck, the barrier might appear to be executing correctly. [*2 marks* ]

(b) Both symmetric and dissemination barrier implementations satisfy barrier semantics as a result of a collection of more localized interactions.

i. Explain the main structural difference between the two implementations in the organization of these interactions, and why this makes one approach more general than the other. [*3 marks* ]

ii. How many stages are required to implement a dissemination barrier for 14 processes? [*2 marks* ]

(c) Sketch the pattern of local interactions used to implement a dissemination barrier for six processes (numbered from 0 to 5). Explain why this ensures that processes 0 and 5 are correctly synchronized. [*5 marks* ]

(d) Explain the connection between Java objects and the concurrency control concept of a *monitor*. Your answer should refer to the Java keyword `synchronized` and methods `wait()`, `notify()` and `notifyAll()`. [*5 marks* ]

(e) Describe, with the help of Java pseudocode, how you would use the operations introduced in part (d) to implement a re-usable counter barrier in Java. Your Java syntax does not have to be perfect, but the key ideas should be clear. [*5 marks* ]

2. (a) (i)

During normal execution, every thread will eventually hit the await condition and wait.

Once *count* is equal to *P*, one by one the threads will complete the await statement:

*< await (count == P) >*

If the thread with value *myID* == P-1 completes the *await* statement **before** other threads complete, there is a chance that the thread with *myID* == P-1 will reset *count* to 0 before the other threads complete the *await* statement. Any threads that are still at the await statement will remain in deadlock and eventually the entire implementation will be broken as *count* can never be equal to *P*, thus never exiting the barrier.

(ii)

If every thread excluding the thread with *myID* == P-1 executes the *await* barrier before the thread with *myID* == P-1 resets *count* in each loop iteration, the barrier will appear to be executing correctly as *count* can equal P in the next loop iteration.

**P-1 can't reset *count* before executing the *await* barrier. The excluding of myID == P-1 is unnecessary.**

(b) (i)

Symmetric Barrier:
Overall synchronization is achieved by implication by choosing set **pairwise** synchronisations, and synchronising in steps. Symmetric barriers only work if the number of synchronising threads is a power of 2.

Dissemination Barrier:

Overall synchronisation is achieved by implication by choosing different thread "partners" for synchronising with and being synchronised by in steps. Dissemination barriers can work with non power of 2 number of synchronising threads. The number of steps for a dissemination barrier to fully synchronise is $ceiling\left[log_2(p)\right]$ steps, where p is the number of processes.

As dissemination barriers do not require a power of 2 number of threads to be execute correctly, dissemination barriers are a more generic barrier solution than symmetric barriers.

(b) (ii)
For 14 processes using a dissemination barrier, there will be $ceiling\left[log_2(p)\right]$ stages.

$$ceiling\left[log_2(p)\right]$$
$$ceiling\left[log_2(14)\right]$$
$$ceiling[3.807]$$
$$4$$

There will be 4 stages for a 14 process dissemination barrier.

(c)

$ceiling[6]$ => $ceiling[2.584]$ => 3 stages

STAGE 1
p0 will set p1, and will be set by p5.
p1 will set p2, and will be set by p0.
p2 will set p3, and will be set by p1.
p3 will set p4, and will be set by p2.
p4 will set p5, and will be set by p3.
p5 will set p0, and will be set by p4.

STAGE 2
p0 will set p2, and will be set by p4.
p1 will set p3, and will be set by p5.
p2 will set p4, and will be set by p0.
p3 will set p5, and will be set by p1.
p4 will set p0, and will be set by p2.
p5 will set p1, and will be set by p3.

STAGE 3

p0 will set p4, and will be set by p2.

p1 will set p5, and will be set by p3.

p2 will set p0, and will be set by p4.

p3 will set p1, and will be set by p5.

p4 will set p2, and will be set by p0.

p5 will set p3, and will be set by p1.

p0 and p5 are synchronized due to:

- Stage 1 where p5 synchronises p0, and p0 synchronises with p1.
- Stage 3 where p1 synchronises p5.

Thus p5 has synchronised with p0 and p0 has implicitly synchronised with p5 through p1 which has explicitly synchronised with p5.

(d)

Any object in Java can act as a monitor by declaring one or more methods as **synchronized** or including any **synchronized** declared code.

Each object that is **synchronized** is associated with a single implicit lock. Entering any **synchronized** section of code is equivalent to obtaining the lock, and exiting the **synchronized** section of code is equivalent to releasing the lock.
Java employs Signal-And-Continue semantics, by allowing a single condition variable queue per monitor which can be interacted with using the following methods:

- *wait()* - wait on this condition variable until it is released.
- *notify()* - release this condition variable and notify a single waiting thread.
- *notifyAll()* - releases this condition variable and notify ALL waiting threads.

(e)

```
volatile int count = 0;
synchronized void func()
{
	while(something)
	{
		// do work
		// barrier
		count++;
		if(count == n)
		{
			count = 0;
			notifyAll();
		}
		else
		{
			wait();
		}
	}
}
```

Alt Solution:

```
// Global Sense Object
public class Sense()
{
	int counter = 0;
	int threshold = 0;
	boolean sense = false;

	public Sense(int t)
	{
		threshold = t;
```

```
	}

	public synchronous boolean getSense()
		return sense;

	public synchronous void updateCount()
	{
		counter++;
		if(counter == t)
		{
			counter = 0;
			sense = !sense;
		}
	}
}

// Spawn a whole load of these
public class Thing()
{
	while(something)
	{
		// Do some stuff
		Sense.updateCount();
		while( Sense.getSense != mySense );
		mySense = !mySense;
	}
}
```

a

---

3. (a)

MPI communicators determine the scope and the “communication universe” in which a point-to-point or collective operation is to operate. In other words, communicators define contexts within which groups of processes interact.

MPI collective operations are complex operations that are composed of a complex sequence of sends, receives and computations. They are collective operations because all processes in a communicator must call the collective operation.

The MPI_Allreduce operation computes a reduction, such as adding a collection of values together. An MPI_Op operation defines the reduction operation. When MPI_Allreduce is called, it reduces all elements in all send buffers using the MPI_Op specified and stores the results in all receive buffers.

(b)

MPI_Send() is the blocking send method.

MPI_Ssend() is the blocking and synchronous version of MPI_Send().

Blocking in MPI context refers to the relationship between the caller of a communication operation and the implementation of the operation. MPI_Send() will only return when the data to be sent is copied somewhere safe from the specified output buffer.

Synchronous in MPI contexts means that operations can only be completed when a matching operation has been posted. In this context, MPI_Ssend() will only return when a matching receive operation has been found and the receiver has started executing the matching receive. Communication does not complete at either end before both processes rendezvous at the communication.

Bad example, can someone flesh out a better one?

A simple situation where this difference can cause the behaviour of the program to vary dramatically for example, if MPI_Ssend() was used and there existed no other matching receiving operation, the sender would be stuck in a deadlock until a matching receiving operation had been posted. With MPI_Send(), the sender would copy the data from the buffer safely and move on.

(c)

(i)

*parallel_for*

Load balanced parallel execution of fixed number of independent loop iterations

*Range*

Range defines the range type to iterate over. A range type must include:

- A copy constructor and destructor must be defined
- The following methods must defined:
    - *is_empty()*
        - True if range is empty
    - *is_divisible()*
        - True if range can be partitioned
    - splitting constructor *R(R &r, split)*
        - Splits r into 2 subranges

*Body*

Body defines the body type that operates on the range or subrange. A body type must include:

- A copy constructor and destructor must be defined
- Defines method *operator()*
    - Apply the body to the sub range

*parallel_for* partitions the original range into subranges and deals out the subranges to worker threads such that cache efficiency is optimal, workload is balanced and scales well. *Range* specifies the range type for *parallel_for* to partition and deal the subranges to workers, and *Body* specifies the operation to perform on each subrange.

(ii)

The TBB runtime system uses *Range* to split the range in subranges, which are effectively tasks. The *Range* method *is_divisible()* defines the range's granularity. The TBB runtime system uses *is_divisible()* to check if a range can be subdivided, if it can be subdivided, the splitting constructor is used to split the range. Once a range can not be divided into subranges, the range is now ready for *Body operation()* to be performed to the range.

(d)

In addition to the operator() (Range &subrange) as in parallel_for a parallel_reduce body also needs to supply the method void::join (Body &rhs) which takes the result in rhs and merges it with the calling object.

...

<TODO>

Parallel reduce would merge and only keep the max of LHS,RHS returning this value

Parallel for would do this differently it would compare two values and keep one and then get another rather than returning both

<If anyone can given more details on the below, please do >

parallel_for

each process will run over the range finding the maximum .
The result will need to be stored in a new range which will have length p (the number of processors) , and will need to run on a single process to give a single result.

parallel_reduce

each process runs the operator() method to seek the max value from the range assigned to it. the results are then combined using a max() operator such that a single result is obtained.