CMU 15-418/618: Exam 2 Practice Exercise SOLUTIONS

Problem 1: Miscellaneous Short Answer Questions

A. The lock-free data structures discussed in lecture assumed a sequentially consistent address space. What new issues must be considered if a lock-free implementation is executed on a system with weaker memory consistency?

Solution: We need to consider the usage of memory fences now; otherwise, if the write performed by compare-and-swap is reordered to occur after a sequence of operations in the critical section (guarded by the compare-and-swap), the lock-free data structure could be structurally compromised.

B. What is the motivation for a content distribution network (CDN)? Why do Facebook images get served to your browser by a CDN but your main news feed is not?

Solution: CDN's are useful for servicing large media assets, such as video and images, to clients at high-bandwidth, low latency. These assets change infrequently and can be cached near end users. Facebook images are serviced to your browser via a CDN since they are easy to collect together and are relatively static, apart from requiring a higher-bandwidth to service efficiently. Your news feed is the accumulation of several different sources, which are highly dynamic (changing every minute). Additionally, no particular items require high-bandwidth to serve effectively. Overall, the extra logic needed to construct your news feed in real-time diminishes the utility of CDN's in optimizing this scenario.

C. Give two reasons why an implementation of a web server might spawn more worker threads than cores present in the machine it is running on?

Solution: The biggest reason is to hide the high latency of disk requests with the processing of other requests. Another answer might be to avoid critical-path latency issues in spinning up new threads (due to network traffic, setup time on the server, etc), it's better to have a pool of available threads that can be utilized on-demand.

D. If you replaced a highly contended lock with an `atomic{ }` block to enable transactional implementation of mutual exclusion, do you think pessimistic or optimistic conflict detection would yield better performance?

Solution: A pessimistic conflict detection scheme would be better since aborts are highly likely. (It's better to quit early instead of proceeding with a lot of wasted work.) However, it wold be important for the conflict manager to adopt a scheduling policy that avoids livelock due to constant rollback.

E. You get a job at Fitbit and your boss asks you to optimize some code to **improve the battery life** of their latest wearable product. You profile the code and find that most time is spent in the function `getting_fit()` which uses 10 arithmetic instructions to compute how much fitter the wearer is from his last 10 steps. You determine that it's possible to reduce the function to only 5 instructions if one of the instructions is a load from a precomputed lookup table stored in DRAM. You propose this solution to your boss and she shoots it down as a bad idea. Why? (Assume the table does not fit in the processor's cache, and that the Fitbit's memory system has infinite bandwidth and no latency.)

Solution: Your boss wanted you to optimize for energy, not performance. Your solution replaces a few arithmetic instructions with a load from off-chip memory. The result is that even your implementation executes fewer instructions than the original program, it has higher energy consumption than the original.

F. Your new startup's codebase makes use of two data structures: a stack and a linked list. Both structures are implemented using a single (per structure) global lock to avoid data races. Your developers suggest there is time before product launch to reimplement one of the two structures in a lock-free manner to enable concurrency. Assuming both structures are equally important to the overall performance of the program and equally difficult to implement, which structure's reimplementation do you recommend that your team focus effort on, and why? (Hint: we are looking for an answer that considers the potential for performance improvement from a lock-free implementation.)

Solution: You should choose the linked list, since there are opportunities for many threads to be operating concurrently in a list (e.g., operating on difference parts of the list). In contrast, since all operations on a stack manipulate the same element (the top of the stack), the amount of concurrency possible is very limited.

G. You are responsible for designing the packet format for the interconnect for the new multi-core chip by Yinzer Processors. It has already been decided that the interconnect will use **cut-through flow control**. You design a packet format where the front of the packet (i.e., the first bits that arrive at a switch) contain the checksum, and the end of the packet contains the destination node address, and present it to your team, and get nasty looks from everyone. Why are they unhappy with your design?

Solution: The packet format you designed basically reduces the network to store-and-forward routing since the routing information does not arrive until the end of the packet. The network switch has to wait until the entire packet has arrived in order to know where to send it next.

H. Consider a spin lock that uses **exponential back off** to reduce coherence traffic generated by threads waiting for the lock. How could this policy result in starvation? Briefly give one approach to avoiding starvation during lock acquisition while maintaining low traffic. (naming an alternative lock implementation is fine)

Solution: The problem here is the possibility of starvation. The more times a thread unsuccessfully tries to acquire the lock, the longer it waits for it's next chance (and the less likely it is to very acquire it). Solutions to this problem include using a fair lock implementation like a ticket lock or an array-based lock. Another possible solution would be to use a random wait time (where the wait time was the same for all threads but potentially increased when contention was detected.)

I. Imagine you are designing a parallel machine with P processors to execute convolutions in parallel on a 2D array. (Recall each array element is updated to be a weighed combination of all neighboring values). Do you advocate for a mesh or torus interconnect for this system? Why? (Hint: think about performance/cost trade-offs)

Solution: Assuming that the convolution is a basic convolution that respects boundaries of the 2D array, a mesh network is advisable since the extra toroidal links of the torus network would not be used. (The networks would yield similar performance, but the mesh would have lower cost.) We also accepted torus as the correct answer if students specified they were thinking about periodic boundary constraints on the convolution. (A periodic convolution would benefit from the "wrap around" links provided by the torus topology.)

J. Consider the difference between store-and-forward and cut-through flow control in a chip interconnect. Although cut-through flow control will typically reduce packet transmission latency, its behavior degenerates to that of the store-and-forward scheme under high contention. Why?

Solution: Consider a packet routed from A to B to C over links L_1 (from A to B) and L_2 (from B to C). Say the head of the packet has arrived at B. Under contention, it may take awhile for a router to reserve the next link (L_2) for the packet. While the router is waiting to reserve L_2, the remainder of the packet is flowing over L_1 and being buffered at B. If the entire packet arrives at B before L_2 can be reserved, then the cut-through scheme's behavior has degenerated to that of store-and-forward routing.

K. You and friend start a company called CMUTube that delivers high-quality video lectures to students around the world. Each day, your company makes all the lectures conducted at CMU available on a server and viewable on a mobile phone. Your friend is a L33t programmer and begins implementing his own video playback software in highly optimized ARM assembly. You recommend that he just use the high-level programming interfaces provided by iOS and Android for video playback. Even if your friend is the best programmer on the planet, why is his strategy misguided? (Hint: we are looking for an answer that addresses energy.)

Solution: Your friend is misguided because the implementation of the Android/iOS high-level interfaces for video processing will use fixed-function hardware present in these devices to perform video decode. Despite how good of a programmer he/she is, this hardware is likely to be significantly more energy efficient that your friend's solution. Your customers will be very unhappy if their phones run out of juice after only watching lecture videos for short time. Some answers also mentioned is that low-level tuning for a particular platform might now translate to other CPUs, which is also true.

L. Could your company's service benefit from content distribution networks? Why or why not?

Solution: This is a great application for content distribution networks. Video requires high bandwidth and is data that tends to change once it is posted. Rather than attempt to build a website that can service many video stream requests at once, replicating this data at CDNs around the world would be a very wise solution. Caching data near users would significantly reduce bandwidth on your servers. It would also yield lower latency to users (although latency is less important in a streaming video application).

M. Your job is to implement an iPhone app that continuously processes images from the camera. To provide a good experience, the app must run at a 30 fps. While developing the app, you always test it on a benchmark that contains 100 frames. After a few days of performance tuning the app consistently meets the performance requirement on this benchmark. Excited, you go to show your boss! She starts walking around the office with the app on, testing out how it works. After a few minutes of running the app, she comes back to your desk and says, "You're going to have to try harder, the app is not fast enough – it's not running at 30 frames per second". No changes have been made to the app. What happened?

Solution: The continuously running app caused the device to heat up and the processor decreased its clock frequency, reducing your app's performance. Although your app was efficient enough to meet the performance requirement when running for a short period of time (about 3 seconds), is was not efficient enough to maintain the desired level of performance at lower clock rates. ***Grading note: a number of students provided answers that were very different from the content of the course. Unexpected systems-related answers may be given consideration for credit.***

N. Given `atomicCAS` please implement a lock-free `atomicOR` below. (You do not need to worry about the ABA problem.) **In this problem, have atomicCAS return the result AFTER the OR occurs.**

```
// atomically bitwise OR the value val with the current value in addr
int atomicOR(int* addr, int val) {
   while (1) {
     int old = *addr;
     int new = old | val;
     if (atomicCAS(addr, old, new) == old)
        return new;
   }
}
```

O. Consider an application whose runtime on a single-core processor of type Core A consists of 80% deep neural network (DNN) evaluation and 20% graph processing. **All the DNN work must complete before the graph processing work is started (graph processing depends on the DNN output, so they cannot be done in parallel). However, each stage itself is perfectly parallelizable.** Imagine there are two possible cores, each with different performance characteristics on the DNN and graph processing workloads. The chart below gives the performance of Core B relative to core A on the two workloads (these are relative throughputs).

	Core Type	
	Core A	Core B
Resource Cost	1	1
Perf (throughput): DNN Eval	1	4
Perf (throughput): Graph Processing	1	0.5

If you had to design a multi-core processor for this workload that could have any mixture of cores so long as the total resource cost did not exceed 4, what mixture would you choose? **In your answer state (approximately) what speedup over a processor with one core of type A would you expect?** Hint: In each step of the problem its wise to parallelize work across all the chosen cores.

Math hint: consider a workload that spends 800 ms in DNN eval and 200 ms in graph processing on a single Core A. How much time would the computation take on a quad-core Core B processor?

Also, since the math in this problem isn't the cleanest... $200/2.5 = 80$, $200/3 \approx 67$, $800/13 \approx 62$

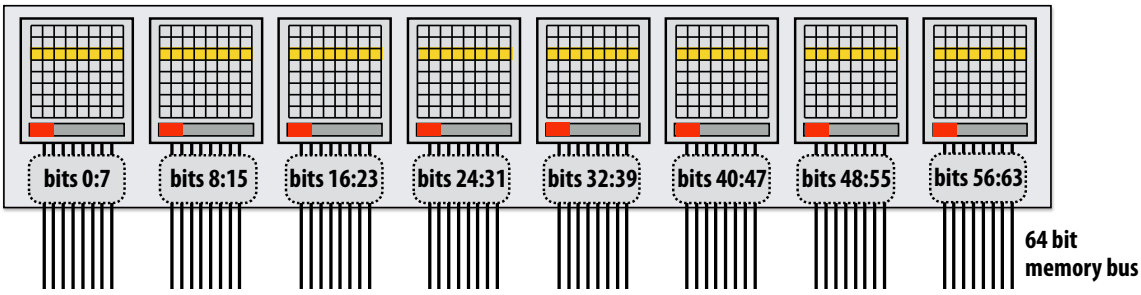
Solution: You want 1 core of type A and 3 cores of type B. The resulting cost is $\frac{800}{1+3\times4} + \frac{200}{1+3\times.5} = \frac{800}{13} + \frac{200}{2.5} = 62 + 80 = 142$. *Therefore the speedup is* $\frac{1000}{142} \approx 7$. *It's useful to note that 4 cores of type B gives a cost of* $\frac{800}{16} + \frac{200}{2} = 50 + 100 = 150$, *which is higher than the heterogeneous case.*

P. (2 pts) Next year all CMU students will be required to take 15-418/618, which will generate increased load on the course website. To keep response times on the course website low, Tom Cortina suggests that we buy a bunch of high-end machines for webservers and keep them in the machine room at CMU. The 15-418/618 TAs, **observing the figure on the front page of this exam**, suggest an alternative way to design the website to meet this load. What is it?

Solution: The figure on the front page suggests that load on the 15-418/618 website is quite bursty; it spikes before exams. Therefore, the TAs suggest we build the website to be elastic, and scale up the number of servers only in the days before exams.

Problem 2: Controlling DRAM

Consider a DRAM DIMM with 8 chips (8-bit interface per chip) just like what we talked about in class and in exercise 6. Physical memory addresses are strided across the chips as in the figure below, so that 64 consecutive bits from the address space can be read in a single clock over the bus. The DRAM row size is 2 kilobits (256 bytes). There is only a single bank per chip. (We ignore banking in this problem.)



The memory controller processes requests with the following logic:

```
int active_row;   // stores active row

handle_64bit_request(void* addr) {

   int row, col;

   compute_row_col(addr, &row, &col);   // compute row/col from addr (0 cycles)

   if (row != active_row)
      activate_row(row);     // this operation takes 15 cycles

   transfer_column(col);     // this operation takes 1 cycle
}
```

Questions are on the next page...

Now consider the following C-program, which executes using 2 pthreads on a dual-core processor with a single shared cache.

```
struct ThreadArg {
  int threadId;
  double sum;  // thread-local variable
  int N;       // assume this is very large
  double* A;   // pointer to shared array
};

// each thread processes one half of array A
void myfunc(void* targ) {
  ThreadArg* arg = (ThreadArg*)targ;
  arg->sum = 0.f;
  int offset = arg->threadId * arg->N / 2;
  for (int i=0; i<arg->N / 2; i++)
    arg->sum += arg->A[offset + i];
}

/* main code */

ThreadArg args[2];
args[0].threadId = 0; args[1].threadId = 1;
args[0].A = args[1].A = new double[N];

// initialize args[].sum, args[].N, args[].A, and launch two pthreads here that run myfunc
// Then wait for threads to complete

print("%f\n", args[0].sum + args[1].sum);
```

A. Assume that the two threads run at approximately the same speed, so the memory controller receives requests from the two threads in interleaved order: thread0_req0, thread1_req0, thread0_req1, thread1_req1, etc. Given this stream, what is the effective bandwidth of the memory system as observed by the processor (the rate at which it receives data)? Assume that:

   - The program is bandwidth bound so that the memory system always has a deep queue of requests to process.
   - The granularity of transfer between the memory controller and the cache is 64 bits. (e.g., 8-byte cache line size)
   - Note that array elements are DOUBLES (8 bytes).

   Solution: Because of the interleaved request streams, each memory access will trigger a row-buffer miss. As a result, the memory system needs 16 clocks to provide 64 bits (8 bytes) back to the processor. The overall effective bandwidth is 8/16 = 0.5 bytes per clock (or 4 bits per clock)

B. Modify the program code to significantly improve the effective memory system bandwidth. What is the new bandwidth you observe?

Solution: Change the blocked assignment of array elements to threads to an interleaved assignment. As a result of this change the memory system receives a stream of contiguous addresses, and the system achieves maximum row buffer locality. The system will obtain very near its peak bandwidth of 8 bytes per clock.

C. Return to the original code given in this assignment (ignore your solution to part B), and assume that requests now arrive at the memory controller every ten cycles. For example...

```
cycle 0: thread 0 req 0
cycle 10: thread 1 req 0
cycle 20: thread 0 req 1
cycle 30: thread 1 req 1
cycle 40: thread 0 req 2
cycle 50: thread 1 req 2
cycle 60: thread 0 req 3
...
```

Write (rough) pseudocode for a memory request scheduling algorithm that allowed the memory system to keep up with this request stream. **Your implementation can assume there is an incoming request buffer called `request_buf` that holds up to 4 requests.** (The processor stalls if the request buffer is full.)

Solution: The main idea is to buffer requests and reorder how they are serviced so that two requests from the same stream are handled back-to-back. As a result, the second request of the pair will be a row-buffer hit. Specifically this scheme will complete the incoming requests at the following times:

```
t0, req 0: 16      buffer: t1:r0
t1, req 0: 32      buffer: t0:r1, t1:r1
t1, req 1: 33      buffer: t0:r1
t0, req 1: 49      buffer: t0:r2
t0, req 2: 50      buffer: t1:r2
t1, req 2: 66      buffer: t0:r3 ...
```

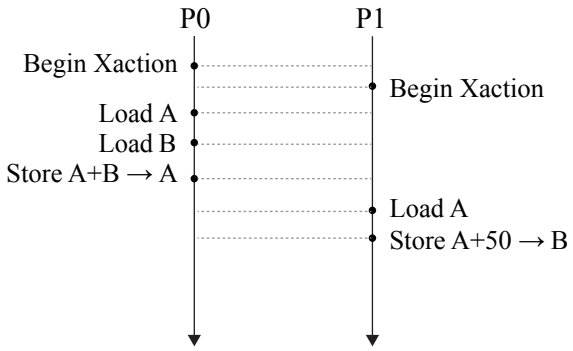
D. (TRICKY, YOU MAY WANT TO COME BACK TO THIS ONE) You add hardware multi-threading to your dual-core processor (2 threads-per core) and spawn four pthreads in your code. You assign contiguous blocks of the input array to each pthread. Assuming the request arrival rate stays the same (but now requests from four threads, rather than two, are interleaved), how would you change your solution in part C to keep up with the request stream? (you may modify the buffer size if need be). Is overall memory latency higher or lower than in part C? Why?

Solution: The same scheme used in part C still works (including a length 4 buffer), but now requests to adjacent addresses (in the same DRAM row) come every 4th request and are separated in time by 40 cycles. On average memory latency is a bit longer than in part C.

```
t0, req0: 16   buffer = t1:r0
t1, req0: 32   buffer = t2:r0, t3:r0
t2, req0: 48   buffer = t3:r0, t0:r1
t3, req0: 64   buffer = t0:r1, t1:r1, t2:r1
t0, req1: 80   buffer = t1:r1, t2:r1, t3:r1, t0:r2
t0, re12: 81   buffer = t1:r1, t2:r1, t3:r1
t1, req1: 97   buffer = t2:r1, t3:r1, t1:r2
t1, req2: 98   buffer = r2:r1, t3:r1 ...
```

Problem 3: Transactional Memory

A. Consider the following schedule of operations performed by processors P0 and P1.



Assume that at the start of the program A=0 and B=100 and that this system implements **lazy, optimistic** transactional memory.

Notice that the schedule above does not designate when the end of transactions occur. Fill in the table below for each possible schedule of transaction commits. Indicate whether P0 or P1 (or both) execute a rollback, and fill in the final values of A and B after **both transactions are complete**.

Important! For row 3, you may wish to state your assumptions about the details of the transactional memory implementation to justify your answer.

	P0 rollback (y/n)	P1 rollback (y/n)	A	B
P0 reaches end of transaction before P1 (but after P1's performs loads/stores to A and B)	no	yes	100	150
P1 reaches end of transaction before P0	yes	no	50	50
P0 reaches end of transaction before P1 performs loads/stores to A and B)	no	no	100	150

The situation in row one yields output that is consistent with a serial ordering where P0 computes its work, then P1 observes these results, and then P1 computes its work.

The situation in row two yields output that is consistent with a serial ordering where P1 computes its work, then P0 observes these results, and then P0 does its work.

The provided answer for row three assumes the hardware transactional memory discussed in class. When P0 commits, the read/write set of the pending transaction by P1 does not contain A or B. Thus there is no need to roll back the pending P1 transaction. When this transaction ultimately does read A, it will observe the results of the transaction. Alternatively, an answer that was equivalent to the correct answer to row one was also acceptable, provides a reasonable justification was given.

Problem 4: Paparazzi Camera

You are designing a heterogeneous multi-core processor to perform real-time "celebrity detection" on a future camera. The camera will continuously process low-resolution live video and snap a high-resolution picture whenever it identifies a subject in a database of 100 celebrities. Pseudocode for its behavior is below:

```
void process_video_frame(Image input_frame)
{
  Image face_image = detect_face(input_frame);
  for (int i=0; i<100; i++)
     if (match_face(face_image, database_face[i]))
        take_high_res_photo();
}
```

In order to not miss the shot, the camera MUST call `take_high_res_photo` within 500 ms of the start of the original call to `process_video_frame`! To keep things simple:

- Assume the code loops through all 100 database images regardless of whether a match is found (e.g., we want to find all matches).
- The system has plenty of bandwidth for any number of cores.

Two types of cores are available to use in your chip. One is a fixed-function unit that accelerates `detect_face`, the other is a general-purpose processor. The cost (in chip resources) of the cores and their performance (in ms) executing important functions in the pseudocode are given below:

Operation	Core Type	
	C1 (fixed-function)	C2 (Programmable)
Resource Cost	1	1
Perf (ms): `detect_face`	100	400
Perf (ms): `match_face`	N/A	20

A. Assume a video frame arrives exactly every 500 ms. **If you only use cores of type C2**, how many cores do you need to meet the performance requirement for the video stream? (You cannot change the algorithm, and please justify your answer).

Solution: You need 20 cores. Since `detect_face` requires 400 ms to execute on C2, 100 ms are left to execute 100 calls to `match_face`. Since each call takes 20 ms, 20 cores are needed to complete the loop in this amount of time. (Each core performs 5 matches.)

B. Your team has just built a multi-core processor that contains a large number of cores of type C2. It achieves 5.9× speedup on the camera workload discussed above. Amdahl's Law says that the maximum speedup of the camera pipeline in this problem should be $2400/400 = 6\times$, so your team is happy. They are shocked when your boss demands a speedup of $10\times$. Your team is on the verge of quitting due "unreasonable demands". How do you argue to them that the goal is reasonable one if they consider all the possibilities in the above table? (Hint: What assumption are they making in their Amdahl's Law calculations, and why does it not hold?)

Solution: You argue that the primary limiter of speedup is the sequential nature of `detect_face`*, but that using C1 instead of C2 can accelerate this sequential computation, effectively making it a smaller fraction of execution time. As a result, and as you will show in the next question, using C1 to reduce the execution time of* `detect_face` *allows for a much greater speedup. (or similarly, it allows for meeting the performance requirement using far fewer resources.).*

C. Now assume you can use both cores of types C1 and C2 in your design. How many of each core do you choose to minimize resource usage, while still meeting the same performance requirements as in part A? Does your new chip use more or fewer total resources than your solution in part A (by how much)?

Solution: Use one core of type C1 and 5 cores of type C2. Now, C1 completes `detect_face` *in 100 ms, allowing 400 ms to perform matching. 5 cores are needed to complete matching in 400 ms (each C2 core performs 20 matches). Using the heterogeneous configuration, your processor meets the performance requirement using only 6 units of resources, down from 20 from part A. This is more than three times less!*

D. Beyonce is about to release a new album, and your paparazzi customers want to follow her around all day. They request a camera that is more energy efficient. Energy efficiency is so important they are willing to relax their performance requirement and allow high-res photos to be taken within 1500 ms (not 500 ms) of a video frame arriving. You find that the primary consumer of power in your application is loading database faces from memory. Describe how you would change the pseudocode to **approximately double** the energy efficiency of the camera while still meeting the performance requirements. You should assume you use the same processor design as in part C (or part A for that matter), and that your processor has a cache that holds up to 4 database images.

Solution: A simple solution is to increase locality by reordering the computation to batch up every two frames. That is, the system waits for two frames to arrive before beginning processing. It then amortizes the load of each database image over two matching operations, effectively reducing the bandwidth required by the system by a factor of two.

This optimization is possible because the latency requirement has been relaxed in this question. In each image pair, the first image is held up for 600 ms until the next image arrives and faces are detected. The matching computation takes 800 ms to complete on the 5 C2 cores, and as a result, the latency of the first frame is 1400 ms, while the latency of the second frame is 800 ms. Thus, all frames meet the latency requirement.

NOTE: Some answers described using many processors to keep the entire database, or a fraction of the database in cache. These solutions were handled on a case-by-case basis. (Many of them resulted in thrash caching and did not reduce total bandwidth requirement at all.)

Problem 5: A Lock-Free Stack

In class we discussed the following implementation of a lock-free stack of integers.

```
// CAS function prototype: update address with new_value if its contents
// match expected_value.  Return value of addr (at start of operation).
Node* compare_and_swap(Node** addr, Node* expected_value, Node* new_value);

struct Node {
  Node* hello;
  int value;
};

struct Stack {
  Node* top;
};

void init(Stack* s) {
  s->top = NULL;
}

void push(Stack* s, Node* n) {
  while (1) {
    Node* old_top = s->top;
    n->next = old_top;
    if (compare_and_swap(&s->top, old_top, n) == old_top)
      return;
  }
}

Node* pop(Stack* s) {
  while (1) {
    Node* top = s->top;
    if (top == NULL)
      return NULL;
    Node* new_top = top->next;
    if (compare_and_swap(&s->top, top, new_top) == top)
      return top;
  }
}
```

A. We talked about how this implementation could fail due to the "ABA problem". What is the ABA problem? Describe a sequence of operations that causes it.

Solution: The ABA problem occurs when the one thread observes the system in state A, then the system changes from state A to B to A', but CAS cannot distinguish between A and A' and incorrectly succeeds. In the stack example, this will be when a node A is initially on the stack and thread 1 attempts to remove A. If prior to thread 1's CAS, another thread removes A, performs other stack operations, and pushes A back on the stack, thread 1's CAS will incorrectly succeed. This will corrupt the stack. The following sequence of operations would corrupt the stack, causing node D to be lost.

```
 Processor 1       |     Processor 2      |   The stack
 ------------------+----------------------+--------------------
                   |                      |      A->B->
  old_top = s->top |                      |      A->B->
                   |     A = pop()        |         B->
                   |     push(D)          |      D->B->
                   |     push(A)          |   A->D->B->
  cas()            |                      |         B->
```

B. Even though the above implementation is lock free, it does not mean it is free of contention. In a system with P processors, imagine a situation where all P processors are contending to pop from the stack. Describe a potential performance problem with the current implementation and describe one potential solution strategy. (A simple descriptive answer is fine.)

Solution: The problem is that "spinning" in the CAS generates a lot of cache coherence traffic. There are a variety of solutions to reduce inefficiencies of spinning. We accepted the solutions "use an exponential back-off strategy" or "perform a non-exclusive read before the CAS" (i.e., a test prior to the compare-and-swap). We also accepted the answer that the problem was a lack of fairness due to the lack of fairness guarantees by the CAS, with the solution being to use a ticket lock system.

Problem 6: Deletion from a Binary Tree

The code below, and continuing on the following two pages implements node deletion from a binary search tree. We have left space in the code for you to insert locks to ensure thread-safe access and high concurrency. You may assume functions `lock(x)` and `unlock(x)` exist (where `x` has type `Lock`). **Your solution NEED ONLY consider the delete operation.**

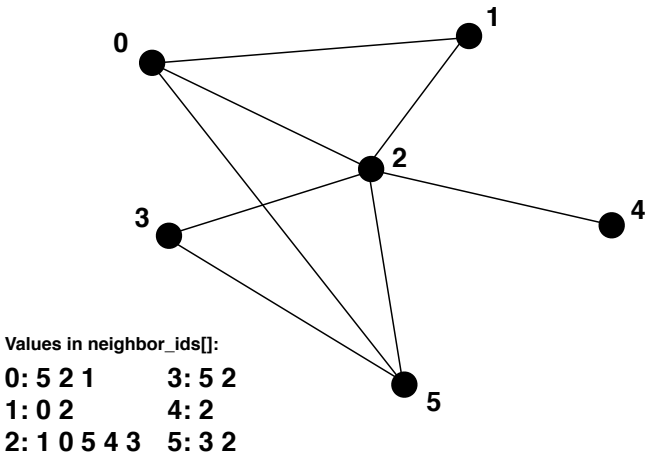
```
// opaque definition of a lock.  You can call 'lock' and 'unlock' on
// instances of this type
struct Lock;


// definition of a binary search tree node.  You may edit this structure.
struct Node {

   int value;
   Node* left;
   Node* right;

   Lock lock;

};
```

NOTE: This solution is an aggressive solution that only takes a lock on `cur` when it is absolutely neccessary. Other hand-over-hand solutions are possible.

```
// Delete node containing value given by 'value'
// Note: For simplicity, assume that the value to be deleted is not the root node!!!
void delete_node(Node* root, int value) {

   Node** prev_link = NULL;              // pointer to link to current node
   Node* cur = root;                     // current node during traversal (the node we look at for value comparison)
   Node* prev = NULL;                    // comes before the cur node (this is the node we hold the lock on)


   // note: no lock on cur at this stage, but that's okay as long as we don't access
   // cur->left or cur->right
   while (cur) {                         // while node we are trying delete is not not found

      if (value == cur->value) {         // found node to delete!
         lock(cur->lock);                // gaurantees cur->left or cur->right cannot be deleted
         // Case 1: Node is leaf, so just remove it, and update the parent node's pointer to
         // this node to point to NULL
         if (cur->left == NULL && cur->right == NULL) {
            if (prev != NULL) {
               *prev_link = NULL;
               unlock(prev);
            }
            unlock(cur);
            free(cur);
            return;
         } else if (cur->left == NULL) {
            // Case 2: Node has one child. Make parent node point to this child
            if (prev != NULL) {
               *prev_link = cur->right;
               unlock(prev);
            }
            unlock(cur);
            free(cur);
            return;
         } else if (cur->right == NULL) {
            // *** ignore this case, symmetric with previous case ***
         } else {
            // Case 3: Node has two children.  Move the next larger value in the tree into this
            // position (smallest node in the right-subtree).  The subroutine delete_helper returns
            // this value and executes the swap by removing the node containing the next larger value.
            // SEE NEXT PAGE

            // We need to hold cur->lock the entire time we're in the right subtree
            // We will need to perform hand-over-hand locking while traversing the subtree since there
            // could be a thread trying to delete a node in the subtree
            if (prev != NULL) unlock(prev->lock);
            cur->value = delete_helper(cur->right, &cur->right);
            unlock(cur->lock);
            return;
         }

      } else if (value < cur->value) {
         // still searching, traverse left
         lock(cur->lock);  // careful: grab cur lock BEFORE letting go of prev (hand over hand)
         if (prev != NULL) unlock(prev->lock);
         prev_link = &cur->left;
         prev = cur;
         cur = cur->left;
      } else {
         // still searching traverse right
         // *** ignore this case, symmetric with previous case ***
      }

   }
}
```

```
// The helper method removes the smallest node in the tree rooted at node n.
// It returns the value of the smallest node
// There is NO LOCK on n at this time, but prev is LOCKED
int delete_helper(Node* n, Node* prev) {

   Node** prev_link = &prev->right;  // link to this subtree from node to delete
   Node* cur = n;
   Node* prev = NULL;   // a little odd this can be NULL when prev_link is not, but use
                        // if (prev != NULL) checks below to avoid releasing lock on node to
                        // delete (which is prev here)

   // at this stage, we just have a lock on the node we are about to delete and cur is
   // the right child of that node. Need a lock on cur to proceed, since otherwise
   // cur->left/right might be modified out from under us
   lock(cur->lock);

   // search for the smallest value in the tree by always traversing left
   while (cur->left) {
      if (prev != NULL) unlock(prev->lock);
      prev_link = &cur->left;
      prev = cur;
      cur = cur->left;
      lock(cur->lock);
   }

   // this is the smallest value.  We are holding a lock on the node we are about to delete.
   // prev_link is the address of the link to NULL
   int value = cur->value;

   // remove the node with the smallest value
   if (cur->right == NULL) {
      // Case 1: Node is a leaf
      *prev_link = NULL;
   } else {
      // Case 2: Node has a right child
      *prev_link = cur->right;
   }

   if (prev != NULL) unlock(prev->lock);
   unlock(cur->lock);
   free(cur);
   return value;
}
```

Problem 7: Updating a Graph




```
struct Graph_node {
   Lock   lock;
   float  value;
   int    num_edges;    // number of edges connecting to node (its degree)
   int*   neighbor_ids; // array of indices of adjacent nodes
};

// a graph is a list of nodes, just like in assignment 3
Graph_node graph[MAX_NODES];
```

Consider the undirected graph representation shown in the code above. The figure shows an example graph. In the bottom-left of the figure are the values in `neighbor_ids` for each node.

Your boss asks you to write a program that atomically updates each graph node's `value` field by setting it to the average of all the values of neighboring nodes. The program must obtain a lock on the current node and all adjacent nodes to perform the update. It does so as follows...

```
void update(int id) {
    Graph_node* n = &graph[id];
    LOCK(n->lock);
    for (int i=0; i<n->num_edges; i++)
      LOCK(graph[n->neighbor_ids[i]].lock);

    // now perform computation...
```

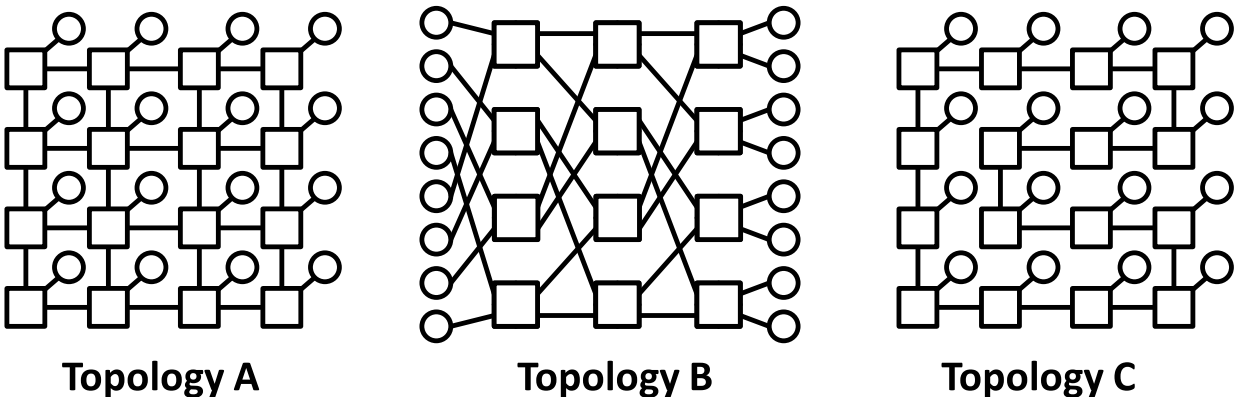
Question on the next page...

You run your update code in parallel on nodes 0 and 2 and deadlock occurs. Please provide a solution (a sketch in pseudocode is fine) to the deadlock problem that maintains high concurrency and limits the amount of extra work that is performed in the `update` function. **Hint: your solution may involve preprocessing of the graph data structure (if so, describe it).**

Solution: The deadlock problem occurs because there is a hold-and wait situation with circular dependencies. The thread processing vertex 0 grabs the lock on vertex 5, and subsequently requires the lock on 1. The thread processing vertex 2 immediately grabs the lock on 1, and subsequently needs a lock on 5. Deadlock! A simple solution would be to sort the edges in the graph structure in ascending order, ***while making sure you code takes the lock on the current node in the correct position in this list.*** *Then all threads would request locks in the same order (1 before 5), removing the circular dependency.*

Problem 8: Interconnection Networks

A. The figure below shows four common network topologies (circles and squares represent network endpoints and routers respectively).



Identify each topology and fill in the table below. Express the bisection bandwidth in Gbit/s, assuming each link is 1 Gbit/s. Express the cost and latency in terms of the number of network nodes N, using Big O notation, e.g., O(logN)).

	Topology A	**Topology B**	**Topology C**
Topology Type/Name	Mesh	Log. Multi-Stage	Ring
Direct or Indirect	Direct	Indirect	Direct
Blocking or Non-Blocking	Blocking	Blocking	Blocking
Bisection Bandwidth	4 Gbit/s	8 Gbit/s	2 Gbit/s
Cost	O(N)	O(NlogN)	O(N)
Latency	O(sqrt(N))	O(logN)	O(N)

B. Briefly describe two advantages and two disadvantages of circuit-switched networks compared to packet-based networks.

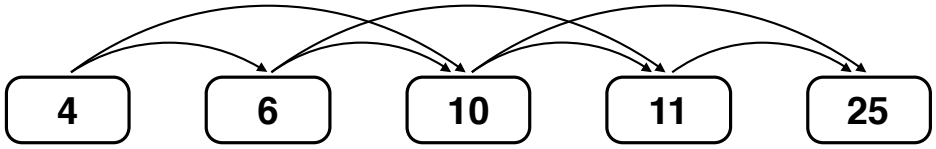
Advantages:
- No need for buffering
- No contention (flow isolation)

Disadvantages:
- Handshake overhead (for setting up and tearing connections)
- Lower link utilization (two flows cannot use the same link)

C. What common networking problem do virtual channels solve?

Solution: Head-of-line blocking

Problem 9: Another Fine-Grained Locking Question



Consider the semi-skip list structure pictured above. Each node maintains a pointer to the next node and the next-next node in the list. **The list must be kept in sorted order.** A node struct is given below.

```
struct Node {
   int value;
   Node* next;
   Node* skip;   // note that skip == next->next
};
```

Please describe a thread-safe implementation of **node deletion** from this data structure. You may assume that deletion is the only operation the data structure supports. Please write C-like pseudocode.

To keep things simple, you can ignore edge-cases near the front and end of the list (assume that you're not deleting the first two or last two nodes in the list, and the node to delete is in the list). If you define local variables like `curNode`, `prevNode`, etc. just state your assumptions about them. **However, please clearly state what per-node locks are held at the start of your process. E.g., "I start by holding locks on the first two nodes.".** Full credit will only be given for solutions that maximize concurrency.

(The solution is on the next page.)

Solution: The basic idea is to execute basic 1-2-1 hand-over hand locking until the current node's skip pointer points at the node to delete. Then lock `next` *and* `skip` *(three locks in total). The invariant is that a node cannot be deleted unless the thread has the two prior node locks. Solutions that employed hand-over-hand locking correctly were given at least 12 points. 15 points were saved for solutions approximately as efficient as that below.*

```
// delete node containing value
void delete_node(Node* head, int value) {

    // lock one node at the beginning
    // * only one lock is enough to keep the operation safe
    // * to traverse the semi-skip list and find the target node
    lock_node(head);
    Node *curr = head;

    // find the target node
    // * note that we check if curr->skip is the target
    while(curr->skip->value != value) {
      Node *next = curr->next;
      // the "hand-over-hand" locking            <-- this is very important!
      // * lock one more node before you release the previous one
      // * otherwise curr node can be deleted by another thread
      lock_node(next);
      unlock_node(curr);
      curr = next;
    }
    // coming out of the loop, the following is true:
    // * curr is locked
    // * curr->skip is the node we want to delete.

    // delete the target node: curr->skip
    // * note that we lock three nodes in total: <-- this is important too!
    // *         curr,                  curr->next,       curr->skip
    // * the node skip to target, the node before target,   target
    // * in order to keep the delete operation safe
    // * need to lock curr and curr->next because you are modifying them
    // * need to lock curr->skip to make sure nobody is traversing target
    Node *next = curr->next;
    Node *to_delete = curr->skip;
    lock_node(next);
    lock_node(to_delete);
    curr->skip = next->skip;
    next->next = next->skip;
    next->skip = next->skip->next;
    unlock_node(curr);
    unlock_node(next);
    unlock_node(to_delete);
}
```

Problem 10: Bringing Locality Back

Justin Timberlake and Kanye West hear that Spark is all the rage and decide they are going to code up their own implementation to compete against that of the Apache project. Justin's first test runs the following Spark program, which creates four RDDs. The program takes Justin's lengthy (1 TB!) list of dancing tips and finds all misspelled words.

```
var lines = spark.textFile("hdfs://mydancetips.txt'');     // 1 TB file
var lower = lines.map( x => x.toLower() );                 // convert lines to lower case
var words = lower.flatmap( x => x.split(`` '') );          // convert RDD of lines to RDD of
                                                           // individual words
var misspelled = words.filter( x => !x.isInDictionary() ); // filter to find misspellings

print misspelled.count();   // print number of misspelled words
```

A. Understanding that the Spark RDD abstraction affords many possible implementations, Justin decides to keep things simple and implements his Spark runtime such that each RDD is implemented by a fully allocated array. This array is stored either in memory or on disk depending on the size of the RDD and available RAM. **The array is allocated and populated at the time the RDD is created — as a result of executing the appropriate operator (map, flatmap, filter, etc.) on the input RDD.**

   Justin runs his program on a cluster with 10 computers, each of which has 100 GB of memory. The program gets correct results, but Justin is devastated because the program runs *incredibly slow*. He calls his friend Taylor Swift, ready to give up on the venture. Encouragingly, Taylor says, "shake it off Justin", just run your code on 40 computers. Justin does this and observes a speedup much greater than $4\times$ his original performance. Why is this the case?

   Solution: The program creates four RDDs, and since Justin's implementation elects to evaluate and materialize the contents RDD's immediately, the implementation will require up to 4 TB of memory to store all of these structures. There is only 1 TB of memory in aggregate across the 10 nodes of the cluster, so they will need to be stored and loaded from disk to implement each operation. The computation will be disk I/O limited. By increasing the cluster size to 40 nodes, there is now 4 TB of memory across the cluster, and the RDDs can be stored in memory and not out on disk. This will yield a significant performance improvement. Note: Even if the RDD were freed immediately after use, the system would need about 2 TB of memory and still require on-disk storage.

B. With things looking good, Kanye runs off to write a new single "All of the Nodes" to use in the marketing for their product. At that moment, Taylor calls back, and says "Actually, Justin, I think you can schedule the computations much more efficiently and get very good performance with less memory and far fewer nodes." Describe how you would change how Justin schedules his Spark computations to improve memory efficiency and performance.

Solution: Taylor is pointing out that the semantics of Spark RDDs permit a much more efficient implementation. The computation can be reordered to that instead of performing one operation on all elements of an RDD before proceeding to the new, the entire pipeline of operations can be performed on a chunk of the dataset all at once, with different chunks running in parallel on all the nodes. For example, an implementation could load 1 MB of lines of the input file from disk, run the entire sequence of RDD operations on this chunk (storing intermediate data in cache). This would be exceptionally memory efficient, and take advantage of the producer-consumer locality in the problem.

C. After hacking all night, the next day, Justin, Kanye, and Taylor run the optimized program on 10 nodes. The program runs for 1 hour, and then right before `misspelled.count()` returns, node 6 crashes. Kanye is irate! He runs onto the machine room floor, pushing Taylor aside and says, "Taylor, I have a single to release, and I don't have time to deal with rerunning your programs from scratch. Geez, I already made you famous." Taylor gives Kayne a stink eye and says, "Don't worry, it will be complete in just a few minutes." Approximately how long will it take after the crash for the program to complete? You should assume the `.count()` operation is essentially free. But please **clearly state any assumptions about how the computation is scheduled in justifying your answer.**

Solution: Assuming the elements of the RDDs are partitioned equally across the cluster, losing one node results in a loss of 1/10th of the output. Since the system partitioned the data across the cluster, the system knows exactly which partition of elements where lost. Given the lineage of the RDD `misspelled`*, this partition can be recomputed in parallel across all remaining 9 nodes. If one node took 60 minutes to compute this partition, 9 nodes would so the work in 60/9 = 6.6 minutes. We also accepted 6 minutes as a perfectly valid answer, since if the failed node was replaced then the lost partition could be recomputed in 60/10=6 minutes.*

Problem 11: Two Box Blurs are Better Than One

An interesting fact is that repeatedly convolving an image with a box filter (a filter kernel with equal weights, such as the one often used in class) is equivalent to convolving the image with a Gaussian filter. Consider the program below, which runs two iterations of box blur.

```
float input[HEIGHT][WIDTH];
float temp[HEIGHT][WIDTH];
float output[HEIGHT][WIDTH];

float weight;   // assume initialized to (1/FILTER_SIZE)^2

void convolve(float output[HEIGHT][WIDTH], float input[HEIGHT][WIDTH], float weight) {

  for (int j=0; j<HEIGHT; j++) {
    for (int i=0; i<WIDTH; i++) {
      float accum = 0.f;
      for (int jj=0; jj<FILTER_SIZE; jj++) {
        for (int ii=0; ii<FILTER_SIZE; ii++) {

          // ignore out-of-bounds accesses (assume indexing off the end of image is
          // handled by special case boundary code (not shown)

          // count as one math op (one multiply add)
          accum += weight * input[j-FILTER_SIZE/2+jj][i-FILTER_SIZE/2+ii];
        }
      }
      output[j][i] = accum;
    }
  }
}

convolve(temp, input, weight);
convolve(output, temp, weight);
```

A. Assume the code above is run on a processor that can comfortably store `FILTER_SIZE*WIDTH` elements of an image in cache, so that when executing `convolve` each element in the input array is loaded from memory exactly once. What is the arithmetic intensity of the program, in units of math operations per element load?

Solution: It is $\texttt{FILTER\_SIZE}^2/2$*, since each input and output pixel are read exactly once, and each* `convolve` *operation performs* $\texttt{FILTER\_SIZE}^2$ *operations per pixel. We also accepted* $\texttt{FILTER\_SIZE}^2$ *for full credit since the question referred to "per element load".*

Many times in class Prof. Kayvon emphasized the need to increase arithmetic intensity by exploiting producer-consumer locality. But sometimes it is tricky to do so. Consider an implementation that attempts to double arithmetic intensity of the program above by producing 2D chunks of output at a time. Specifically the loop nest would be changed to the following, **which now evaluates BOTH CONVOLUTIONS.**

```
for (int j=0; j<HEIGHT; j+=CHUNK_SIZE) {
  for (int i=0; i<WIDTH; i+=CHUNK_SIZE) {

    float temp[..][..];  // you must compute the size of this allocation in 6B

    // compute required elements of temp here (via convolution on region of input)

    // Note how elements in the range temp[0][0] -- temp[FILTER_SIZE-1][FILTER_SIZE-1] are the temp
    // inputs needed to compute the top-left corner pixel of this chunk

    for (int chunkj=0; chunkj<CHUNK_SIZE; chunkj++) {
      for (int chunki=0; chunki<CHUNK_SIZE; chunki++) {
        int iidx = i + chunki;
        int jidx = j + chunkj;
        float accum = 0.f;
        for (int jj=0; jj<FILTER_SIZE; jj++) {
          for (int ii=0; ii<FILTER_SIZE; ii++) {
            accum += weight * temp[chunkj+jj][chunki+ii];
          }
        }
        output[jidx][iidx] = accum;
      }
    }
  }
}
```

B. Give an expression for the number of elements in the `temp` allocation.

Solution: `(CHUNK_SIZE+FILTER_SIZE-1)`^{2}*. We also accepted* `(CHUNK_SIZE+FILTER_SIZE)`^{2} *for full credit.*

C. Assuming `CHUNK_SIZE` is 8 and `FILTER_SIZE` is 5, give an expression of the **total amount of arithmetic performed per pixel of output** in the code above. You do not need to reduce the expression to a numeric value.

Solution: Need $12 \times 12 = 144$ *elements of* `temp` $= 5 \times 5 \times 144 = 3600$ *operations. Producing 64 elements of* `output` *is another* $64 \times 25 = 1600$ *operations. So there are now* $\frac{3600+1600}{64} = \frac{5200}{64} \approx 81$ *operations per pixel, compared to* $2 \times 25 = 50$ *operations per pixel in part A.*

D. Will the transformation given above improve or hurt performance if the original program from part A was *compute bound* for this `FILTER_SIZE`? Why?

*Solution: It will hurt performance since it increases the number of arithmetic operations that need to be performed, and the program is already compute bound. Note that a fair number of students said that the problem is was **arithmetic intensity was increased**, hence the slowdown. Increasing arithmetic intensity of a compute-bound program will not change its runtime if the total amount of work stays the same (it just reduces memory traffic). The essence of the answer here is that more work is being done.*

E. Why might the chunking transformation described above be a useful transformation in a mobile processing setting regardless of whether or not it impacts performance?

Solution: Since the energy cost of data transfer to/from DRAM is significantly higher than the cost of performing an arithmetic operation, reducing the amount of data transfer is likely to reduce the energy cost of running the program. Some students mentioned that reduced memory footprint was also a nice property of the transformed program (it doesn't have to allocate `temp`), particular since DRAM sizes are smaller on mobile devices. We also accepted this answer for credit.