CMU 15-418/618: Exam 2 Practice Exercises

Problem 1: Miscellaneous Short Answer

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?

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?

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?

D. If you replaced a tightly 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?

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.)

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.)

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?

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)

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)

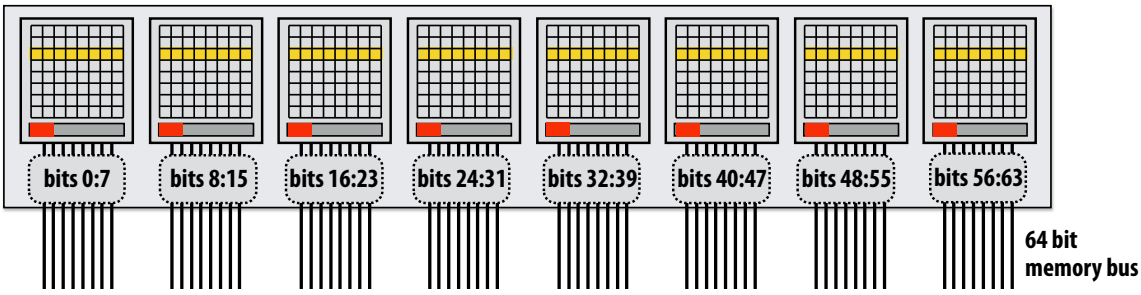
J. Consider the difference between store-and-forward and cut-through flow control in a chip interconnect. Although cut-through flow control will typically reduce packet transmission latency, its behavior degenerates to that of the store-and-forward scheme under high contention. Why?

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.)

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

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).

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

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...

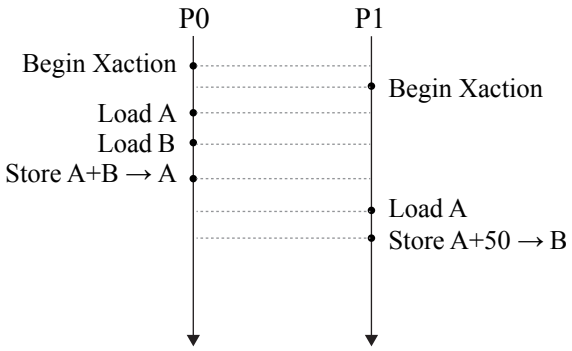
```
cycle 0: thread 0 req 0
cycle 10: thread 1 req 0
cycle 20: thread 0 req 1
cycle 30: thread 1 req 1
cycle 40: thread 0 req 2
cycle 50: thread 1 req 2
cycle 60: thread 0 req 3
...
```

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.)

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?

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)				
P1 reaches end of transaction before P0				
P0 reaches end of transaction before P1 performs loads/stores to A and B)				

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).

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?)

.

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)?

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.

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.

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.)

Problem 6: Deletion from a Binary Tree

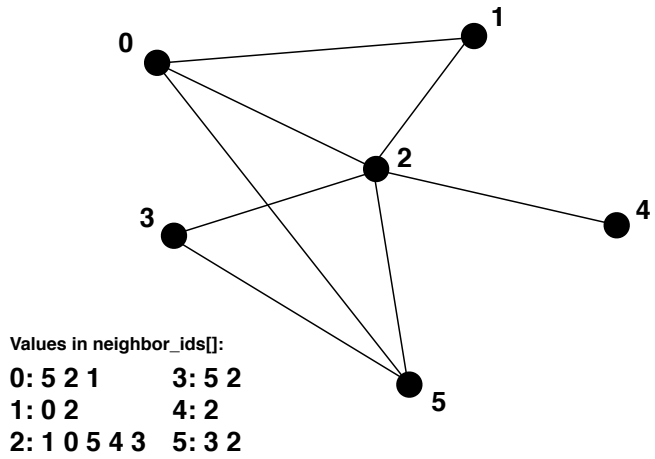
The code below, and continuing on the following two pages implements node deletion from a binary search tree. We have left space in the code for you to insert locks to ensure thread-safe access and high concurrency. You may modify the `Node` struct (below) and assume functions `lock(x)` and `unlock(x)` exist (where `x` has type `Lock`). **Your solution NEED ONLY consider the delete operation. THE ORIGINAL EXAM OMITTED ALL THE LOCKS AND UNLOCKS AND THE STUDENTS HAD TO ADD THEM.**

```
// 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;

};
```

```
// 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

   while (cur) {                         // while node not found
      if (value == cur->value) {         // found node to delete!

         // 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) {

            *prev_link = NULL;


            free(cur);
            return;
         } else if ( cur->left == NULL ) {
            // Case 2: Node has one child. Make parent node point to this child

            *prev_link = cur->right;

            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.  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 tree subtree */
            cur->value = delete_helper(cur->right, &cur->right);

            return;
         }
      } else if (value < cur->value) {
         // still searching, traverse left

         prev_link = &cur->left;
         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.
int delete_helper(Node* n, Node** prev) {

   Node** prev_link = prev;
   NODE* cur = n;

   // search for the smallest value in the tree by always traversing left
   while (cur->left) {

      prev_link = &cur->left;

      cur = cur->left;

   }

   // this is the smallest value
   int value = cur->value;

   // remove the node with the smallest value
   if (cur->right == NULL) {
      // Case 1: Node is a leaf

      *prev_link = null;

      free(cur);
   } else {
      // Case 2: Node has a right child

      *prev_link = cur->right;

      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...

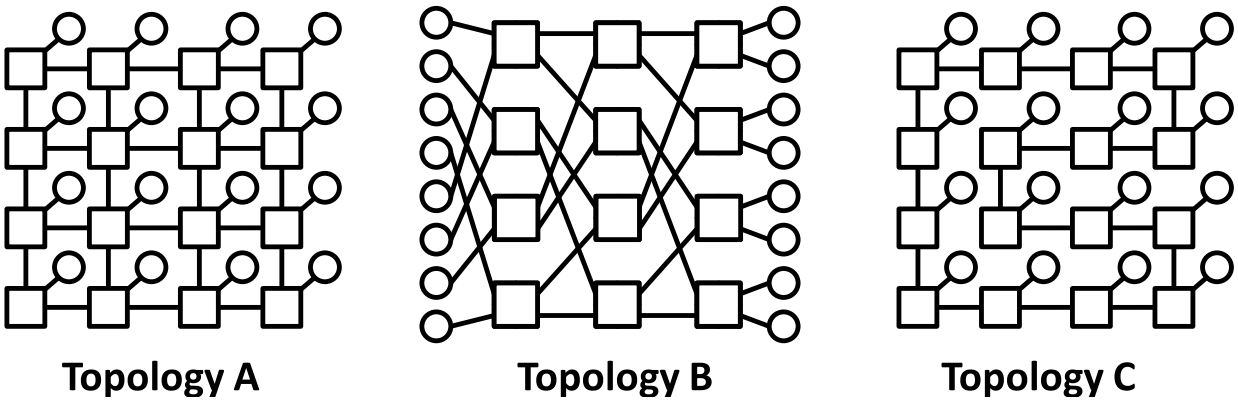
```
void update(int id) {
    Graph_node* n = &graph[id];
    LOCK(n->lock);
    for (int i=0; i<n->num_edges; i++)
      LOCK(graph[n->neighbor_ids[i]].lock);

    // 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).**

Problem 8: Interconnection Networks

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



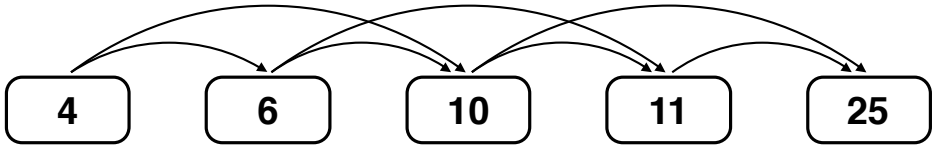
Identify each topology and fill in the table below. Express the bisection bandwidth in Gbit/s, assuming each link is 1 Gbit/s. Express the cost and latency in terms of the number of network nodes N, using Big O notation, e.g., O(logN)).

	Topology A	**Topology B**	**Topology C**
Topology Type/Name			
Direct or Indirect			
Blocking or Non-Blocking			
Bisection Bandwidth			
Cost			
Latency			

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

C. What common networking problem do virtual channels solve?

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.

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









}
```