University of California, Berkeley
College of Engineering
Computer Science Division – EECS

Spring 2014 Anthony D. Joseph

Midterm Exam #1 *Solutions*

March 12, 2014
CS162 Operating Systems

Your Name:	
SID AND 162 Login:	
TA Name:	
Discussion Section Time:	

General Information:
This is a **closed book and one 2-sided handwritten note** examination. You have 80 minutes to answer as many questions as possible. The number in parentheses at the beginning of each question indicates the number of points for that question. You should read **all** of the questions before starting the exam, as some of the questions are substantially more time consuming.

Write all of your answers directly on this paper. ***Make your answers as concise as possible.*** If there is something in a question that you believe is open to interpretation, then please ask us about it!

Good Luck!!

QUESTION	POINTS ASSIGNED	POINTS OBTAINED
1	34	
2	9	
3	15	
4	18	
5	24	
TOTAL	100	

1. (34 points total) Short answer
   a. True/False and Why? **CIRCLE YOUR ANSWER.**
      i) A fully associative cache will *always* have a higher hit rate than a direct mapped cache (on the same reference pattern).

TRUE FALSE

Why?

FALSE. *The replacement policy of cache has an effect on the hit rate –for a direct mapped cache the replacement policy is fixed because there is only one entry that a reference maps to, however for a fully associative cache the replacement policy chooses which entry to evict. So, a poor replacement policy (or bad luck for a random replacement policy) could cause a fully associative cache to perform worse than a direct mapped cache. The correct answer was worth 1 points and the justification was worth an additional 2 points.*

      ii) When the total execution times of each of the long-running (no I/O) processes waiting to be scheduled are equal, FIFO is the better choice (in terms of response time) of scheduling algorithm to use instead of round robin.

TRUE FALSE

Why?

TRUE. *Using FIFO yields a lower average response time because jobs finish sooner than with round robin, where all jobs finish in the last N timeslices. Cache benefit also. The correct answer was worth 1 point and the justification was worth an additional 2 points.*

iii) In a multi-level address translation scheme, the amount of memory used by the translation tables for each process' virtual address space is always proportional to the size of the virtual address space.

TRUE FALSE

Why?

FALSE. *The amount of memory used by the translation tables is proportional to the amount of the virtual address space that is in use (mapped) by the process. The correct answer was worth 1 points and the justification was worth an additional 2 points.*

iv) Monitors are more powerful than Semaphores because Monitors can implement solutions to synchronization problems that Semaphores cannot solve.

TRUE FALSE

Why?

FALSE. *Monitors and Semaphores have equivalent expressive power for synchronization problems – semaphores can be used to implement monitors and vice versa. The correct answer was worth 1 points and the justification was worth an additional 2 points.*

b. (8 points) Resource Allocation Tables.
Consider the following snapshot of a system with 5 processes (P1, P2, P3, P4, P5) and 4 resources (R1, R2, R3, R4). There are no outstanding queued unsatisfied requests.

	Current Allocation				Max Need				Still Needs			
Process	**R1**	**R2**	**R3**	**R4**	**R1**	**R2**	**R3**	**R4**	**R1**	**R2**	**R3**	**R4**
P1	2	0	1	4	5	3	2	4	3	3	1	0
P2	2	0	0	3	2	4	3	3	0	4	3	0
P3	0	0	3	4	2	5	5	6	2	5	2	2
P4	1	0	1	0	2	2	3	2	1	2	2	2
P5	4	1	1	0	7	3	2	2	3	2	1	2

i) Is this system currently deadlocked? Why or why not? If not deadlocked, give an execution order. Consider the currently available resources below:

Currently Available Resources

R1	R2	R3	R4
1	2	2	2

The system will be in an unsafe state according to Banker's Algorithm. P4 will be able to finish, but nothing else will be able to finish, we call this state "deadlock."

-0 Correct. It was okay to say that this was an unsafe state but that it is not necessarily deadlock because threads may not need all of the resources they declare as max needs.

-3 For saying "No there is not deadlock"

ii) If a request from a process P1 arrives for (2, 1, 1, 0), should the request be immediately granted? Why or why not? If yes, show an execution order.

Currently Available Resources

R1	R2	R3	R4
3	4	1	0

P1 will finish first. Then either P4 or P5 will finish. All schedules following are valid except those beginning with P1 P4 P3. The request made here is not in addition to nor does it replace the max needs or still needs columns.

-3 For saying it can be granted but not providing a valid schedule.

-5 For saying that it can't or shouldn't be granted for any reason.

c. (6 points) We discussed the Therac-25 in lecture and in an assigned reading.

i) Briefly explain the function of the Therac-25 and how it differs from the Therac-20?

The Therac-25 is a machine for radiation therapy that produces electron beam and X-ray radiation. It differs from the Therac-20 in that it uses only software control and safety interlocks in the electron accelerator and electron beam/X-ray production process, and software control of dosage. The Therac-20 included hardware safety interlocks.

ii) What problems did the Therac-25 introduce and what were the underlying causes.

Software errors caused the deaths and injuries of several patients. There were a series of race conditions on shared variables and poor software design that lead to the machine's malfunction under certain conditions.

d. (8 points total) Consider a demand paging system where a dedicated disk is used for paging, and all other filesystem activity uses other separate disks. Measured utilizations are:

Component	**Utilization of <u>each</u> component** (in terms of **time**, not space)
CPU	20%
Paging disk	99.7%
Other I/O devices	5%

For each of the following changes, *assume processes have similar memory usage* and CIRCLE what its likely impact will be on CPU utilization.

i) (2 points) Get a bigger paging disk.
Effect on CPU utilization (CIRCLE one of the following 3 choices):

Significantly decrease **Minimal effect** **Significantly increase**

ii) (2 points) Increase the number of running programs
Effect on CPU utilization (CIRCLE one of the following 3 choices):

Significantly decrease **Minimal effect** **Significantly increase**

iii) (2 points) Decrease the number of running programs
Effect on CPU utilization (CIRCLE one of the following 3 choices):

Significantly decrease **Minimal effect** **Significantly increase**

iv) (2 points) Get faster other I/O devices
Effect on CPU utilization (CIRCLE one of the following 3 choices):

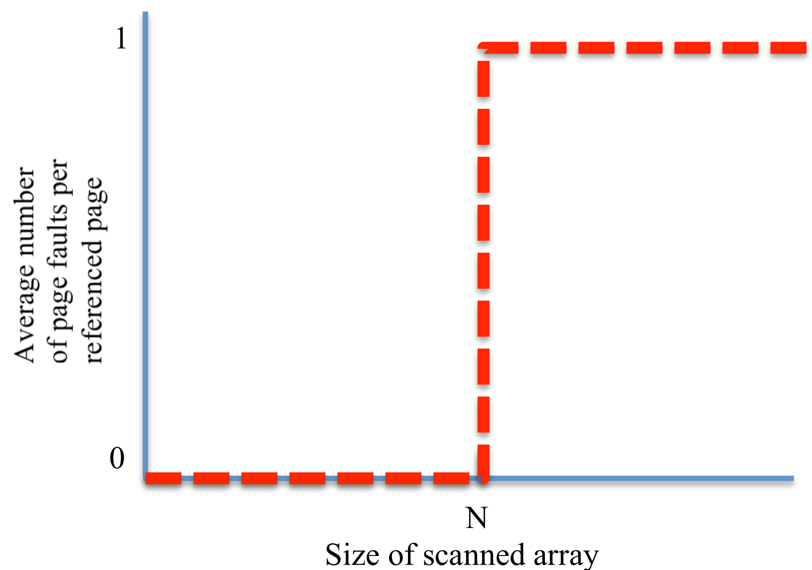
Significantly decrease **Minimal effect** **Significantly increase**

2. (9 points total) Paging

   Suppose a program that **repeatedly** scans through a *very large* array is running in virtual memory. In other words, if the array is 4 pages long, its page reference pattern is ABCDABCDABCD...

   For each of the following page replacement algorithms, sketch a graph showing the paging behavior. The y-axis of the graph is the number of page faults per referenced page (ranging from 0 to 1) and the x-axis is the size of the array being scanned (ranging from smaller than physical memory to much larger than physical memory).
   Assume the size of memory is N pages.
   Label any interesting points on each graph on both the x and y axes.

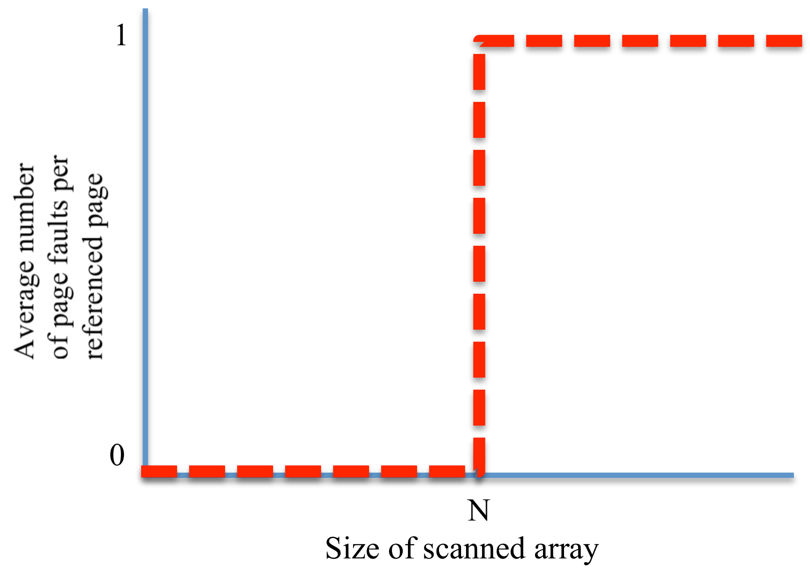
   a. (3 points) FIFO:



We expected a graph that is low (zero was accepted, but we also accepted a ramp upward due to compulsory misses) before N, then jumps suddenly to 1 after N. This is because once you are reading more than N, the cache is constantly being recycled in its entirety, so every lookup is a cache miss.

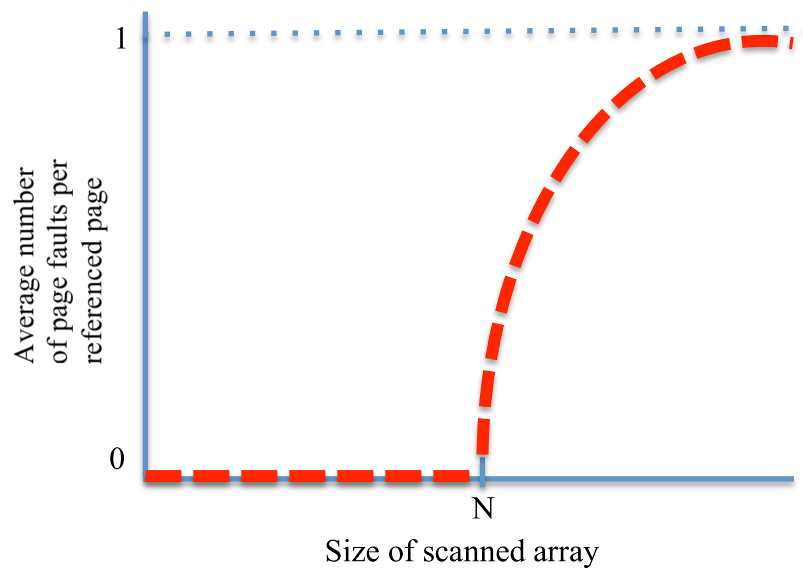
-1.5 didn't get zero or very low cache miss rate before/at N
-1.5 didn't get miss rate of 1 after N

b. (3 points) LRU:



Due to the access pattern, LRU behaves exactly the same as FIFO for this situation. Thus, we graded this graph the same as part (a).

c. (3 points) MIN:



This one was pretty tricky! Once again, there is a zero or low cache miss rate before and at N. After N, though, MIN does an interesting thing - it is smart enough to keep the first (cache_size - 1) elements in the cache, and just keeps replacing the last cache entry once there are more than (cache_size) elements in the access pattern. Thus, we get a gradual increase after N, which asymptotically approaches N.
-1 didn't get zero or very low cache miss rate before N
-1 didn't get zero or very low cache miss rate at N
-1 missed asymptotic increase after N

3. (15 points total) Memory management

Consider a multi-level memory management scheme with the following format for virtual addresses:

Virtual Page # (10 bits)	Virtual Page # (10 bits)	Offset (12 bits)

Virtual addresses are translated into physical addresses of the following form:

Physical Page # (20 bits)	Offset (12 bits)

Page table entries (PTE) are 32 bits and contain the 20-bit physical page number and OS bookkeeping bits (e.g., valid, dirty, used, etc.).

a. (2 points) How big is a page? Explain your answer.

The size of a page is $2^{(\#\ offset\ bits)}$ bytes = 2^{12} bytes = 4 KB, because each offset corresponds to a unique byte address within a page. You needed to explain your answer or show a calculation for full credit (as mentioned in the instructions).

-0 Correct

-1 For interpreting the size as an amount in bits, then converting to bytes.

-1 For minor errors, or missing/incorrect explanation

-2 Completely incorrect.

b. (2 points) What is the maximum amount of memory (in bytes) in a single virtual address space? Explain your answer.

The size of the virtual address space is $2^{(\#\ of\ total\ virtual\ address\ bits)}$ bytes = 2^{32} bytes = 4 GB, because the fields combine into a single virtual address, with each unique address corresponding to a unique byte of virtual memory. You needed to explain your answer for full credit (as mentioned in the instructions).

-0 Correct

-1 For interpreting the size as an amount in bits, then converting to bytes.

-1 For minor errors, or missing/incorrect explanation

-2 Completely incorrect.

c. (2 points) What is the maximum amount of physical memory (in bytes) that this memory management scheme supports? Explain your answer.

Similar to part (b), the size of the physical address space is 2^(# of total physical address bits) bytes = 2^32 bytes = 4 GB, because the fields combine into a single physical address, with each unique address corresponding to a unique byte of physical memory. You needed to explain your answer for full credit (as mentioned in the instructions).

-0 Correct

-1 For interpreting the size as an amount in bits, then converting to bytes.

-1 For minor errors, or missing/incorrect explanation

-2 Completely incorrect.

d. (4 points) Sketch the format of the page table for the multi-level virtual memory management scheme. Illustrate the process of resolving an address as well as possible. Assume there is no TLB or cache.

A complete, correct diagram would show the following:

- *a page table base pointer to the first page table*
- *VPN1 indexing into the first page table, to get the relevant PTE*
- *the chosen PTE pointing to the base of a second level page table*
- *VPN2 indexing into the second page table, to get the relevant PTE*
- *the chosen PTE pointing to a PPN, using the original offset to recover the physical memory address*

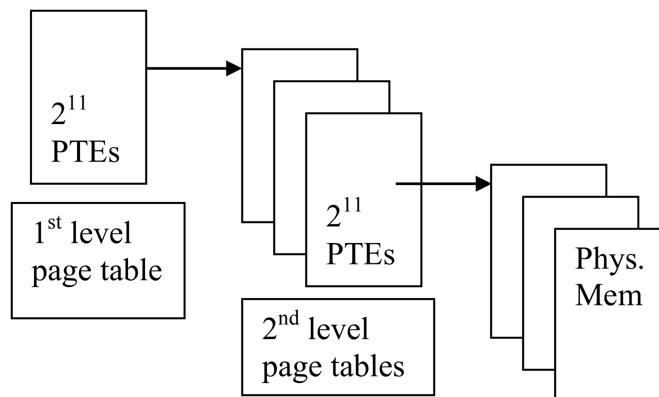
-0 Correct

-1 For minor mistakes/omissions: missing multiple pointers or labels (page table base, VPN1, VPN2, offset), missing physical memory (with PPN and offset address), placing the offset into page table entries, or using paged segmentation.

-2 For omitting actual page table entries in page tables, and failing to show a large number of the pointers or address resolution

-4 Completely incorrect.

We were looking for something like the following diagram:



e. (2 points) What is the maximum size in bytes of the *entire* multi-level data structure that maps virtual addresses to physical addresses for this scheme? Explain your answer. You may leave your answer in non-simplified form (e.g., powers of 2).

The maximum size in bytes of the entire multi-level structure is when every second level page table is used (there are no null pointers in the first level page table). There are 2^{10} entries in the first level page table, and thus 2^{10} second level page tables in total, each one page in size. There is also one page allocated for the first level page table. Maximum size is therefore $2^{12}*(1 + 2^{10})$ bytes. Due to the wording of the question, it was also acceptable to include the space allocated to the actual data pages in physical memory, which would add another $2^{12}*(2^{20})$ bytes.

-0 Correct

-1 For realizing the correct number of tables, but incorrect computation.

-2 Completely incorrect.

f. (3 points) Suppose that we have a process with a virtual address space with one physical page at the top of the address space, one physical page in the middle of the address space, and one physical page at the bottom of the address space. How big would the page table be (in bytes)? Explain your answer.

The total size of the page table would include a single first level page table, as well as three second level page tables. There would be three second level page tables because each page at the top, middle, and bottom of the virtual address space would have sufficiently different addresses (recall our address space is $2^{\wedge 32}$ bytes), and would thus point to different second level page tables after indexing into the first level with VPN1. In total, this accounts to 4 total pages, of 2^{12} bytes each, resulting to 2^{14} bytes.

-0 Correct

-0.5 For including the physical data pages in this scheme (the question only asked for the page table)

-1 For minor mistakes:

- *realizing the correct number of tables, but incorrect computation*
- *interpreting the top/middle/bottom locations to be as physical pages, and calculating 2 pages required for the page table*

-2 For more significant mistakes:

- *giving the size of a single page, rather than the entire page table structure*
- *including more than one first level page table*
- *correct intuition of 1 page per table, but using incorrect second level tables*

-3 Completely incorrect, especially only considering individual page table entries, or attempting to change the page table scheme.

4. (18 points total) Scheduling. Consider the following processes, arrival times, waiting, priority, and CPU processing requirements:

Process Name	Arrival Time	Priority	Waits on Lock Held by	Processing Time
1	0	1	None	4
2	2	20	1	3
3	5	1	None	3
4	6	10	3	2

For each scheduling algorithm, fill in the table with the process that is running on the CPU (for timeslice-based algorithms, assume a 1 unit timeslice). Notes:

- For RR and Priority, assume that an arriving thread is run at the beginning of its arrival time, if the scheduling policy allows it.
- If a thread tries to acquire a lock, it will do so at the end of its first time slice. If thread A waits for a lock (formerly) held by thread B and B has already finished, it does not wait and acquires the lock immediately.
- Assume the currently running thread is not in the ready queue while it is running.
- Turnaround time is defined as the time a process takes to complete after it arrives

Time	**FIFO**	**RR**	**Priority**
0	**1**	**1**	**1**
1	*1*	*1*	*1*
2	*1*	*2*	*2*
3	*1*	*1*	*1*
4	*2*	*1*	*1*
5	*2*	*3*	*2*
6	*2*	*4*	*2*
7	*3*	*2*	*4*
8	*3*	*3*	*4*
9	*3*	*2*	*3*
10	*4*	*3*	*3*
11	*4*	*4*	*3*
Average Turnaround Time	*((4-0)+(7-2)+(10-5)+(12-6))/ 4 =* ***5***	*((5-0)+(10-2)+ (11-5)+(12-6))/4 =* ***6.25***	*((5-0)+(7-2)+(12-5)+(9-6))/4=* ***5.0***

The original intention of this problem was that if A tries to acquire a lock that B holds, it would only do so after running for one time slice. The idea here is that if you saw "A waits on B" in the table, if A normally runs at time t, at time approaches t+1, it would try to acquire that lock, thus either acquiring the lock if B hasn't run yet or if B has already finished, or going to sleep until B finishes running. A common misinterpretation of this problem was that A was already waiting on B before it ran, so you would not see A on the table until B finished. We believed the rule "If a thread tries to acquire a lock, it will do so at the end of its first time slice" was sufficiently clear, so we gave alternate/reasonable interpretations partial credit rather than full credit.

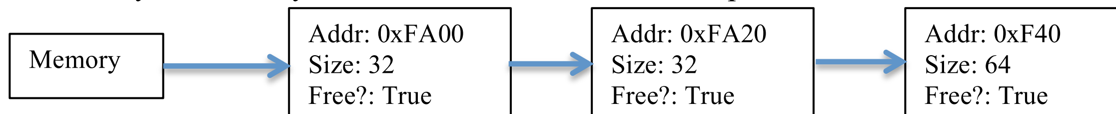
Each column was graded separately with the same breakdown of 6 points. The sequence was 4 of the 6 points and the turnaround time was 2 of the 6 points.
-0 Correct
-4 Major errors in schedule. The answer provided was not one of the common misinterpretations.
-2 Minor errors in schedule. The answer provided is one of the common misinterpretations or has a small mistake in it.
-2 Incorrect turnaround time. We did not double penalize here. If you provided the correct turnaround time according to your schedule, you received full credit for the final two points.

Problem 4.1 - FIFO
Most mistakes were in the turnaround time calculations, but otherwise the class did well and ran each process in order to completion (1, 2, 3, 4). Locks did not matter in this case because 1 and 3 finished and released locks before 2 and 4 would run.

Problem 4.2 - Round Robin
The most commonly misinterpreted solution to this problem was [1 1 1 1 2 3 2 3 2 3 4 4 5.5].

Problem 4.3 - Priority
The most commonly misinterpreted solution to this problem was [1 1 1 1 2 2 2 3 3 3 4 4 5.0].
We also accepted [1 1 2 1 1 2 2 4 3 3 3 4 5.5] as a fully correct solution.

5. (24 points total) Synchronization – a thread safe memory allocator
A memory allocator is a userspace library that helps user processes acquire memory using methods called `malloc` and `free`. In this problem we assume the allocator only returns chunks sized in powers of two (i.e., it gives out 32 (2^5) bytes, 64 (2^6) bytes, 128 (2^7) bytes, etc.). To manage the allocated and unallocated virtual memory, the allocator maintains a linked list of nodes where each node corresponds to a block of memory, which may be free or used. Here is an example of the linked list:



Note that adjacent nodes in this linked list are also adjacent in memory. To avoid wasting memory, the allocator will split a node into smaller nodes when a smaller size is asked for than available chunks – if a thread requests 32 bytes and only one 64 byte node is available, the allocator will split that node into two 32 byte chunks and give the user one 32 byte chunk.

What we want is an algorithm that combines (from left to right) adjacent free blocks, whose sizes add up to at least the requested size. The following simple algorithm solves this problem, but is not thread safe (guaranteed to operate correctly in the presence of requests from multiple threads).

```
def malloc(size):
   for node in malloc_list:
         if (node.size >= size and node.free):
             # break node into multiple nodes if necessary
             break_apart_node(node, size)
             node.free = false
             return node.address

def free(ptr):
   for node in malloc_list:
         if (node.address == ptr):
             node.free = true

def combine_left_to_right(node, begin_size):
   if (not node->free):
         return false
   if (combine_left_to_right(node->next, node->next->size
                                         + begin_size)):
         node->size += node->next->size
         node->next = node->next->next
         return 1

   return is_power_of_two(node->next->size + begin_size)
```

<u>Note that a background thread periodically iterates through the linked list and calls</u> `combine_left_to_right(node, 0)` <u>for every node.</u>

Make the above code thread safe by using locks.

```
def malloc(size):
   for node in malloc_list:
         if (node.size >= size and node.free):
             # break node into multiple nodes if necessary
             break_apart_node(node, size)
             node.free = false
             return node.address
```

a. (5 points) Thread safe `malloc`:

The goal of this question was to make sure you understood how locks work and what would be an important resource that should only be accessed one at a time. In this case, there was one lock for the entire `malloc_list` *and this means you should just lock at the top of the function. And make sure you release the lock before each of the two returns.*

-0 Correct
-2 Forget one of the two possible releases (more commonly the bottom one)
-2 Acquire in the wrong place, must be above the if statement. Time to check problems
-2 Too many releases when only acquired it once.
-4 Did not use locks
-5 Did not attempt the question

```
def free(ptr):
   for node in malloc_list:
         if (node.address == ptr):
             node.free = true
```

b. (5 points) Thread safe `free`:

The original function given for the `free` *was already thread safe. This is because for the functions that we have given we always check the address first. Since we didn't explicitly state what the behavior is if you free the same pointer twice, we did not dock points off for this.*
-0 Correct
-3 If you used locks and made it deadlock. Most of the mistakes happened when people try to release a lock in the wrong place based off of where they acquired it.
-4 If you attempted to use something that isn't a lock or changed the functionality of the function.
-5 Incorrect.

```
def combine_left_to_right(node, begin_size):
   if (not node->free):
         return false
   if (combine_left_to_right(node->next, node->next->size
                                               + begin_size)):
         node->size += node->next->size
         node->next = node->next->next
         return 1

   return is_power_of_two(node->next->size + begin_size)
```

c. (8 points) Thread safe `combine_left_to_right`:

Note that you may want to restructure the code.
There were three main different ways of solving it. The main thing that you would have to notice is that you can't just put locks at the top and bottom of the function like the previous two problems. If you did that then since it is a recursive call, it would try to acquire a lock that it has already acquired and it will deadlock. There are a couple ways of getting around this problem. The three possible ways of solving it is through recursion, iterative, or creating a helper function.

Recursive:
This solution is the most tricky one. This is due to the fact that you have to make sure you protect the entire list and if a `malloc` comes and context switches and gets to your node while you're coalescing it won't work. To be able to do this, you have to make sure that acquire locks in the right place. The portion that you have to lock is above the if statement that calls the `combine_left_to_right()` call. While you lock it, you have to make sure that you keep a temp of the next node. And set it to not free, so that the `malloc` won't try to `malloc` it out. Then, you have to make sure you make `node.next` = to `NULL` and this will make sure that the `malloc` won't go further. Then after that, do the combine call. If the combine call returns true, then we have to make sure we lock the nodes before we do any of that. Then we have to make sure we restore the current node's next to the next correct one and return the `is_power_of_two()`.

```
def combine_left_to_right(node, begin_size):
      lock.acquire()
      if (not node.free):
            lock.release()
            return False
      temp = node.next
      node.next = None
      node.free = False
      lock.release()
      if (combine_left_to_right(temp, node.size + begin_size)):
            lock.acquire()
            node.size += temp.size
            node.next = temp.next
            node.free = True
            lock.release()
            return True
      lock.acquire()
```

```
        node.next = temp
        node.free = True
        return_value = is_power_of_two(node.size + begin_size)
        lock.release()
        return return_value
```

Iterative:
Another way is to lock the whole function. And then do it iteratively with a while loop. But, you have to make sure that you restore the correct list back to itself. So to be able to unravel the while loop, you have to make sure that you have a stack that you push onto. After you get to where you can't combine anymore, and then you do the normal checks by checking each one on the stack until the stack is empty and combine any that is possible. Then, make sure you release the lock before you return.

```
def combine_left_to_right(node, begin_size):
      lock.acquire()
      s = new Stack()
      while(node.free and node.next != None):
            s.push(node)
            node = node.next
      while(not s.empty()):
            node = s.pop()
            if (is_power_of_two(node.size + node.next.size):
                  node.size += node.next.size
                  node.next = node.next.next
      lock.release()
```

Helper Function:
This is the simplest solution out of the three. All you have to do is make sure you a helper function that calls the original function. Then put locks around the call, so you just have to lock once. This is the one we hoped students would do. Here is a sample solution.

```
def combine_left_to_right_helper(node, begin_size):
      same code as original combine_left_to_right()

def combine_left_to_right(node, begin_size):
      lock.acquire()
      combine_left_to_right_helper(node, begin_size)
      lock.release()
```

-0 Correct
-2 Iterative without the stack since it won't be the correct functionality.
-2 Recursive call doesn't protect the link list while calling the functions.
-2 One of the solutions, but doesn't end up being the correct functions.
-8 Doesn't understand why it deadlocks because of recursive calls

d. (3 points). Your friend Alice suggests you use **recursive locks**, which are like normal locks except that the same thread can acquire a recursive lock multiple times *without blocking* (any other thread will block as usual). However, a recursive lock must be released as many times as it is acquired. Make the original `combine_left_to_right` thread safe using recursive locks:

This problem was designed to help you figure out what was wrong with the problem before. You have to figure out that when you call a lock within a recursive call it will deadlock because it can't acquire a normal mutex that has already been acquired. The use of the recursive lock essentially in the locking portion checks if the thread id using the lock is the same thread id as the one that is currently holding it. If it is, then you are able to acquire the lock multiple times and then a count goes up. So, this means that we can just use the original code that was given for `combine_left_to_right()` and just put a lock at the top and bottom of the call.

-0 Correct

-1 Acquire needs to be above the first if. If it isn't, then there could be a difference between when you check and when you actually act on it.

-1 Will Deadlock if you miss a release in all the exit conditions

-1 For anything that isn't thread safe.

-3 Blank or wrong

e. (3 points). Your friend Charlie says that **recursive locks** are just like Semaphores and you can simply replace all the calls to `Acquire` with `P()` and `Release` with `V()`. Is Charlie correct? If so, justify his claim, if not explain why.

Correct?

NO

Justification (in four sentences or less):

You cannot use semaphores as a replacement for the recursive lock. The main thing that we were looking for is that semaphores don't have a sense of ownership. Another important thing to note is that it will only actually lock when the semaphore hits 0. If it doesn't hit zero, then any other thread who tries and acquire it will be able to acquire it. Since if other threads are able to acquire this, then this is not a viable replacement for a recursive lock.

-0 Correct
-1 Wrong justification of a recursive lock.
-3 Wrong answer or did not answer

This page intentionally left blank as scratch paper

Do not write answers on this page

This page intentionally left blank as scratch paper

Do not write answers on this page