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

Fall 2010 John Kubiatowicz

Midterm I
SOLUTIONS

October 18th, 2010

CS162: Operating Systems and Systems Programming

Your Name:	
SID Number:	
Circle the letters of CS162 Login	First: a b c d e f g h I j k l m n o p q r s t u v w x y z Second: a b c d e f g h I j k l m n o p q r s t u v w x y z
Discussion Section:	

General Information:
This is a **closed book** exam. You are allowed 2 pages of notes (both sides). You may use a calculator. You have 3 hours to complete as much of the exam as possible. Make sure to read all of the questions first, 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.* On programming questions, we will be looking for performance as well as correctness, so think through your answers carefully. If there is something about the questions that you believe is open to interpretation, please ask us about it!

Problem	**Possible**	**Score**
1	16	
2	22	
3	20	
4	24	
5	18	
Total	100	

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899

Problem 1: True/False [16 pts]

Please *EXPLAIN* your answer in TWO SENTENCES OR LESS (Answers longer than this may not get credit!). Also, answers without an explanation *GET NO CREDIT*.

Problem 1a[2pts]: The size of an inverted page table grows with the size of the virtual address space that it is supporting.

True / False

Explain: *Since an inverted page table holds a* hash *on entries, it only needs to grow with the size of the physical address space. Entries not found in the inverted page table can be assumed to be invalid.*

Problem 1b[2pts]: The term "Core" in "Dump Core" refers to the interior of a machine, since memories used to be in the middle of room-sized computers.

True / False

Explain: *The term "Core" refers to "core memory", a type of memory that uses magnetic rings (cores) to store bits of information.*

Problem 1c[2pts]: Threads within the same process can share data with one another by passing pointers to objects on their stacks.

True / False

Explain: *Since the threads in the same process share an address space, they can all access the same memory (including each other's stacks).*

Problem 1d[2pts]: Resource cycles always lead to deadlock.

True / False

Explain: *No. If there are multiple equivalent resources, then a cycle could exist that wasn't a deadlock: The reason is that some thread that wasn't a part of the cycle could release a resource needed by a thread in the cycle, thereby breaking the cycle.*

Problem 1e[2pts]: Anything that can be done with a monitor can also be done with semaphores.

True / False

Explain: *Since one can construct monitors using semaphores, one could implement any monitor-based algorithm with semaphores by simply replacing the monitor implementation (i.e. locks and condition variables) with one using semaphores.*

Problem 1f[2pts]: **"**Hyperthreading" is a term used to describe systems with thousands of threads.

True / False

Explain: *Hyperthreading refers to a hardware-multithreading technique that interleaves a small number of threads on a cycle-by-cycle basis to maximize the utilization of the processor pipeline.*

Problem 1g[2pts]: A Lottery Scheduler can be used to implement any other scheduling algorithm by adjusting the number of tickets that each process holds.

True / False

Explain: *A lottery scheduler cannot be used to implement strict priority scheduling.*

Problem 1h[2pts]: A *MicroKernel* can improve the resilience of a system against bugs in the OS.

True / False

Explain: *By isolating components of the system (such as file system, network stack, etc.) in their own address spaces, we gain resilience against bugs because buggy components are prevented from interfering with other components by the address-space isolation.*

Problem 2: Short Answer [22pts]

Problem 2a[2pts]: What is priority donation and why is it important?

Priority donation is the process of promoting the priority of lock holders to be equal to that of the corresponding lock-waiters in order to prevent priority inversion (a situation in which a high-priority process ends up waiting on a low-priority process). Priority donation helps to avoid variety of bad situations that can arise from priority inversion such as livelocks or other violations of resource usage that result from not adhering to the priority scheme.

Problem 2b[3pts]: Name three ways in which the processor can transition from user mode to kernel mode. Can the user execute arbitrary code after transitioning?

Some examples include (you only need three):

(1) User executing a yield() system call

(2) User executing an arbitrary system call (such as for I/O)

(3) External interrupt (such as a timer interrupt)

(4) System fault (such as a divide-by-zero or memory error).

(5) Page Fault

No, the user is not able to execute arbitrary code, because all of the entrypoints to the kernel are carefully enforced by the hardware. To say this another way, the hardware makes sure that all transitions to kernel mode are accompanied by entry into a well-defined set of entry points.

Problem 2c[2pts]: What happens when an interrupt occurs? What does the interrupt controller do?

Assuming that the interrupt is currently enabled, the hardware saves the PC, disables interrupts, then jumps to an interrupt vector based on the type of interrupt. Typically, the interrupt handler adjusts which interrupts are enabled and saves all registers (including the stack pointer) before proceeding to service the interrupt. The interrupt controller provides an interface that allows the kernel to control which interrupts are enabled.

Problem 2d[2pts]: Explain how to fool the multi-level feedback scheduler's heuristics into giving a long-running task more CPU cycles.

The multi-level feedback scheduler uses the time between bursts (i.e. between I/O operations) to distinguish between interactive and background tasks. Long-running tasks are typically run at lower priority. By inserting a bunch of I/O (say printfs) into the long-running task, one can fool the multi-level feedback scheduler into thinking that the long-running tasks is actually interactive and should be run at higher priority.

Problem 2e[3pts]: What is the difference between Mesa and Hoare scheduling for monitors? Include passing of locks between signaler and signalee, scheduling of CPU resources, and impact on the programmer.

For Mesa scheduling, the signaler keeps the lock and CPU, while the signaled thread is simply put on the ready queue and will run at a later time. Further, a programmer with Mesa scheduled monitors must recheck the condition after being awoken from a Wait() operation [i.e. they need a while loop around the execution of Wait(). For Hoare scheduling, the signaler gives the lock and CPU to the signaled thread which begins running until it releases the lock, at which point the signaler regains the lock and CPU. A programmer with Hoare scheduled monitors does not need to recheck the condition after being awoken, since they know that the code after the Wait() is executed immediately after the Signal() [i.e. they do not need a while loop around the execution of Wait()].

Problem 2f[2pts]: What needs to be saved and restored on a context switch between two threads in the same process? What if the two threads are in different processes? Be explicit.

Need to save the processor registers, stack pointer, program counter into the TCB of the thread that is no longer running. Need to reload the same things from the TCB of the new thread. When the threads are from different processes, need to not only save and restore what was given above, but you also need to load the pointer for the top-level page-table of the new address space. You don't need to save the old pointer, since this will not change and is already stored in the PCB.

Problem 2g[2pts]: List two reasons why overuse of threads is bad (i.e. using too many threads for different tasks). Be explicit in your answers.

There are many reasons why overusing threads may be bad. For instance: (1) the overhead of switching will become significant, (2) you would end up using a lot of memory for stacks, (3) the threads may spend all of their time synchronizing on shared data structures (i.e. sleeping in locks), (4) You may be more likely to introduce concurrency bugs with increased concurrency.

Problem 2h[2pts]: As specified, the SRTF algorithm requires knowledge of the future. Name two ways to approximate the information required to implement this algorithm.

(1) One can use the past to predict the future by building a model of burst time (e.g. a Kalmann filter or an exponential-averaging filter).

(2) *One could build a scheduler structure (such as the multi-level scheduler) that attempts to sort threads into a small number of categories based on their past behavior and using these categories to subsequently assign processor time (i.e. schedule) these threads.*

Problem 2i[4pts]:
Here is a table of processes and their associated arrival and running times.

Process ID	Arrival Time	CPU Running Time
Process 1	0	2
Process 2	1	6
Process 3	4	1
Process 4	7	4
Process 5	8	3

Show the scheduling order for these processes under 3 policies: First Come First Serve (FCFS), Shortest-Remaining-Time-First (SRTF), Round-Robin (RR) with timeslice quantum = 1. *Assume that context switch overhead is 0 and that new processes are added to the* ***head*** *of the queue except for FCFS, where they are added to the tail.*

Time Slot	FCFS	SRTF	RR
0	*1*	*1*	*1*
1	*1*	*1*	*2*
2	*2*	*2*	*1*
3	*2*	*2*	*2*
4	*2*	*3*	*3*
5	*2*	*2*	*2*
6	*2*	*2*	*2*
7	*2*	*2*	*4*
8	*3*	*2*	*5*
9	*4*	*5*	*2*
10	*4*	*5*	*4*
11	*4*	*5*	*5*
12	*4*	*4*	*2*
13	*5*	*4*	*4*
14	*5*	*4*	*5*
15	*5*	*4*	*4*

[This page intentionally left blank]

Problem 3: Readers-Writers Access to Database [20 pts]

<pre>Reader() {
   //First check self into system
   lock.acquire();
   while ((AW + WW) > 0) {
      WR++;
      okToRead.wait(&lock);
      WR--;
   }
   AR++;
   lock.release();
 	
   // Perform actual read-only access
   AccessDatabase(ReadOnly);
 	
   // Now, check out of system
   lock.acquire();
   AR--;
   if (AR == 0 && WW > 0)
      okToWrite.signal();
   lock.release();
}</pre>
	<pre>Writer() {
	   // First check self into system
	   lock.acquire();
	   while ((AW + AR) > 0) {
	      WW++;
	      okToWrite.wait(&lock);
	      WW--;
	   }
	   AW++;
	   lock.release();
	
	   // Perform actual read/write access
	   AccessDatabase(ReadWrite);
	
	   // Now, check out of system
	   lock.acquire();
	   AW--;
	   if (WW > 0){
	      okToWrite.signal();
	   } else if (WR > 0) {
	      okToRead.broadcast();
	   }
	   lock.release();
	}</pre>

Problem 3a[2pts]: Above, we show the Readers-Writers example given in class. It used two condition variables, one for waiting readers and one for waiting writers. Suppose that *all* of the following requests arrive in very short order (while R_1 and R_2 are still executing):

Incoming stream: R_1 R_2 W_1 R_3 W_2 W_3 R_4 R_5 R_6 W_4 R_7 W_5 W_6 R_8 R_9 W_7 R_{10}

In what order would the above code process the above requests? If you have a group of requests that are equivalent (unordered), indicate this clearly by surrounding them with braces '{}'. You can assume that the wait queues for condition variables are FIFO in nature (i.e. signal() wakes up the oldest thread on the queue). Explain how you got your answer.

SOLN: Processed as follows: {R_1 R_2} $W_1$$W_2$ W_3 W_4 W_5 W_6 W_7 { R_3 R_4 R_5 R_6 R_7 R_8 R_9 R_{10}}

Since this algorithm gives precedence to writes over reads, although the first two reads get to execute together and right away, the remaining reads are deferred until each write is processed (one at a time). After all writes are finished, then the reads execute as a group.

Problem 3b[2pts]: Let us define the *logical arrival order* by the order in which threads first acquire the monitor lock. Suppose that we wanted the results of reads and writes to the database to be the same as if they were processed one at a time – in their logical arrival order; for example, W_1 W_2 R_1 R_2 R_3 W_3 W_4 R_4 R_5 R_6 would be processed as W_1 W_2 {R_1 R_2 R_3} W_3 W_4 {R_4 R_5 R_6} *regardless of the speed with which these requests arrive.* Explain why the above algorithm does not satisfy this constraint.

Once it starts buffering items (i.e. putting requests to sleep on condition variables), the above algorithm does not keep track of the arrival order. Or, to say it another way, the above algorithm loses all ordering between buffered reads and buffered writes. Thus, it cannot guarantee that reads execute after earlier writes and before later writes (as defined by logical arrival order) unless it never buffers items (in which case it is not really doing anything at all).

Assume that our system uses Mesa scheduling and that condition variables have FIFO wait queues. Below is a sketch of a solution that uses only two condition variables and that *does* return results as if requests were processed in logical arrival order. Rather than separate methods for Reader() and Writer(), we have a single method which takes a “NewType” variable (0 for read, 1 for write):

```
 1. Lock MonitorLock;             // Methods: acquire(),release()
 2. Condition waitQueue, onDeckQueue;// Methods: wait(),signal(),broadcast()
 3. int Queued = 0, onDeck = 0;   // Counts of sleeping threads
 4. /* Missing code */

 5. Accessor (int NewType) {      // type = 0 for read, 1 for write
 6.    /* Monitor to control entry so that one writer or multiple readers */
 7.    MonitorLock.acquire();
 8.    /* Missing wait condition */
 9.    {  Queued++;
10.       waitQueue.wait();
11.       Queued–;
12.    }
13.    /* Missing wait condition */
14.    {  onDeck++;
15.       onDeckQueue.wait();
16.       onDeck--;
17.    }
18.    /* Missing code */
19.    MonitorLock.release();

20.    // Perform actual data access
21.    AccessDatabase(NewType);

22.    /* Missing code */
23. }
```

The `waitQueue` condition variable keeps *unexamined* requests in FIFO order. The `onDeckQueue` keeps a *single* request that is currently incompatible with requests that are executing. W*e want to allow as many parallel requests to the database as possible, subject to the constraint of obeying logical arrival order*. Here, logical arrival order is defined by the order in which requests acquire the lock at line #7.

Problem 3c[2pts]: What additional variable(s) (and initialization) do you need at Line #4? *Hint: think about what you need to describe the current set of threads accessing the database in parallel.* Give explicit code to insert at Line #4 (can be more than one line of actual code).

We need to know which type of request is currently occurring and how many of these requests are occurring in order to adhere to the Readers-Writers conditions (multiple reads or one write at a time). Although we can accomplish this requirement in multiple ways, the following works:

```
int CurType = 0;
int NumAccessing = 0;
```

Problem 3d[2pts]: Explain why you might not want to use a “while()” statement in Line #8 – *despite the fact that the system has Mesa scheduling*:

Because a while loop will potentially reorder requests on the `waitQueue` *condition variable – and we want to keep requests in their original logical arrival order. Thus whatever we do, we do not want to execute* `waitQueue.wait()` *more than once per thread.*

Problem 3e[2pts]: What is the missing code in Line #8? *Hint: it is a single "if" statement.*

If anyone in the system is waiting, then we want to wait so that we don't get ahead of others (i.e. so that we preserve the logical arrival order):

```
if (Queued + onDeck > 0)
```

Problem 3f[2pts]: What is the missing code in Line #13? *This should be a single line of code.*

If the current request is incompatible with the set of executing requests, then we have to wait:

```
while ((NumAccessing > 0) && (NewType == 1 || CurType == 1))
```

You could also use an "if" here, but then you would need to conditionalize the "onDeckQueue.signal()" statement in (3h).

Problem 3g[2pts]: What is the missing code in Line #18? *You should not have more than three (3) actual lines of code.*

```
NumAccessing++;       // Have another thread accessing the database
CurType = NewType;    // In case we are first of new group
waitQueue.signal();   // Wake up next thread (if they exist)
```

Problem 3h[3pts]: What is the missing code in Line #22? *You should not have more than five (5) actual lines of code.*

```
MonitorLock.acquire();
NumAccessing--;       // One less accessing
onDeckQueue.signal(); // Wake up someone stalled on conflict
MonitorLock.release();
```

If you use an "if" in (3h), then need this instead of the above signaling

```
if(NumAccessing ==0)
 onDeckQueue.signal(); // Wake up someone stalled on conflict
```

Problem 3i[3pts]: Suppose that condition variables did not have FIFO ordered wait queues. What changes would you have to make to your code? Be explicit (you should not need more than 6 new or changed lines). *Hint: consider assigning a sequentially increasing ID to each thread.*

New variables to track sequential IDs added to Line #4:

```
int HeadID = 0, TailID = 0;   //(NEW)
```

New code for Line #8 (one new line, one changed line):

```
int MyCurrentID = TailID++;   //(NEW)Next logical ID to new request
while (MyCurrentID != HeadID) //(CHANGED)
```

Finally, code for Line #18 is slightly different (only added 1 line, changed 1 line):

```
NumAccessing++;         //(UNCHANGED LINE)
CurType = NewType;      //(UNCHANGED LINE)
HeadID++;               //(NEW) Next in-order thread can wake
waitQueue.broadcast(); //(CHANGED) Wake up next thread in order
```

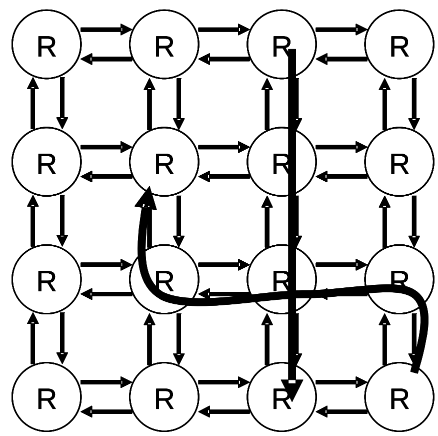
[This page intentionally left blank]

Problem 4: Deadlock [24pts]

Problem 4a[2pts]: Suppose that we utilize the Banker's algorithm to determine whether or not to grant resource requests to threads. The job of the Banker's algorithm is to keep the system in a "SAFE" state. It denies resource requests by putting the requesting thread to sleep if granting the request would cause the system to enter an "UNSAFE" state, waking it only when the request could be granted safely. What is a SAFE state?

In a safe state, there is some ordering of the threads in the system such that threads can complete, one after another without deadlocking and without requiring threads to give up resources that they already have.

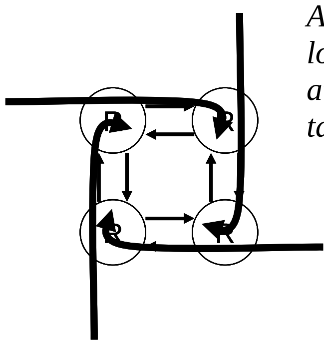
Problem 4b[4pts]:



The figure at the right illustrates a 2D mesh of network routers. Each router is connected to each of its neighbors by two network links (small arrows), one in each direction. Messages are routed from a source router to a destination router and can stretch through the network (i.e. consume links along the route from source to destination). Messages can cross inside routers.

Assume that no network link can service more than one message at a time, and that each message must consume a continuous set of channels (like a snake). Messages always make progress to the destination and never wrap back on themselves. The figure shows two messages (thick arrows).

Assume that each router or link has a very small amount of buffer space and that each message can be arbitrarily long. Show a simple situation (with a drawing) in which messages are deadlocked and can make no further progress. Explain how each of the four conditions of deadlock are satisfied by your example. *Hint: Links are the limited resources in this example.*



Answer: The simplest deadlock example is a set of four messages in a loop (as shown in the figure at the left). Each message is blocked attempting to make a counter-clockwise turn by a message utilizing the target channel. Four conditions:

1. *Mutual Exclusion: Each channel held by one message at a time*
2. *Hold and Wait: messages hold channels while waiting to acquire other channels*
3. *No preemption: Channels can not be preempted from messages after they are acquired by them*
4. *Circular wait: We have a cycle of waiting here – four messages*

Problem 4c[3pts]:
Define a routing policy that avoids deadlocks in the network of (4b). Prove that deadlocks cannot happen when using this routing policy. *Hint: assume that a deadlock occurs and show why that leads to a contradiction in the presence of your new routing policy.*

Several answers are possible here. The one given in class was to force messages to route in the X direction first, then Y. To prove that deadlocks cannot occur, assume that one did occur. Then, such a deadlock would involve a cycle in messages. However, such a cycle would involve at least one message that routed from the Y direction to the X direction (in fact, at least 2 of them). Since we have disallowed such a route, this cycle cannot exist. Thus, no deadlocks can exist.

Problem 4d[3pts]: Suppose that we wanted a less restrictive routing policy than your answer to (4c). Explain why the Banker's algorithm is not well adapted to this routing problem. What could you do that was similar in spirit to the Banker's algorithm to prevent deadlock in networks such as (4b), while allowing arbitrary routing of messages? Why is it unlikely that such an algorithm would be used in a real network?

The problem here is that the resource needs of a message depends on the current location of its head, while the Banker's algorithm requires declaration of resource needs prior to routing. The Banker's algorithm is so conservative that it would allow only a small number of messages to be present in the network at the same time.

What one could do is to have each message request its next link from a deadlock-avoider that checks inflight messages to see if they could still be drained from the network if the link request were granted. Although this mechanism would be in the spirit of the Banker's algorithm, it is very expensive (requires essentially routing all messages to their destinations to decide whether the system would deadlock).

Problem 4e[2pts]: Is it possible for a system with a single monitor (i.e one lock with multiple condition variables) to deadlock? Explain.

Yes. The lock provides a guard on the condition variables, which could be involved in a circular wait condition. The simplest way to see this situation is to construct two semaphores from one monitor with two condition variables. For instance:

```
X.p()⇒  lock.acquire();                X.v()⇒  lock.acquire();
        while (Xcount = 0)                     Xcount++;
            Xcondition.wait();                 Xcondition.signal()
        Xcount–;                               lock.release();
        lock.release();
```

Same for semaphore Y. Then, set up a circular wait condition:

Thread 1	*Thread 2*
X.p()	*Y.p()*
Y.p()	*X.p()*

Problem 4f[4pts]:
Suppose that we have the following resources: A, B, C and threads T1, T2, T3, T4. The total number of each resource is:

Total		
A	B	C
12	9	12

Further, assume that the processes have the following maximum requirements and current allocations:

Thread ID	Current Allocation			Maximum		
	A	B	C	A	B	C
T1	2	1	3	4	9	4
T2	1	2	3	5	3	3
T3	5	4	3	6	4	3
T4	2	1	2	4	8	2

Is the system in a safe state (as defined by the Banker's algorithm)? If "yes", show a non-blocking sequence of thread executions. Otherwise, provide a proof that the system is unsafe. Show all steps, intermediate matrices, etc.

Answer: Yes, this system is in a safe state.

To prove this, we first compute the currently free allocations:

Available		
A	*B*	*C*
2	*1*	*1*

Further, we compute the number needed by each thread (Maximum – Current Allocation):

Thread ID	*Needed Allocation*		
	A	*B*	*C*
T1	*2*	*8*	*1*
T2	*4*	*1*	*0*
T3	*1*	*0*	*0*
T4	*2*	*7*	*0*

Thus, we can see that a possible sequence is: T3, T2, T4, T1:

Thread ID	*Needed Allocation*			*Current Allocation*			*Available Before*		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
T3	*1*	*0*	*0*	*5*	*4*	*3*	*2*	*1*	*1*
T2	*4*	*1*	*0*	*1*	*2*	*3*	*7*	*5*	*4*
T4	*2*	*7*	*0*	*2*	*1*	*2*	*8*	*7*	*7*
T1	*2*	*8*	*1*	*2*	*1*	*3*	*10*	*8*	*9*

Problem 4g[3pts]:
Assume that we start with a system in the state of (4f). Suppose that T1 asks for 2 more copies of resource A. Can the system grant this if it wants to avoid deadlock (i.e. will the result be a SAFE state)? Explain.

No. This cannot be granted. Assume that T1 gets 2 more of A.
Then, our available allocation is:

Available		
A	*B*	*C*
0	*1*	*1*

Then, looking at our needed allocations, we see:

Thread ID	*Needed Allocation*		
	A	*B*	*C*
T1	*0*	*8*	*1*
T2	*4*	*1*	*0*
T3	*1*	*0*	*0*
T4	*2*	*7*	*0*

At this point, the available allocation is insufficient to start any of the threads, much less find a safe sequence that finishes all of them.

Problem 4h[3pts]:
Assume that we start with a system in the state of (4f). What is the maximum number of additional copies of resources (A, B, and C) that T1 can be granted in a single request without risking deadlock? Explain.

We cannot ask for more than (A,B,C)=(2,1,1) since this is all that is available. However, the previous problem showed that A must be < 2. Further, note that the resources given to T1 are tied up until the very end of the execution (look at sequence in 4f).

Thus, looking at our safe sequence from 4f, we can see that it can still work if it is missing one A and one C. Just work through it with the first "Available Before" allocation of 1,1,0 instead of 2,1,1. However, it will not work if it is missing one more B (we would be unable to execute T4 in the sequence), i.e. setting "Available Before" to 1,0,0 prevents the execution of T4.

Thus the maximum number of additional resources that can be requested by T1 is (A,B,C)=(1,0,1)

Problem 5: Virtual Memory [18pts]

Consider a two-level memory management scheme on 24-bit virtual addresses using the following format for virtual addresses:

Virtual Page # (8 bits)	Virtual Page # (8 bits)	Offset (8 bits)

Virtual addresses are translated into 16-bit physical addresses of the following form:

Physical Page # (8 bits)	Offset (8 bits)

Page table entries are 16 bits in the following format, *stored in big-endian form* in memory (i.e. the MSB is first byte in memory).

Page Table Entry (PTE)

Physical Page # (8 bits)	Kernel Only	Uncacheable	0	0	Dirty	Use	Write	Valid

Note that a virtual-physical translation can fail at any point if an incompatible PTE is encountered. Two types of errors can occur during translation: "invalid page" (page is not mapped at all) or "access violation" (page exists, but access was illegal).

Problem 5a[2pts]: How big is a page? Explain.

Pages are 256 bytes in size $(2^{Offset}) = (2^8)$.

Problem 5b[2pts]: What is the largest size for a page table with this address space? We are asking for the total size of both levels of the page table. Explain.

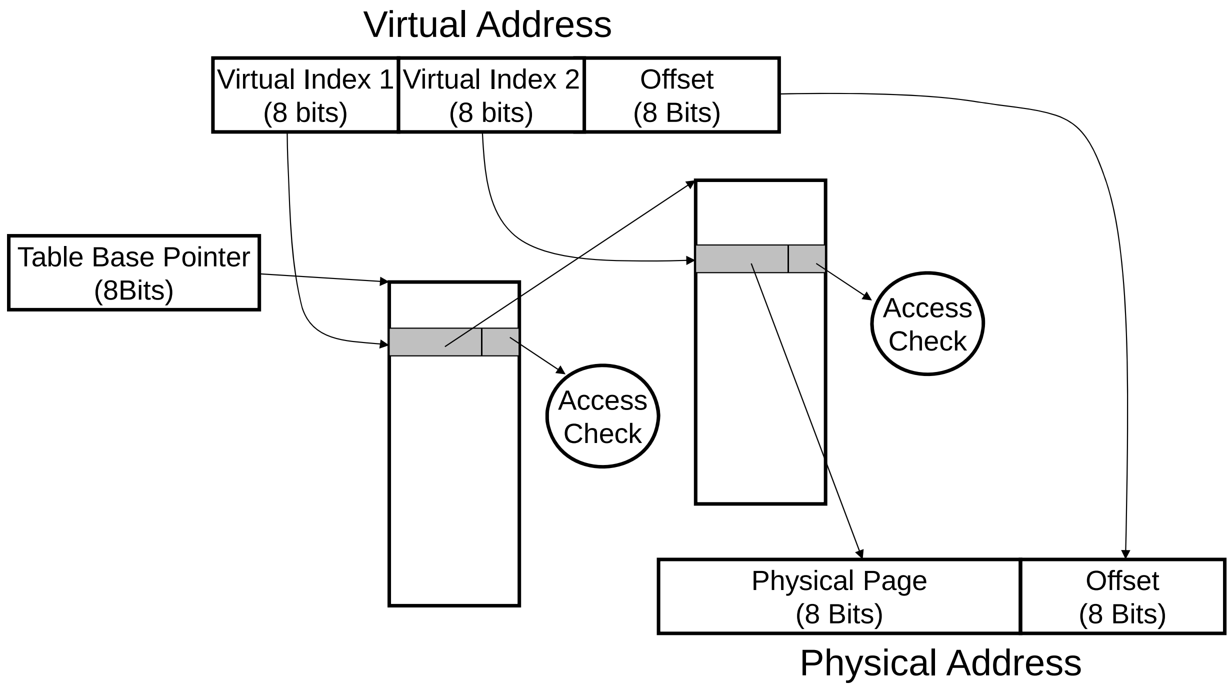
The largest page table would occur when there is a mapping for all pages in the address space. All entries would be filled at both levels of the page table. In this double-level page table, there will be 1 page-table chunk at the top level and 2^8 *page-table chunks at the second level for a total of* $1+2^8$ *chunks. Each chunk has* $2^8 \times SizeOf(PTE) = 512$ *bytes. Thus, the total size of the full page table is:* $(1+2^8) \times (512) = 131584$.

Problem 5c[3pts]: What does "TLB" stand for and what is its function? How big would a TLB entry be for this system?

"TLB" stands for Translation Lookahead Buffer. It serves as a cache for translations from the page table. Each TLB entry would contain a virtual page # (16 bits), a PTE (16 bits), and a valid bit (1 bit) for a total of 33 bits. Since there are two zero bits in the PTE, one could argue that the TLB entry only needs 31 bits.

Problem 5d[3pts]: Sketch the format of the page-table for this multi-level virtual memory management scheme. Illustrate the process of resolving an address as well as possible.

We were looking for something like this:



Problem 5e[2pts]: What is "Copy on Write"? How would you perform Copy on Write with the Virtual Memory system discussed in this problem?

> *Copy on Write is a mechanism for making cheap copies of an address space by creating a duplicate page table that points at the same physical pages as an existing page table. All PTEs are marked as read-only, allowing any write to be caught; at the point of a write, the target page can be copied so that each page table points at a unique copy. As just mentioned, Copy On Write is accomplished in this Virtual Memory system by setting the PTE to Read-Only so that the target page can be copied at the first write.*

Problem 5f[6pts]: The contents of physical memory are given on the next page. *Assume that the page-table base pointer = 0x2000, and that the CPU is in user-mode*. Please return the results from the following load/store instructions. Addresses are virtual. The return value for load is an 8-bit data value or an error, while the return value for a store is either "**ok**" or an error. For errors, please specify which type of error (either "invalid page" or "access violation").

Instruction	Return Value
Load [0x700FE]	**0xEE**
Store [0x700FE]	**Access violation**
Load [0xC2345]	*Invalid Page*
Load [0x00115]	*0x57*

Instruction	Return Value
Store [0x10310]	*OK*
Load [0x20102]	*0x01*
Store [0x20731]	*Access Violation*
Load [0x81015]	*Access Violation*

Virtual Address Format

Virtual Page # (8 bits)	Virtual Page # (8 bits)	Offset (8 bits)

Page Table Entry (PTE)

Physical Page # (8 bits)	Kernel	Not Cacheable	0	0	Dirty	Use	Write	Valid

Physical Memory

Address	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
0x0000	E0	F0	01	11	21	31	41	51	61	71	81	91	A1	B1	C1	D1
0x0010	1E	1F	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D
….																
0x1010	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
0x1020	40	07	41	06	30	06	31	07	00	07	00	00	00	00	00	00
….																
0x2000	21	01	22	03	25	01	22	01	2F	03	28	03	30	03	22	03
0x2010	40	81	41	81	42	81	43	83	00	00	00	00	00	00	00	00
….																
0x2100	30	05	31	01	32	03	33	07	34	00	35	00	36	00	37	00
0x2110	38	00	39	00	3A	00	3B	00	3C	00	3D	00	3E	00	3F	00
….																
0x2200	30	01	31	83	00	01	00	0F	04	00	05	00	06	00	07	00
0x2210	08	00	09	00	0A	00	0B	00	0C	00	0D	00	0E	00	0F	00
….																
0x2500	10	01	00	03	12	85	13	05	14	05	15	05	16	05	17	05
0x2510	18	85	19	85	1A	85	1B	85	1C	85	1D	85	1E	85	00	00
….																
0x2800	50	01	51	03	00	00	00	00	00	00	00	00	00	00	00	00
….																
0x2F00	60	03	28	03	62	00	63	00	64	03	65	00	66	00	67	00
0x2F10	68	00	69	00	00	00	00	00	00	00	00	00	00	00	00	00
0x2F20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
….																
0x30F0	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF
0x3100	01	12	23	34	45	56	67	78	89	9A	AB	BC	CD	DE	EF	00
0x3110	02	13	24	35	46	57	68	79	8A	9B	AC	BD	CE	DF	F0	01
….																
0x4000	30	00	31	06	32	07	33	07	34	06	35	00	43	38	32	79
0x4010	50	28	84	19	71	69	39	93	75	10	58	20	97	49	44	59
0x4020	23	87	20	07	00	06	62	08	99	86	28	03	48	25	34	21

[This page intentionally left blank]

[Scratch Page: Do not put answers here!]

[Scratch Page: Do not put answers here!]