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

Spring 2015 John Kubiatowicz

Midterm I

SOLUTIONS
March 11th, 2015
CS162: Operating Systems and Systems Programming

Your Name:	
SID Number:	
Discussion Section:	

General Information:
This is a **closed book** exam. You are allowed 1 page of **hand-written** notes (both sides). 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	18	
2	18	
3	24	
4	20	
5	20	
Total	100	

[This page left for π]

3.141592653589793238462643383279502884197169399375105820974944

Problem 1: TRUE/FALSE [18 pts]

In the following, it is important that you *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 kernel on a multiprocessor can use the local disabling of interrupts (within one CPU) to produce critical sections between the OSs on different CPUs.

True / [✓ False]

Explain: *Disabling of interrupts on one CPU does nothing to prevent the execution of code running on another CPU.*

Problem 1b[2pts]: Simultaneous Multithreading is a hardware mechanism that can switch threads every cycle.

[✓ True] / False

Explain: *Simultaneous Multithreading switches between threads that are loaded into the processor and can switch every cycle – in fact, it can even combine instructions from multiple threads in the same cycle.*

Problem 1c[2pts]: In a multi-process HTTP server (like in HW#2), only the child process is responsible for closing the client socket (e.g. the file descriptor returned by `accept()`), since the parent doesn't know when the child is done using the socket.

True / [✓ False]

Explain: *The `fork()` system call duplicates the client socket in both the parent and child. Consequently, both of them must close their copy of the socket; in fact, the parent typically closes the client socket immediately after executing the `fork()` operation.*

Problem 1d[2pts]: A user-level library implements each system call by first executing a "transition to kernel mode" instruction. The library routine then calls an appropriate subroutine in the kernel.

True / [✓ False]

Explain: *Such a mechanism (transition to kernel mode) would give too much control to user-level code. Instead, entry into the kernel is carefully controlled through a system call mechanism that* simultaneously *transitions into kernel mode while jumping to a well-defined entry point in the kernel.*

Problem 1e[2pts]: A thread can be blocked on multiple condition variables simultaneously.

True / [✓ False]

Explain: *After the thread calls wait() on the first condition (i.e. while it is still blocked on the first condition variable), it couldn't have called the second one yet and cannot be blocked on the second one.*

Problem 1f[2pts]: Floating point numbers are not used in Pintos because floating point operations are too slow and have rounding issues.

True / [✓ False]

Explain: *Floating point numbers are not used in Pintos simply because floating point registers are not saved/restored during thread switches.*

Problem 1g[2pts]: In Pintos, implementing priority scheduling for semaphores will also take care of priority scheduling for locks and condition variables. This is because locks and condition variables are implemented using semaphores.

True / [✓ False]

Explain: *Since locks and condition variables use queues of semaphores to hold blocked threads, priority scheduling within semaphores does not provide priority scheduling to locks or condition variables.*

Problem 1h[2pts]: The only way to resolve a resource deadlock is to reboot the system.

True / [✓ False]

Explain: *Resource deadlocks can be resolved in other ways such as transactional abort (/unrolling transactions).*

Problem 1i[2pts]: Calls to `printf()` always enter the kernel to perform an output to `stdout`.

True / [✓ False]

Explain: *`printf()` is a stream operation that buffers its output inside the user-level . Consequently, many calls to printf()exit without ever entering the kernel.*

Problem 2: Short Answer [18pts]

Problem 2a[3pts]: Name at least two disadvantages of disabling interrupts to serialize access to a critical section. When does it make sense to use interrupt disable/enable around a critical section?

Some possible answers include:

1) *Does not work at user-level (i.e. user-level code cannot disable interrupts).*
2) *Locks out other hardware interrupts and may cause critical events to be missed.*
3) *Is a very coarse-grained method of serializing – there ends up being only one such lock for the whole machine.*

It makes sense to disable interrupts at the core of the kernel when either the critical section is very short or when the arrival of interrupts would actually cause incorrect behavior (such as during context switching or during the entry/exit of interrupt handlers).

Problem 2b[2pts]: What is the difference between Mesa and Hoare scheduling for monitors? How does this affect the programming pattern used by programmers (be explicit)?

With Hoare scheduling, a `signal()` operation from one thread immediately wakes up a sleeping thread, hands the lock to the sleeping thread, and starts the sleeping thread executing; control returns to the signaling thread after the signaled thread attempts to release the lock (which is then handed back to the signaling thread). With Mesa scheduling, the signaling thread simply placed the signaled thread on the run queue and continues executing with the lock.

The practical consequence is that Mesa-scheduled monitors require programmers to recheck conditions after waking (typically with a "while" loop):

```
while (condition not satisfied)
        condition.wait();
```

With Hoare-scheduled monitors, the "while" statement can often be replaced with an "if" statement.

Problem 2c[2pts]: What needs to be saved and restored on a context switch between two threads in the same process? What if we have two different processes?

The registers (integer and floating-point), program counter, condition registers, and any other execution state for a thread must be saved and restored between context switches. If we are switching between two different processes, we must additionally save and restore the page table root pointer and/or segment registers.

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

The processor transitions from user mode to kernel mode when (1) the user executes a system call, (2) the processor encounters an synchronous exception such as divide by zero or page fault, (3) the processor responds to an interrupt. The user cannot execute arbitrary code because entry into the kernel is through controlled entry points (not under control of the user).

Problem 2e[2pts]: What is the difference between fork() and exec() on Unix?

The fork() system call creates a new child process whose address space duplicates that of the parent. In contrast, the exec() system call throws away the contents of a process' address space and replaces it by loading an executable (new program) from the filesystem.

Problem 2f[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 a number of reasons that overuse of threads is bad. Some of them include:

1) *The overhead of switching between too many threads can waste processor cycles such that overhead outweighs actual computation (i.e. thrashing).*
2) *Excessive threading can waste memory for stacks and TCBs*
3) *The overhead of splitting tasks into threads (the launching/exit process) may not be offset by the resulting gain in performance from parallelism.*

Problem 2g[2pts]: What is the default scheduler in PintOS?

The default scheduler in PintOS round-robin (with quantum = 4).

Problem 2h[2pts]: In PintOS, the code for `thread_unblock()` contains a comment that says "This function does not preempt the running thread". Explain why you should not modify `thread_unblock()` in a way that could cause it to preempt the running thread.

Many functions call thread_unblock with the assumption that it can do so and update other data structures atomically. Specifically, sema_up and potentially whatever wakes up sleeping threads.

Problem 3: Atomic Synchronization Primitives [24pts]

In class, we discussed a number of *atomic* hardware primitives that are available on modern architectures. In particular, we discussed "test and set" (TSET), SWAP, and "compare and swap" (CAS). They can be defined as follows (let "expr" be an expression, "&addr" be an address of a memory location, and "M[addr]" be the actual memory location at address addr):

Test and Set (TSET)	Atomic Swap (SWAP)	Compare and Swap (CAS)

`TSET(&addr) {`
`  int result = M[addr];`
`  M[addr] = 1;`
`  return (result);`
`}`
	`SWAP(&addr, expr) {`
	`  int result = M[addr];`
	`  M[addr] = expr;`
	`  return (result);`
	`}`
		`CAS(&addr, expr1, expr2) {`
		`  if (M[addr] == expr1) {`
		`     M[addr] = expr2;`
		`     return true;`
		`  } else {`
		`     return false;`
		`  }`
		`}`

Both TSET and SWAP return values (from memory), whereas CAS returns either true or false. Note that our &addr notation is similar to a reference in c++, and means that the &addr argument must be something that can be stored into (an "lvalue"). For instance, TSET could be used to implement a spin-lock acquire as follows:

```
int lock = 0; // lock is free

// Later: acquire lock
while (TSET(lock));
```

CAS is general enough as an atomic operation that it can be used to implement both TSET and SWAP. For instance, consider the following implementation of TSET with CAS:

```
TSET(&addr) {
   int temp;
   do {
      temp = M[addr];
   } while (!CAS(addr,temp,1));
   return temp;
}
```

Problem 3a[3pts]:
Show how to implement a spinlock acquire with a single while loop using CAS instead of TSET. You must only fill in the arguments to CAS below:

```
// Initialization
int lock = 0;  // Lock is free


// acquire lock

while (  !CAS(    lock     ,     0     ,     1     ) );
```

Problem 3b[2pts]:
Show how SWAP can be implemented using CAS. Don't forget the return value.

```
SWAP(&addr, reg1) {



   Object returnvalue;
   do {
      return = M[addr];
   } while (!CAS(addr, returnvalue, reg1));
   Return returnvalue;


}
```

Problem 3c[2pts]:
With spinlocks, threads spin in a loop (busy waiting) until the lock is freed. In class we argued that spinlocks were a bad idea because they can waste a lot of processor cycles. The alternative is to put a waiting process to sleep while it is waiting for the lock (using a blocking lock). Contrary to what we implied in class, there are cases in which spinlocks would be more efficient than blocking locks. Give a circumstance in which this is true and explain why a spinlock is more efficient.

If the expected wait time of the lock is very short (such as because the lock is rarely contested or the critical sections are very short), then it is possible that a spin lock will waste many fewer cycles than putting threads to sleep/waking them up. The important issue is that the expected wait time must be less than the time to put a thread to sleep and wake it up.

Short expected wait times are possible to capitalize on, for instance, in a multiprocessor because waiting threads can be stalled on other processors while the lock-holder makes progress. Spin-locks are much less useful in a uniprocessor because the lock-holder is sleeping while the waiter is spinning.

Some people mentioned I/O. However, you would have had to come up with a specific example of locks in use between a thread and an I/O operation as well as mentioned interrupts for releasing the lock.

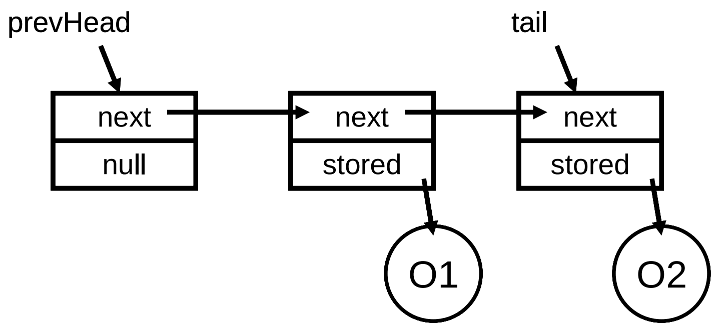
We were looking for mention of (1) expected wait time being important, (2) the length of time for putting locks to sleep relative to the expected wait time of the lock, (3) a viable scenario such as a multiprocessor.

An object such as a queue is considered “lock-free” if multiple processes can operate on this object simultaneously without requiring the use of locks, busy-waiting, or sleeping. In this problem, we are going to construct a lock-free FIFO queue using the atomic CAS operation. This queue needs both an `Enqueue` and `Dequeue` method.

We are going to do this in a slightly different way than normally. Rather than `Head` and `Tail` pointers, we are going to have “`PrevHead`” and `Tail` pointers. `PrevHead` will point at the last object returned from the queue. Thus, we can find the head of the queue (for dequeuing). If we don’t have to worry about simultaneous Enqueue or Dequeue operations, the code is straightforward:

```
// Holding cell for an entry
class QueueEntry {
     QueueEntry next = null;
     Object stored;

     QueueEntry(Object newobject) {
          stored = newobject;
     }
}

// The actual Queue (not yet lock free!)
class Queue {
     QueueEntry prevHead = new QueueEntry(null);
     QueueEntry tail = prevHead;

     void Enqueue(Object newobject)  {
          QueueEntry newEntry = new QueueEntry(newobject);
          QueneEntry oldtail = tail;
          tail = newEntry;
          oldtail.next = newEntry;
     }

     Object Dequeue() {
          QueueEntry oldprevHead = prevHead;
          QueueEntry nextEntry = oldprevHead.next;
          if (nextEntry == null)
               return null;
          prevHead = nextEntry;
          return nextEntry.stored;
     }
}
```

Problem 3d[3pts]:
For this non-multithreaded code, draw the state of a queue with 2 queued items on it:



Problem 3e[3pts]: For each of the following potential context switch points, state whether or not a context switch at that point could cause incorrect behavior of Enqueue(); Explain!

```
        void Enqueue(Object newobject)  {
1 ---------> QueueEntry newEntry = new QueueEntry(newobject);
2 ---------> QueueEntry oldtail = tail;
3 ---------> tail = newEntry;
            oldtail.next = newEntry;
        }
```

Point 1: *No. Construction of a QueueEntry is a purely local operation (and does not touch shared state in any way).*

Point 2: *Yes. An intervening Enqueue() operation will move the shared variable "tail" (and enqueue another object). As a result, the subsequent "tail=newEntry" will overwrite the other entry.*

Point 3: *No. At this point in the execution, only the local thread will ever touch "oldtail.next" (since we have moved the tail). Thus, we can reconnect at will. People who worried that the linked list is "broken" until this operation can relax. The worse that will happen is that the list appears to be shorter than it actually is until execution of "oldtail.next=newEntry," at which point the new entry becomes available for subsequent dequeue.*

Problem 3f[4pts]: Rewrite code for Enqueue(), using the CAS() operation, such that it will work for any number of simultaneous Enqueue and Dequeue operations. You should never need to busy wait. **Do not use locking (i.e. don't use a test-and-set lock).** The solution is tricky but can be done in a few lines. We will be grading on conciseness. Do not use more than one CAS() or more than 10 lines total (including the function declaration at the beginning). *Hint: wrap a do-while around vulnerable parts of the code identified above.*

```
void Enqueue(Object newobject)  {
    QueueEntry newEntry = new QueueEntry(newobject);

    // Insert code here


    // Here, 'tail' is the shared variable that needs to be
    // protected by CAS. We must atomically swap in 'newEntry'
    // to 'tail', giving us the old value so that we can link
    // it to the new item.

    QueueEntry oldtail;
    do {
        oldtail = tail;      // Tentative pointer to tail
    } while {!CAS(tail,oldtail,newEntry);
    oldtail.net = newEntry;





}
```

Problem 3g[3pts]: For each of the following potential context switch points, state whether or not a context switch at that point could cause incorrect behavior of Dequeue(); Explain! (Note: *Assume that the queue is not empty when answering this question, since we have removed the null-queue check from the original code):*

```
        Object Dequeue() {
1  --------> QueueEntry oldprevHead = prevHead;
2  --------> QueueEntry nextEntry = oldprevHead.next;
3  --------> prevHead = nextEntry;
            return nextEntry.stored;
        }
```

Point 1: *Yes. The problem is that an intervening Dequeue() could end up getting the same entry 'nextEntry' that we are returning; consequently we end up dequeing the same entry multiple times.*

Point 2: *Yes. The problem is that an intervening Dequeue() could end up getting the same entry 'nextEntry' that we are returning; consequently we end up dequeing the same entry multiple times.*

Point 3: *No. The nextEntry has already been detached from the queue and is purely local. Thus, all that we are doing is removing the stored value from nextEntry for returning it.*

Problem 3h[4pts]: Rewrite code for Dequeue(), using the CAS() operation, such that it will work for any number of simultaneous Enqueue and Dequeue operations. You should never need to busy wait. **Do not use locking (i.e. don't use a test-and-set lock).** The solution can be done in a few lines. We will be grading on conciseness. Do not use more than one CAS() or more than 10 lines total (including the function declaration at the beginning). You should correctly handle an empty queue by returning "null". *Hint: wrap a do-while around vulnerable parts of the code identified above and add back the null-check from the original code.*

```
Object Dequeue() {

    // Insert code here

    // Here, 'prevHead' is the shared variable that needs to be
    // protected by CAS. We must atomically grab the value of
    // prevHead.next and swap it into prevHead. The CAS lets
    // us do this operation by making sure that prevHead is
    // still equal to oldprevHead at the time that we swap
    // in prevHead.next. Note that we have included a check to
    // handle empty queues (not required for your solution)

    QueueEntry oldprevHead, nextEntry;
    do {
        oldprevHead = prevHead;
        nextEntry = oldprevHead.next;
        if (nextEntry == null) // handle empty queue
            return null;
    } while (!CAS(prevHead, oldprevHead, nextEntry));
    return oldprevHead.stored;
}
```

Problem 4: Scheduling and Deadlock [20 pts]

Problem 4a[2pts]: How could a priority scheduler be used to emulate Earliest Deadline First (EDF) scheduling? Would computing of priorities be an expensive operation (assume that we schedule periodic tasks characterized by period T and computational time of C)? Explain.

Current tasks must be sorted by deadline (i.e. "earliest deadline first"), then priorities assigned to these tasks in order of deadline, with a unique priority for each task and highest priority to the earliest deadline. This process of sorting and priority assignment must happen every time a new instance of a realtime task arrives (i.e. every T time units for every task's value of T). Priorities do not have to be changed when a task finishes.

Problem 4b[2pts]: What is a multi-level feedback scheduler and how can it approximate SRTF?

A multi-level feedback scheduler is one that includes multiple queues sorted in a priority order. Each queue (except for the highest-priority queue) is fed from the immediate queue of the next highest priority. New tasks are placed into the highest-priority queue. In addition to scheduling queues by priority, each queue can have its own scheduling policy or variation of a scheduling policy (such as round robin with different quantum). When an thread computes for too long (i.e. its quantum runs out), it is placed on the next lower-priority queue. When a thread finishes, it either moves up to the next higher-priority queue or into the top-level queue. This approximates SRTF because long-running tasks tend to get lower priority (and short-running tasks get higher priority).

Problem 4c[3pts]: What is priority donation? What sort of information must the OS track to allow it to perform priority donation? Is priority donation targeted at preventing a deadlock or a livelock?

Priority donation is a process by which a thread blocked on a synchronization construct (e.g. a lock) "donates" its priority to the holder of that synchronization construct (in order to make sure that there is no priority inversion, i.e that the blocking thread does not end up waiting for a thread of lower priority – which might itself be blocked). The OS must keep track of which threads are currently holding locks and which others are waiting on locks (of course it must already do the latter). Priority donation is targeted at preventing a livelock, since there is not necessarily a cyclic dependency.

Problem 4d[3pts]: Suppose that you utilize a scheme that schedules threads within a process at user level. Why might a naïve scheduling scheme run into problems when accessing I/O? Can the operating system help resolve this problem? Explain

Because a naïve scheme would utilize one "kernel" thread for all of the user-level threads. Then, if any of the threads were to do a system call that blocked on I/O, all other threads would be blocked as well. One way for the operating system to help would be to return a new kernel thread to the process for every thread that is put to sleep in the kernel. This idea is typically called "scheduler activations." As a result, even if a thread blocks in the kernel, the process will always retain a running thread which it can be utilized to scheduler user-level thread.

Pwnage Games, a fairly unknown arcade in Downtown Berkeley, decided to purchase Super Smash Bros. for Wii U -- a popular fighting video game -- in the hope that it would draw customers to the business. However, due to limited resources, the store could only buy one copy of the game. Luckily, the owners know Gill Bates -- a Cal EECS undergrad -- who offers her help in exchange for free arcade credits. Her job is to allow multiple consoles to play the game at the same time. Thanks to her hacking skills, Gill completes the task in no time, but she is forced to impose some conditions on the gameplay:

- each console only allows for two players to fight at a time;
- the same character cannot be used by more than one player at a time.

The enforcement of these conditions is handled after character selection. That is, all fighters appear available at all times, and the following function loads the fight. Each character has a global fighter_t* representing it across consoles.

```
void smash (fighter_t* first, fighter_t* second)
{
   pthread_mutex_lock (&first->lock);
   pthread_mutex_lock (&second->lock);
   fight (first, second);
   pthread_mutex_unlock (&second->lock);
   pthread_mutex_unlock (&first->lock);

}
```

Problem 4e[4pts]: Despite Gill's effort, her algorithm has an obvious flaw: it can lead to deadlock! Present an example of how this can happen. List the four conditions for deadlock and show how they are satisfied by this example:

Simple case: two consoles request the same two players, but in reverse order. In this case, the "first" and "second" arguments to `smash()` at each console are reversed. Consequently, first console (Console #1) locks Player A and is waiting for Player B while the second console (Console #2) locks Player B and is waiting for Player A. This is a cycle that will never resolve.

1) *Mutual Exclusion: Each player can be owned by only one console at a time.*
2) *Hold and Wait: The `smash()` function for console #1 in this example is holding one resource (player A) while it is waiting for another resource (player B). A similar argument applies to the `smash()` function for console #2 (with resources in reverse order).*
3) *No Premption: Resources are only released by the owner (never preempted).*
4) *Circular wait: We have a circular wait since Console #1 (T_1) is waiting for Console #2 (T_2) which is waiting for Console #1 (T_1).*

Problem 4f[3pts]: Redesign the `smash()` function to avoid deadlock. Write your new version in the space below. Which of the four conditions are now missing? Name one downside of your approach.

We can fix this problem by introducing a global lock that is acquired before any players are acquired:

```
void smash (fighter_t* first, fighter_t* second)
{
   pthread_mutex_lock (&global_lock);
   pthread_mutex_lock (&first->lock);
   pthread_mutex_lock (&second->lock);
   pthread_mutex_unlock (&global_lock);
   fight (first, second);
   pthread_mutex_unlock (&second->lock);
   pthread_mutex_unlock (&first->lock);

}
```

This prevents deadlock by removing the circular wait condition: threads only wait for either (1) the global lock, which means that they are waiting on some thread trying to acquire its players or (2) the one thread that has acquired the global lock may be waiting to acquire one of its players which can only be owned by a console that is fighting (and which will eventually release this player). Consequently, the wait graph is a tree.

Note that a downside of this "fix" is that it could delay unrelated fights from occurring. For instance, if one console is playing with Player A and B, a second one is waiting for Player A, then subsequent consoles are unable to play – even if they are interested in Players C and D (for instance). One could release the global lock after the last unlock (at the end of the smash() function), but this would introduce additional waiting.

Note: solutions involving busywaiting (such as use of `trylock()`*) were not given full credit.*

Problem 4g[3pts]: Explain how the Banker's algorithm could prevent the deadlock identified in Problem (4e) and what changes would need to be made to the code to support it. Clearly identify the behavior that would result, and why the four conditions for deadlock are not simultaneously satisfied. Would this solution be better or worse than your solution to Problem (4f)?

The bankers algorithm would prevent deadlock because it would never grant either Player A to Console #1 or Player B to Console #2 if it would result in deadlock. The changes to the code from (4e) would involve (1) declaring which players a thread was interested in before trying to acquire them (i.e. at the beginning of `smash()` *), changing the mutex_lock code to keep track of all acquired resources and to perform the Banker's algorithm before every grant operation and put a requesting thread to sleep if any proposed acquisition would introduce deadlock. As already stated, this type of tracking would prevent cycles from forming by preventing the acquisition of any resource that would form a cycle. This solution would be better than the solution in (4f), since it would never interfere with acquisition of unrelated players.*

[This page intentionally left blank]

Problem 5: Address Translation [20 pts]

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 in the following format, *stored in big-endian form* in memory (i.e. the MSB is first byte in memory):

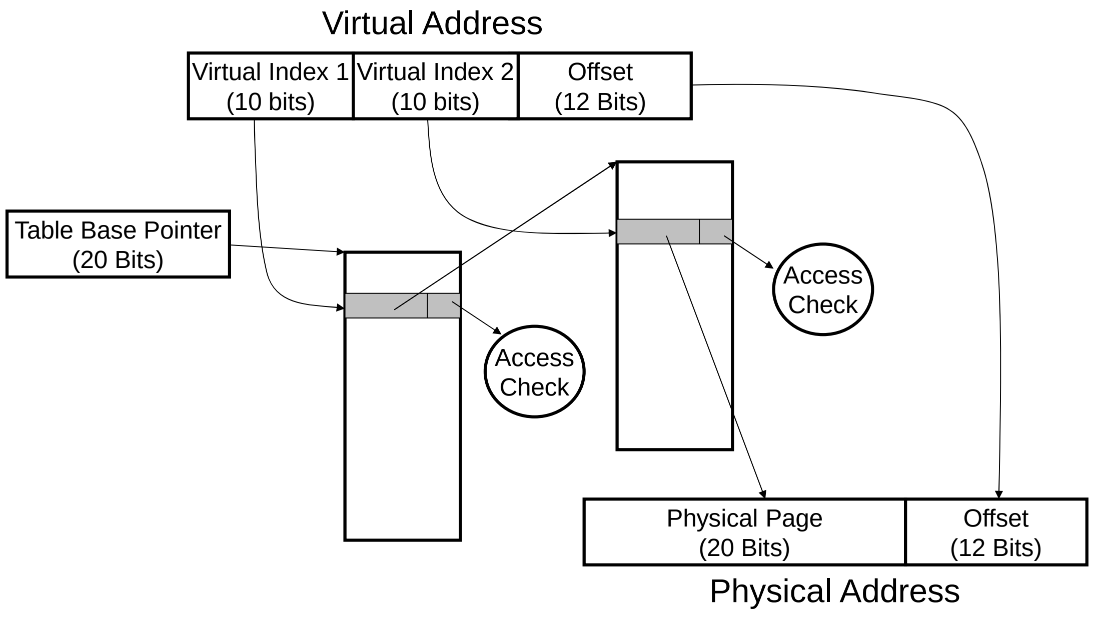
Physical Page # (20 bits)	OS Defined (3 bits)	0	Large Page	Dirty	Accessed	Nocache	Write Through	User	Writeable	Valid

Here, "Valid" means that a translation is valid, "Writeable" means that the page is writeable, "User" means that the page is accessible by the User (rather than only by the Kernel). *Note: the phrase "page table" in the following questions means the multi-level data structure that maps virtual addresses to physical addresses.*

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

Pages are 4K in size, since the offset is 12 bits ($2^{12} = 4096$)

Problem 5b[4pts]: Draw a picture of the page table. What good property(s) result from dividing the address into three fields in this way (i.e. 32 bits = 10bits + 10bits + 12bits)?



One really nice property resulting from this division (and the size of the PTE) is that every piece of the page-table is the same size as the pages; consequently, the OS could "page out the page table"

Problem 5c[2pts]: Suppose that we want an address space with one physical page at the top 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.

To address a physical page at the top and bottom of the address space, we would need at least 3 page table components (the top-level one, and two second-level page table entries). Thus, the page table would be 3 × 4096 = 12288 bytes.

Problem 5d[2pts]: What is the maximum amount of physical memory that can be addressed by this page table. Explain.

This page table scheme can address all of physical memory, which is 2^{32} = 4294967296 bytes. For those of you who thought we were talking about the particular page table organization in (5c), we would also accept an answer of 2 pages = 8192 bytes.

Problem 5e[10pts]: Assume the memory translation scheme from (5a). Use the Physical Memory table given on the next page to predict what will happen with the following load/store instructions. Assume that the base table pointer for the current ***user level process*** is `0x00200000`.

Addresses in the "Instruction" column are virtual. You should translate these addresses to physical address (i.e. in middle column), then attempt to execute the specified instruction on the resulting address. The return value for a load is an 8-bit data value or an error, while the return value for a store is either "**ok**" or an error. Possible errors are: **invalid, read-only, kernel-only.**
Hints: (1) Don't forget that Hexidecimal digits contain 4 bits! (2) PTEs are 4 bytes!

Instruction	Physical Address	Result
Load [0x00001047]	0x00002047	0x50
Store [0x00C07665]	0xEEFF0665	ok
Store [0x00C005FF]	0x112205FF	ERROR: read-only
Load [0x00003012]	*0x00004012*	*0x36*
Store [0x02001345]	*0x00002345*	*ok*
Load [0xFF80078F]	*0x0415078F*	*ERROR: invalid*
Load [0xFFFFF005]	*0X00103005*	*0X66*
Test-And-Set [0xFFFFF006]	*0X00103006*	*0X77*

Physical Memory [All Values are in Hexidecimal]

Address	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
00000000	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
00000010	1E	1F	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D
….																
00001010	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
00001020	40	03	41	01	30	01	31	03	00	03	00	00	00	00	00	00
00001030	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF
00001040	10	01	11	03	31	03	13	00	14	01	15	03	16	01	17	00
….																
00002030	10	01	11	00	12	03	67	03	11	03	00	00	00	00	00	00
00002040	02	20	03	30	04	40	05	50	01	60	03	70	08	80	09	90
00002050	10	00	31	01	10	03	31	01	12	03	30	00	10	00	10	01
….																
00004000	30	00	31	01	11	01	33	03	34	01	35	00	43	38	32	79
00004010	50	28	36	19	71	69	39	93	75	10	58	20	97	49	44	59
00004020	23	03	20	03	00	01	62	08	99	86	28	03	48	25	34	21
….																
00100000	00	00	10	67	00	00	20	67	00	00	30	00	00	00	40	07
00100010	00	00	50	03	00	00	00	00	00	00	00	00	00	00	00	00
…																
00103000	11	22	00	05	55	66	77	88	99	AA	BB	CC	DD	EE	FF	00
00103010	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF	00	67
…																
001FE000	04	15	00	00	48	59	70	7B	8C	9D	AE	BF	D0	E1	F2	03
001FE010	10	15	00	67	10	15	10	67	10	15	20	67	10	15	30	67
…																
001FF000	00	00	00	00	00	00	00	65	00	00	10	67	00	00	00	00
001FF010	00	00	20	67	00	00	30	67	00	00	40	65	00	00	50	07
…																
001FFFF0	00	00	00	00	00	00	00	00	10	00	00	67	00	10	30	67
…																
00200000	00	10	00	07	00	10	10	07	00	10	20	07	00	10	30	07
00200010	00	10	40	07	00	10	50	07	00	10	60	07	00	10	70	07
00200020	00	10	00	07	00	00	00	00	00	00	00	00	00	00	00	00
…																
00200FF0	00	00	00	00	00	00	00	00	00	1F	E0	07	00	1F	F0	07
…																

[This page intentionally left blank]

[This page left for scratch]