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

Fall 2015 John Kubiatowicz

Midterm I

SOLUTIONS

October 14th, 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	20	
4	24	
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]: When you type Ctrl+C to quit a program in your terminal, you are actually sending a SIGINT signal to the program, which makes it quit.

True / False

Explain: *SIGINT is a vehicle for conveying a Ctrl-C requests from the terminal to the program. Assuming that the SIGINT handler has not be redirected, this will cause the program to quit. Note: we would also take "False" if you explain that the SIGINT handler might have been redirected.*

Problem 1b[2pts]: The function `pthread_intr_disable()` is a crude but viable way for user programs to implement an atomic section.

True / False

Explain: *This function does not exist. However, you cannot disable interrupt at user level in any case. A similarly-named routine (*`pthread_setintr()`*) simply disables the delivery of signals; it does not prevent multiple threads from executing concurrently—thus it could not be used to implement a critical section.*

Problem 1c[2pts]: If the banker's algorithm finds that it's safe to allocate a resource to an existing thread, then all threads will eventually complete.

True / False

Explain: *When the banker's algorithm finds that it's safe to allocate a resource, this simply means that threads could complete from a resource allocation standpoint. However, threads could still go into an infinite loop or otherwise fail to complete.*

Problem 1d[2pts]: The lottery scheduler can be utilized to implement strict priority scheduling.

True / False

Explain: *Strict priority scheduling would require the ability to have high-priority threads that receive all CPU time – at the expense of lower priority threads. The lottery scheduler will give some CPU time to every thread (except those which have zero tokens).*

Problem 1e[2pts]: Locks can be implemented using semaphores.

True / False

Explain: *Initializing a semaphore with the value "1" will cause it to behave like a lock. Semi.P() ⇒ acquire() and Semi.V() ⇒ release().*

Problem 1f[2pts]: Two processes can share information by reading and writing from a shared linked list.

True / False

Explain: *If a shared page is mapped into the same place in the address space of two processes, then they can share data structures that utilize pointers as long as they are stored in the shared page and pointers are to structures in the shared page.*

Problem 1g[2pts]: In Pintos, a kernel-level stack can grow as large as it needs to be to perform its functions.

True / False

Explain: *Stacks in the Pintos kernel must fit entirely into a 4K page. In fact, they share the 4K page with the corresponding Thread Control Block (TCB).*

Problem 1h[2pts]: Suppose that a shell program wants to execute another program and wait on its result. It does this by creating a thread, calling exec from within that thread, then waiting in the original thread.

True / False

Explain: *The shell program must create a new process (not thread!) before calling* exec() *otherwise, the* exec() *call will terminate the existing process and start a new process – effectively terminating the shell.*

Problem 1i[2pts]: A network server in Linux works by calling `bind()` on a socket, and then calling `listen()` on the socket in a loop to wait for new clients.

True / False

Explain: *The server calls* listen() *only once, then* accept() *multiple times in a loop.*

Problem 2: Short Answer [18pts]

Problem 2a[2pts]: How does a modern OS regain control of the CPU from a program stuck in an infinite loop?

> *Assuming that we are talking about a user-level program (or a kernel thread with interrupts enabled), the timer interrupt handler (triggered by the timer) will enter the scheduler and recover the CPU from a program that is stuck in an infinite loop.*

Problem 2b[2pts]: Is it possible for an interrupt handler (code triggered by a hardware interrupt) to sleep while waiting for another event? If so, explain how. If not, explain why not.

> *There are actually two possible answers to this question. (1) "NO": Strictly speaking, an interrupt handler must not sleep while waiting for another event, since it doesn't have a thread-control block (context) to put onto a wait queue and is operating with interrupts disabled. However, one could also answer (2) "YES": an interrupt handler that wants to sleep must allocate a new kernel thread to finish its work, place the thread on a wait queue, then return from the interrupt (reenabling interrupts in the process).*

Problem 2c[2pts]: Why is it important for system calls to be vectored through the syscall table (indexed by an integer syscall number) rather than allowing the user to specify a function address to be called by the kernel after it transitions to kernel mode?

> *If the user were able to specify an arbitrary address for execution in the kernel, then they could bypass checking and find many ways to violate protection. Consequently, the user must specify a syscall* number *during the execution of a system call. The hardware then atomically raises the hardware level to "kernel level" while executing the system call from the specified entry point.*

Problem 2d[3pts]: Name two advantages and one disadvantage of implementing a threading package at user level (e.g. "green threads") rather than relying on thread scheduling from within the kernel.

> *Advantages include: very fast context switch (all at user level, no system call), very low overhead thread fork, and user-configurable scheduling. One very important disadvantage is the fact that all threads will get put to sleep when any one thread enters into the kernel and blocks on I/O.*

Problem 2e[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 a number of possible answers (1) Too many threads scheduled simultaneously can lead to excessive context switch overhead. (2) Too many threads can lead to memory overutilization. (3) Too many threads can cause excessive synchronization overhead (many locks to handle all the parallelism).

Problem 2f[2pts]: What was the problem with the Therac-25? Your answer should involve one of the topics of the class.

The Therac-25 had a number of synchronization problems, including improper synchronization between the operator console and the turntable mechanism which caused patients to receive the wrong type and dosage of radiation

Problem 2g[2pts]: Why is it possible for a web browser (such as Firefox) to have 2 different tabs opened to the same website (at the same remote IP address and port) without mixing up content directed at each tab?

Because a unique TCP/IP connection consists of a 5-tuple, namely [source IP, source Port, destination IP, destination Port, and protocol] (where the protocol is "6" for TCP/IP – which you didn't need to know). Consequently, although the web browser might have many connections with the same source IP address, destination IP address and destination Port, they will all have unique source ports, allowing them to be unique.

Problem 2h[3pts]: What are some of the hardware differences between kernel mode and user mode? Name at least three.

There are a number of differences: (1) There is at least one status bit (the "kernel mode bit") which changes between kernel mode and user mode. In an x86 processor, there are 2 bits which change (since there a 4 modes). (2) Additional kernel-mode instructions are available (such as those that modify the page table registers, those that enable and disable interrupts, etc). (3) Pages marked as kernel-mode in their PTEs are only available in kernel mode. (4) Control for I/O devices (such as the timer, interrupt controllers, and device controllers) are typically only available from kernel mode.

Problem 3: Boy-Girl Lock [20pts]

A boy-girl lock is a sort of generalized reader-writer lock: in a reader-writer lock there can be any number of readers or a single writer (but not both readers and writers at the same time), while in a boy-girl lock there can be any number of boys or any number of girls (but not both a boy and a girl at the same time). Assume that we are going to implement this lock at user level utilizing pThread monitors (i.e pThread mutexes and condition variables). Note that the assumption here is that we will put threads to sleep when they attempt to acquire the lock as a Boy when it is already acquired by one or more Girls and vice-versa. *You must implement the behavior using condition variable(s). Points will be deducted for any spin-waiting behavior.*

Some snippets from POSIX Thread manual pages showing function signatures are shown at end of this problem. They may or may not be useful.

Our first take at this lock is going to utilize the following structure and enumeration type:

```
/* The basic structure of a boy-girl lock */
struct bglock {
    pthread_mutex_t lock;
    pthread_cond_t wait_var;

    // Simple state variable
    int state;
};

/* Enumeration to indicate type of requested lock */
enum bglock_type {
    BGLOCK_BOY = 0;
    BGLOCK_GIRL = 1;
};

/* interface functions: return 0 on success, error code on failure */
int bglock_init(struct bglock *lock);
int bglock_lock(struct bglock *lock, enum bglock_type type);
int bglock_unlock(struct bglock *lock);
```

Note that the lock requestor specifies the type of lock that they want at the time that they make the request:

```
/* Request a Boy lock */
if (bglock_lock(mylock, BGLOCK_BOY) {
    printf(“Lock request failed!”);
    exit(1);
}
/* . . . Code using lock . . . */

/* Release your lock */
bglock_unlock(mylock);
```

Problem 3a[3pts]: Complete the following sketch for the initialization function. Note that initialization should return zero on success and a non-zero error code on failure (e.g. return the failure code, if you encounter one, from the various synchronization functions). *Hint: the state of the lock is more than just "acquired" or "free"*.

```
/* Initialize the BG lock.
 *
 * Args: pointer to a bglock
 * Returns: 0 (success)
 *          non-zero (errno code from synchronization functions)
 */
int bglock_init(struct bglock *lock) {
    int result;

    lock->state = 0; // No lock holders of any type

    if (result = pthread_mutex_init(&(lock->mutex), NULL))
        return result;  // Error

    result = pthread_cond_init(&(lock->wait_var), NULL);
    return result;
}
```

Problem 3b[5pts]: Complete the following sketch for the lock function. Think carefully about the state of the lock; when you should wait, when you can grab the lock.

```
/* Grab a BG lock.
 *
 * Args: (pointer to a bglock, enum lock type)
 * Returns: 0 (lock acquired)
 *          non-zero (errno code from synchronization functions)
 */
int bglock_lock(struct bglock *lock, enum bglock_type type) {
    int dir = (type == BGLOCK_BOY)?1:-1;  // Direction
    int result;

    // Grab monitor lock
    if (result = pthread_mutex_lock(lock->lock))
        return result; // error

    while (lock->state * dir < 0) {
        // Incompatible threads already have bglock, must sleep
        if (result = pthread_cond_wait(lock->wait_var, lock->lock))
            return result; // error
    }
    lock->state += dir;  // register new bglock holder of this type

    // Release monitor lock
    result = pthread_mutex_unlock(lock->lock);
    return result;
}
```

Problem 3c[5pts]: Complete the following sketch for the unlock function.

```
/* Release a BG lock.
 *
 * Args: pointer to a bglock
 * Returns: 0 (lock acquired)
 *          non-zero (errno code from synchronization functions)
 */
int bglock_unlock(struct bglock *lock) {
    int result;

    // Grab monitor lock
    if (result = pthread_mutex_lock(lock->lock))
        return result; // error

    // one less bglock holder of this type
    lock->state -= (type == BGLOCK_BOY)?1:-1;  // Direction

    // If returning to neutral status, signal any waiters
    if (lock->state == 0)
        if (result = pthread_cond_signal(lock->wait_var))
            return result; // error

    // Release monitor lock
    result = pthread_mutex_unlock(lock->lock);
    return result;
}
```

Problem 3d[2pts]: Consider a group of "nearly" simultaneous arrivals (i.e. they arrive in a period much quicker than the time for any one thread that has successfully acquired the BGlock to get around to performing `bglock_unlock()`). Assume that they enter the `bglock_lock()`routine in this order:

$B_1, B_2, G_1, G_2, B_3, G_3, B_4, B_5, B_6, B_7$

How will they be grouped? (Place braces, namely "{}" around requests that will hold the lock simultaneously). This simple lock implementation (with a single state variable) is subject to starvation. Explain.

All of the boy requests will go first, followed by girl requests:

$\{ B_1, B_2, B_3, B_4, B_5, B_6, B_7 \}, \{ G_1, G_2, G_3 \}$

This implementation experiences starvation because a series of waiting girl lock requests could be arbitrarily held off if there is a stream of boy requests (and vice-versa).

Problem 3e[5pts]: Suppose that we want to enforce fairness, such that Boy and Girl requests are divided into phases based on arrival time into the `bglock_lock()` routine. Thus, for instance, an arrival stream of Boys and Girls such as this:

$B_1, B_2, G_1, G_2, G_3, G_4, B_3, G_5, B_4, B_5$

will get granted in groups such as this:

$\{B_1, B_2\}, \{G_1, G_2, G_3, G_4\}, \{B_3\}, \{G_5\}, \{B_4, B_5\}$

Explain what the minimum changes are that you would need to make to the `bglock` structure to meet these requirements and sketch out what you would do during `bglock_init()` and `bglock_lock()` and `bglock_unlock()` routines. You do not need to write actual code, but should be explicit about what your `bglock` structure would look like and how you would use its fields to accomplish the desired behavior.

Here, we number phases starting from zero (with wraparound). We expand our state variable to queue of state variables. Each incoming thread figures out which phase they are in and then optionally sleeps (if they are not in the current phase). Our new bglock looks like this:

```
struct bglock {
    pthread_mutex_t lock;
    pthread_cond_t wait_var;

    int headphase, tailphase;
    int state[MAX_PHASES+1];
};
```

bglock_init(): initialize lock and wait_var; headphase=0; tailphase=0; state[x]=0 for all x;

bglock_lock(): if state[tailphase] doesn't match request, increment tailphase (with wrapping – may have to sleep if already have MAX_PHASES phases). In whatever case, increment state[tailphase] in correct direction (+1 or -1) depending on desired type of lock. Save current tailphase as your phase. Then, wait on condition variable until headphase == current tailphase.

bglock_unlock(): Decrement state[headphase] in correct direction (-1 or +1) depending on desired type of lock. If state[headphase]==0, check to see if headphase!=tailphase. If so, headphase++, broadcast to wake up everyone on condition variable.

Note that you can optimize wakeup behavior by adding a queue of condition variables as well, although this will increase the amount of state in the bglock.

Assorted POSIX Thread Manual Snippits for Problem 3

```
PTHREAD_MUTEX_DESTROY(3P): initialization/destruction of mutexes
       int pthread_mutex_destroy(pthread_mutex_t *mutex);

       int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                              const pthread_mutexattr_t *restrict attr);

       pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;


PTHREAD_MUTEX_LOCK(3P): use of mutex
       int pthread_mutex_lock(pthread_mutex_t *mutex);

       int pthread_mutex_trylock(pthread_mutex_t *mutex);

       int pthread_mutex_unlock(pthread_mutex_t *mutex);


PTHREAD_COND_DESTROY(3P): initialization/destruction of condition variables
       int pthread_cond_destroy(pthread_cond_t *cond);

       int pthread_cond_init(pthread_cond_t *restrict cond,
                             const pthread_condattr_t *restrict attr);

       pthread_cond_t cond = PTHREAD_COND_INITIALIZER;


PTHREAD_COND_TIMEDWAIT(3P): sleeping on condition variables
       int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                                  pthread_mutex_t *restrict mutex,
                                  const struct timespec *restrict abstime);

       int pthread_cond_wait(pthread_cond_t *restrict cond,
                             pthread_mutex_t *restrict mutex);


PTHREAD_COND_BROADCAST(3P): signaling of threads waiting on condition variables
       int pthread_cond_broadcast(pthread_cond_t *cond);

       int pthread_cond_signal(pthread_cond_t *cond);
```

Problem 4: Scheduling and Deadlock [24 pts]

Problem 4a[3pts]: What is priority inversion and why is it an important problem? Present a priority inversion scenario in which a lower priority process can prevent a higher-priority process from running (assume that there is no priority donation mechanism):

Priority inversion is a situation in which a lower-priority task is allowed to run over a higher-priority task. Consider three tasks in priority order: T1, T2, and T3 (i.e. T1 is lowest, T3 is highest). Suppose that T1 grabs a lock, T2 starts running, then T3 tries to grab the lock (and sleeps). Here, T2 is effectively preventing T3 from running (since T2 is preventing T1 from running, which is preventing T3 from running). The result is priority inversion.

Problem 4b[3pts]: How does the Linux CFS ("Completely Fair Scheduler") scheduler decide which thread to run next? What aspect of its behavior is "fair"? (You can ignore the presence of priorities or "nice" values in your answer):

The Linux CFS scheduler computes something called "virtual time" which is a scaled version of real CPU time. The scheduler attempts to make sure that every thread has an equal amount of virtual time. Thus, to decide which thread to run next, it simply picks the thread with the least amount of accumulated virtual time. This behavior is considered "fair" because it attempts to distribute the same total virtual time to every thread.

```
  void main (void) {
1     thread_set_priority(10);
2     struct lock a, b, c;
3     lock_init(&a);
4     lock_init(&b);
5     lock_init(&c);
6     lock_acquire(&a);
7     lock_acquire(&b);
8     lock_acquire(&c);
9     printf("1");
10    thread_create("a",15,func,&a);
11    printf("6");
12    thread_create("b",20,func,&b);
13    printf("2");
14    thread_create("c",25,func,&c);
15    lock_release(&c);
16    lock_release(&a);
17    lock_release(&b);
18    printf("!");
  }
```

```
  void func(void* lock_) {
19   struct lock *lock = lock_;

20   lock_acquire(&lock);
21   lock_release(&lock);

22   printf("%s",thread_current()->name);

23   thread_exit();
  }
```

Problem 4c[2pts]: Consider the above PintOS test that exercises your priority scheduler. Assume that no priority donation has been implemented. What does it output to the terminal? Is the output affected by priorities in any way? Explain.

This code will output "162cab!". This result is not affected by priorities (as long as all threads running "func()" are higher priority than "main()", since high-priority threads go to sleep almost immediately after they start and are released in order by lines #15, #16, and !37.

Problem 4d[5pts]: Next, assume that the code from (4c) is executed utilizing priority donation. Fill in the following table to detail execution. This table includes 7 columns as following:

1) The current executing thread
2) Which line this thread was executing when it yielded
3) To which thread it yielded

4-7) The priorities of each thread (N/A if a thread is not created or has exited)

`thread_current()`	Line at which yielded	Thread which it yielded to	Main	a	b	c
`main`	`10`	`a`	`10`	`15`	`N/A`	`N/A`
a	20	main	15	15	N/A	N/A
main	12	b	15	15	20	N/A
b	20	main	20	15	20	
main	14	c	20	15	20	25
c	20	main	25	15	20	25
main	15	c	20	15	20	25
c	23	main	20	15	20	N/A
main	17	b	10	15	20	N/A
b	23	a	10	15	N/A	N/A
a	23	main	10	N/A	N/A	N/A

Problem 4e[2pts]: What is printed according to the order of execution in (4d)? Is the output affected by priorities in any way? Explain.

What is printed is: "162cba!". Yes, the output is affected by priorities in that thread "b" gets to acquire lock "b" before thread "a" acquires lock "a". The ordering of letters happens in priority order.

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? If "yes", show a non-blocking sequence of thread executions. Otherwise, provide a proof that the system is unsafe. Show your work and justify each step of your answer.

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? 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[2pts]: Assume that a set of threads (T_1, T_2, … T_n) contend for a set of non-preemptable resources (R_1, R_2, … R_m) that may or may not be unique. Name at least two techniques to *prevent* this system from deadlocking:

We discussed several possible ways of preventing deadlock in class. Possibilities include:

1) *Pick a fixed order of allocation (say R_1 then R_2 then … R_m). All threads should allocate resources in this order.*
2) *Every thread should indicate which resources they want at the beginning of execution. Then, the thread is not allowed to start until after the requested resources are all available.*
3) *Use the Bankers algorithm on every allocation request to make sure that the system stays in a safe state.*

[This page intentionally left blank]

Problem 5: Address Translation [20 pts]

Problem 5a[3pts]: In class, we discussed the "magic" address format for a multi-level page table on a 32-bit machine, namely one that divided the address as follows:

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

You can assume that Page Table Entries (PTEs) are 32-bits in size in the following format:

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

What is particularly "magic" about this configuration? Make sure that your answer involves the size of the page table and explains why this configuration is helpful for an operating system attempting to deal with limited physical memory.

> *Each page is 4K in size (12-bit offset ⇒ 2^{12} bytes). Because the PTE is 4-bytes long and each level of the page table has 1024 entries (i.e. 10-bit virtual page #), this means that each level of the page table is 4K in size, i.e. exactly the same size as a page. Thus the configuration is "magic" because every level of the page table takes exactly one page. This is helpful for an operating system because it allows the OS to page out parts of the page table to disk.*

Problem 5b[2pts]: Modern processors nominally address 64-bits of address space both virtually and physically (in reality they provide access to less, but ignore that for now). Explain why the page table entries (PTEs) given in (5a) would have to be expanded from 4 bytes and justify how big they would it need to be. Assume that pages are the same size and that the new PTE has similar control bits to the version given in (5a).

> *Since we are attempting to address 64-bits of physical DRAM and the page offset is 12-bits, this leaves 52-bits of physical page # that will have to fit into the PTE. The old PTE had only 20-bits of space for physical page #. In fact, we need another 32-bits of offset⇒PTE needs 52-bits of offset+12-bits of control bits (to be the same), yielding 64-bits of PTE, or 8-bytes.*

Problem 5c[2pts]: Assuming that we reserve 8-bytes for each PTE in the page table (whether or not they need all 8 bytes), how would the virtual address be divided for a 64-bit address space? Make sure that your resulting scheme has a similar "magic" property as in (5a) and that all levels of the page table are the same size—with the exception of the top-level. How many levels of page table would this imply? Explain your answer!

> *To have the same "magic" property, we would like each level of the page table to be the same size as a page – so that the OS could page out individual parts of the page table. Since a PTE is 8-bytes (3-bits in size), this means we need 12-3 = 9-bits of virtual page # at each level. This means that the virtual address needs to be divided into groupings of 9-bits (although the top level be smaller, since we only have 64-bits):*
>
> *[7-bits][9-bits][9-bits][9-bits][9-bits][9-bits][12-bits offset]*
>
> *Thus, there are 6 levels of page table.*

Problem 5d[3pts]: Consider a multi-level memory management scheme using the following format for *virtual addresses*, including 2 bits worth of segment ID and an 8-bit virtual page number:

Virtual seg # (2 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 (PTE) are 16 bits in the following format, *stored in big-endian form* in memory (i.e. the MSB is first byte in memory):

Physical Page # (8 bits)	Kernel	Nocache	0	0	Dirty	Use	Writeable	Valid

2) How big is a page? Explain.

A page is 2^8=256 bytes.

2) What is the maximum amount of physical memory supported by this scheme? Explain

Physical addresses have 16-bits $\Rightarrow 2^{16}$=65536 bytes (i.e. 64K bytes)

Problem 5e[10pts]: Using the scheme from (5d) and the Segment Table and Physical Memory table on the next page, state what will happen with the following loads and stores. Addresses below are virtual, while base addresses in the segment table are physical. If you can translate the address, make sure to place it in the "Physical Address" column; otherwise state "**N/A**".

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. If there is an error, say which error. Possibilities are: "**bad segment**" (invalid segment), "**segment overflow**" (address outside segment), or "**access violation**" (page invalid/attempt to write a read only page). A few answers are given:

Instruction	Translated Physical Address	Result (return value)
Load [0x30115]	**0x3115**	**0x57**
Store [0x10345]	**0x3145**	**Access violation**
Store [0x30316]	***0xF016***	***ok***
Load [0x01202]	***0xF002***	***0x22***
Store [0x31231]	***0xE031***	***Access violation***
Store [0x21202]	***N/A***	***Bad segment***
Load [0x11213]	***N/A***	***Segment overflow***
Load [0x01515]	***0x3015 or N/A***	***Access violation***

Segment Table (Segment limit = 3)

Seg #	Page Table Base	Max Pages in Segment	Segment State
0	0x2030	0x20	Valid
1	0x1020	0x10	Valid
2	0xF040	0x40	Invalid
3	0x4000	0x20	Valid

Physical Memory

Address	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
0x0000	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
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	03	41	01	30	01	31	01	00	03	00	00	00	00	00	00
0x1030	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF
0x1040	10	01	11	03	31	03	13	00	14	01	15	03	16	01	17	00
….																
0x2030	10	01	11	00	12	03	67	03	11	03	00	00	00	00	00	00
0x2040	02	20	03	30	04	40	05	50	01	60	03	70	08	80	09	90
0x2050	10	00	31	01	F0	03	F0	01	12	03	30	00	10	00	10	01
….																
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
0x3120	03	01	25	36	47	58	69	7A	8B	9C	AD	BE	CF	E0	F1	02
0x3130	04	15	26	37	48	59	70	7B	8C	9D	AE	BF	D0	E1	F2	03
….																
0x4000	30	00	31	01	11	01	F0	03	34	01	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	03	20	03	E0	01	E1	08	E2	86	28	03	48	25	34	21
….																
0xE000	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55
0xE010	A5	5A	A5	5A	A5	5A	A5	5A	A5	5A	A5	5A	A5	5A	A5	5A
….																
0xF000	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF
0xF010	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF	00
0xF020	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF	00	11
….																

[This page intentionally left blank]

[This page left for scratch]