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

Spring 2019 John Kubiatowicz

Midterm II
SOLUTIONS

April 4th, 2019
CS162: Operating Systems and Systems Programming

Your Name:	
Your SID:	
TA Name:	
Discussion Section Time:	

General Information:
This is a **closed book** exam. You are allowed 2 pages of notes (both sides). You have 2 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	20	
3	22	
4	18	
5	22	
Total	100	

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899

Problem 1: True/False [18 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]: A direct mapped cache can sometimes have a higher hit rate than a fully associative cache with an LRU replacement policy (on the same reference pattern).

[✓ True] ☐ False

Explain: *Consider a cache of N cache lines with an N+1 sequential access pattern whose addresses stride across cache lines (i.e. with 32-byte cache lines, address 0, 32, 64, ... 32N, 0, 32, 64, ... 32N, ...). The LRU cache would miss on every access (since the N+1st entry is never in the cache); the direct-mapped cache would miss on only 2 out of N+1 accesses, with only 0 and 32N conflicting.*

Problem 1b[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: *Threads fail to complete for many reasons other than cycles in the resource graph. For instance, a thread could go into an infinite loop independent of the Banker's algorithm.*

Problem 1c[2pts]: Even though most processors use a *physical address* to address the cache, it is possible to overlap the TLB lookup and cache access in hardware.

[✓ True] ☐ False

Explain: *Not all bits in a Virtual address are translated during mapping to Physical address. Particularly, the page offset stays the same. If the cache index fits entirely in the page offset, then the SRAM lookup of the cache can happen in parallel with the TLB lookup, with the Tag check happening after the TLB access finishes. [For instance, with 4K pages and a 16-way, 64K cache, the cache index would come entirely from the offset. For larger caches, those bits of the index that fit inside the offset could start the cache lookup, with the remainder happening after the TLB lookup completes.]*

Problem 1d[2pts]: The lottery scheduler prevents CPU starvation by assigning at least one ticket to each scheduled thread.

[✓ True] ☐ False

Explain: *By assigning at least one ticket to each scheduled thread, the kernel can ensure that the lottery scheduler will always give at least a little CPU to every thread (probabilistically).*

Problem 1e[2pts]: You can always reduce the number of page faults by increasing the amount of memory available to a process.

☐ True [✓ False]

Explain: *This result depends on the replacement policy. A FIFO page replacement policy exhibits Belady's anomaly, in which certain access patterns experience increased page faults with increased memory.*

Problem 1f[2pts]: The Shortest Remaining Time First (SRTF) algorithm is the best preemptive scheduling algorithm that can be implemented in an Operating System.

☐ True [✓ False]

Explain: *Since SRTF relies on knowledge of the future, it cannot be implemented. It is merely a strawman against which to compare practical scheduling algorithms.*

Problem 1g[2pts]: A Patch for the Meltdown security flaw can increase the cost of execution in Linux significantly (by over 800% in some reports) on some process/kernel combinations.

[✓ True] ☐ False

Explain: *The primary software patch for Meltdown involves separate page tables for a user and kernel (i.e. no more inaccessible kernel mappings in the user's page table). On processors which do not have process-ID tags in their TLB (or on OSes that don't use these bits), this means that every system call must flush the TLB twice – once on entrance to the kernel and once on exit. [The result is a huge performance hit – reported to be 800% in some cases…]*

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

[✓ True] ☐ False

Explain: *Since it is possible to create both locks and condition variables out of semaphores, once can simply build monitors out of semaphores, then use these monitors to do whatever we want—thus demonstrating the equivalence.*

Problem 1i[2pts]: Page Tables have an important advantage over simple Segmentation Tables (i.e. using base and bound) for virtual address translation in that they eliminate external fragmentation in the physical memory allocator.

[✓ True] ☐ False

Explain: *Since page tables operate on pages, which are of fixed size, the physical memory allocator can use any physical page for any part of a virtual address space; thus, there is no externally wasted physical memory. In a simple segmentation scheme, however, the chunks of physical memory vary widely in size (based on the size of the segments); thus, the physical memory allocator is forced to find variable sized chunks – leading to external fragmentation.*

Problem 2: Multiple Choice [20pts]

Problem 2a[2pts]: When a process asks the kernel for resources that cannot be granted without causing deadlock (as determined by the Banker's algorithm), the kernel must (*choose one)*:

A: ☐ Preempt the requested resources from another process and give them to the requesting process, thereby avoiding cycles in the resource dependency graph.

B: [✓] Put the requesting process to sleep until the resources become available.

C: ☐ Send a SIGKILL to the requesting process to prevent deadlock from occurring.

D: ☐ This question does not make sense because the Banker's algorithm is run only when a new process begins execution.

Problem 2b[2pts]: Suppose that two chess programs are running against one another on the same CPU with equal "nice" values. Also assume that the programs are given real-time limits on the total time they spend making decisions (similar to a real chess tournament). Why might one of the programs want to perform a lot of superfluous I/O operations? (*choose one):*

A: ☐ To hide the fact that the program was cheating by receiving information from a real chess master over the network.

B: ☐ The random completion time for these I/O operations will serve as an important source of randomness, thereby improving the chess playing heuristics.

C: [✓] This will split up the long-running heuristic computation into many short bursts, thereby causing the chess program to be classified as "interactive" by the scheduler and thus receiving higher priority than the competing program.

D: ☐ This question does not make sense, since the extra I/O operations will only slow down chess program.

Problem 2c[2pts]: A processor which provides "Precise Exceptions" is one in which (*choose one)*:

A: ☐ There is never any ambiguity as to *which* type of exception occurred in the user's program.

B: [✓] At the time that an exception handler begins execution, there is a well-defined instruction address in the instruction stream for which all prior instructions have committed their results and no following instructions (including the excepting one) have modified processor state.

C: ☐ Important exceptions are *synchronous* (always occurring at the same place in the instruction stream) as opposed to *asynchronous*. This helps during restart after the exception is handled.

D: ☐ The Operating System can disable out-of-order execution during critical sections of the user's program, thereby preventing instructions following an exception from being partially executed.

Problem 2d[2pts]: The Meltdown security flaw (made public in 2018) was able to gain access to protected information in the kernel because (*choose one)*:

A: ☐ Data-specific variability in the processing of system-calls (related to how Intel processors handled synchronous exceptions) by one process (the victim) allowed another process (the attacker) to successfully guess values stored in the kernel stack of the victim process.

B: ☐ A POSIX-compatible system call implemented by multiple operating systems neglected to properly check arguments and could thus be fooled into returning the contents of protected memory to users.

C: ☐ There was a wide-spread bug in the x86 architecture that caused certain loads to ignore kernel/user distinctions in the page table under the right circumstances. This allowed user-code to directly load and use data that was supposed to be protected in kernel space.

D: ☑ Many processors allowed timing windows in which illegal accesses could be performed speculatively and made to impact cache state – even though the speculatively loaded data was later squashed in the pipeline and could not be directly used.

Problem 2e[2pts]: Earliest Deadline First (EDF) scheduling has an important advantage over other scheduling schemes discussed in class because (*choose one)*:

A: ☐ It can hand out more total processor cycles to an asynchronously arriving mix of real-time and non-realtime tasks than other scheduling schemes.

B: ☑ It can allocate up to 100% of processor cycles to real-time periodic tasks while still providing a guarantee of meeting real-time deadlines.

C: ☐ It can operate non-preemptively and is thus simpler than many other scheduling schemes.

D: ☐ It can provide the lowest average responsiveness to a set of tasks under all circumstances—even in the presence of long-running computations.

Problem 2f[2pts]: What is the Clock Algorithm and where is it used? (*choose one)*

A: ☐ The Clock Algorithm provides fair queueing of CPU cycles within the Linux CFS scheduler; it uses virtual time to give each thread its fair fraction of cycles and is considered a simpler alternative to the Linux O(1) scheduler.

B: ☐ The Clock Algorithm helps to synchronize time over the network and is capable of synchronizing to better than 10ms over the scale of a metropolitan area. It is used to help timestamp all activities within the operating system.

C: ☑ The Clock Algorithm is used to select replacement pages in the virtual memory subsystem. It places physical pages in a giant ring and scans through them for idle pages by manipulating the hardware "use" bit. Pages in the ring are marked as "valid" and accessible by running programs until they are chosen for replacement.

D: ☐ The Clock Algorithm is used to select replacement pages in the virtual memory subsystem. It places pages into two groups – an active group that is mapped as "valid" and managed in a FIFO list and an inactive group that is mapped as "invalid" and managed as an LRU list.

Problem 2g[2pts]: Consider a computer system with the following parameters:

Variable	Measurement	Value
P_{TLB}	Probability of TLB miss	0.1
P_F	Probability of a page fault when a TLB miss occurs on user pages (assume page faults do not occur on page tables).	0.0002
P_{L1}	Probability of a first-level cache miss for all accesses	0.1
P_{L2}	Probability of a second-level cache miss for all accesses	0.0004
T_{TLB}	Time to access TLB (hit)	1 ns
T_{L1}	Time to access L1 cache (hit)	5ns
T_{L2}	Time to access L2 cache (hit)	20ns
T_M	Time to access DRAM	100ns
T_D	Time to transfer a page to/from disk	10 ms = 10,000,000 ns

The TLB is refilled automatically by the hardware on a miss. The 2-level page tables are kept in physical memory and are cached like other accesses. Assume that the costs of the page replacement algorithm and updates to the page table are included in the T_D measurement. Also assume that no dirty pages are replaced and that pages mapped on a page fault are not cached.

What is the effective access time (the time for an application program to do one memory reference) on this computer? Assume physical memory is 100% utilized and ignore any software overheads in the kernel.

A: ☐ $T_{TLB} + T_{L1} + P_{L1} \times (T_{L2} + P_{L2} \times T_M) + P_{TLB} \times P_F \times T_D$

B: ☐ $T_{TLB} + T_{L1} + P_{L1} \times (T_{L2} + P_{L2} \times T_M) + P_{TLB} \times \{2[T_{L1} + P_{L1} \times (T_{L2} + P_{L2} \times T_M)] + P_F \times T_D\}$

C: ☐ $T_{TLB} + T_{L1} + P_{L1} \times (T_{L2} + P_{L2} \times T_M) + P_{TLB} \times (2T_M + P_F \times T_D)$

D: ☑ $T_{TLB} + (1 - P_{TLB} \times P_F) \times [T_{L1} + P_{L1} \times (T_{L2} + P_{L2} \times T_M)] +$
$P_{TLB} \times \{2[T_{L1} + P_{L1} \times (T_{L2} + P_{L2} \times T_M)] + P_F \times (T_{L1} + T_{L2} + T_M + T_D)\}$

Problem 2h[3pts]: Debugging in Pintos can often be very difficult due to the complexity of the code base and often just staring at the code is an ineffective strategy, *Mark all of the following that are good general strategies to debug Pintos:*

A: ☑ Setting breakpoints in GDB to walk through specific function calls.

B: ☑ Setting memory watchpoints in GDB to watch for variable modifications.

C: ☑ Checking function pre- and post-conditions with ASSERT statements.

D: ☑ Checking execution progress with PANIC statements.

E: ☑ Checking variables with print statements.

F: ☐ None of the above.

Problem 2i[3pts]: Which of the following schedulers gives priority to interactive processes (those getting input from users) over long-running ones and can be used in a workstation without requiring annotations from the programmer or other supplemental mechanisms (*mark all that apply):*

A: [✓] Linux CFS

B: ☐ SRTF

C: ☐ EDF

D: ☐ Round-Robin

E: [✓] Multi-Level Scheduler

F: ☐ None of the above.

Problem 3: Potpourri [22pts]

Problem 3a[8pts]: For the following problem, assume a hypothetical machine with 4 pages of physical memory and 7 pages of virtual memory. Given the access pattern:

`A B C D E A A E C F F G A C G D C F`

Indicate in the following table which pages are mapped to which physical pages for each of the following policies. Assume that a blank box matches the element to the left. We have given the FIFO policy as an example.

Access→		**A**	B	C	D	E	A	A	E	C	F	F	G	A	C	G	D	C	F
FIFO	1	A				E									C				
	2		B				A										D		
	3			C							F								
	4				D								G		*Any one of these*				
MIN	1	*A*																	*F*
	2		*B*			*E*					*F*		*G*						*F*
	3			*C*															*F*
	4				*D*														*F*
LRU	1	*A*				*E*								*A*					*F*
	2		*B*				*A*						*G*						
	3			*C*															
	4				*D*						*F*						*D*		

Problem 3b[2pts]: Suppose that you decide to try running PintOS on a real piece of hardware. Everything seems to work at first, but after testing for some time in user mode you realize that none of your syscalls are working. It turns out that the CPU you're testing is faulty; for some reason, the `int` (interrupt) instruction is treated as a no-op by your CPU. In one sentence, briefly explain how you could modify the Pintos kernel in order to service syscalls if the int instruction does not work on your CPU. Your modification is allowed reduce the usability of the CPU in other ways.

You could change any of the synchronous exception handlers (e.g. divide by zero, bad instruction opcode, page fault to a particular unmapped address, etc) to handle system calls. You could also potentially use x86 features like the sysenter/sysexit or call gate mechanisms to handle syscalls. The only requirements are that the operation synchronously enter the kernel and be recognizable as the intended syscall rather than a fault that kills the process. So, if you did a divide-by-zero, the kernel would have to notice that the divide came from libc() to be recognized as a syscall rather than error.

Problem 3c[2pts]: Assuming that you made the above modification to the kernel correctly, show an example of x86 code that a user could write in order to call the practice() system call, which takes one argument that is stored in register eax, and has a syscall number of 1. Do not worry about a return value or the exact syntax of x86 assembly code.

```
Practice:

    push eax      #push argument onto stack___
    push 1        #push syscall number onto stack
    move ebx, 0   #setup for divide by zero______
    div ebx       # divide by zero (syscall!)____
```

Problem 3d[4pts]: You are designing a kernel from scratch and need to make sure that system calls properly check user memory. Implement `validate_buffer()` so that it ensures that the memory between ptr and ptr + size - 1 points to valid user memory.

Assume you have access to the following:

```
size_t PGSIZE; // the size of a page

/* returns the byte offset of ptr within its page */
size_t pg_ofs(void *ptr);

/* returns true if the pointer is on a mapped, userspace page */
int validate_pointer(void *ptr);
```

Finish the implementation of validate_buffer() by adding code to the missing lines, below. Your code should be as **efficient** as possible and there should be no more than one semicolon per line. You will not be given full credit for solutions that check one byte at a time. *Also, you cannot write any control flow statements that we have not laid out for you (e.g. if, while, for).*

```
// Ensure memory between ptr and ptr+size-1 is valid user memory
void validate_buffer(void *ptr, size_t size)
{
   while (size > 0) {

      if (  !validate_pointer(ptr)    )
         thread_exit();     // Invalid buffer

      size_t bytes_validated;

      size_t page_remaining =   PGSIZE - pg_ofs(ptr);

      if (page_remaining > size)
         bytes_validated = size;
      else

         bytes_validated = page_remaining;

      ptr += bytes_validated;

      __________________________________________

      size -= bytes_validated;
   }
}
```

Problem 3e[6pts]:
Here is a table of processes and their associated arrival and running times.

Process ID	Arrival Time	CPU Running Time
Process A	0	2
Process B	1	6
Process C	4	1
Process D	7	4
Process E	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, that new processes are available for scheduling as soon as they arrive, 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	*A*	*A*	*A*
1	*A*	*A*	*B*
2	*B*	*B*	*A*
3	*B*	*B*	*B*
4	*B*	*C*	*C*
5	*B*	*B*	*B*
6	*B*	*B*	*B*
7	*B*	*B*	*D*
8	*C*	*B*	*E*
9	*D*	*E*	*B*
10	*D*	*E*	*D*
11	*D*	*E*	*E*
12	*D*	*D*	*B*
13	*E*	*D*	*D*
14	*E*	*D*	*E*
15	*E*	*D*	*D*

[This page intentionally left blank!]

Problem 4: Deadlock and the Alien Nosh [18pts]

Consider a large table with multi-armed aliens. They are bilaterally symmetric, *with the same number of left and right appendages*. In the center of the table is a pile of Foons and Sporks. In order to eat, they must have a "Foon" in each left hand and a "Spork" in each right hand. The aliens are so busy talking that *they can only grab one Foon or Spork at a time.* In this problem, we are going to use monitor-style programming in combination with the Banker's Algorithm to design a deadlock-free API that allows an alien to grab utensils one at a time and to return all utensils after eating. The prototypes for our API calls are as follows:

```
// Allocate and initialize a new table (numarms is number on each side!)
alien_table_t *InitTable(int numAliens, int numArms,
                         int numFoons, int numSporks);

// Allow alien to grab utensil.  Sleep if insufficient resources.
// utype = 0 for a Foon and 1 for a Spork
int GrabOne(alien_table_t *table, int alien, int utype);

// Return all of the given alien's utensils to center table.
// Wake sleeping aliens.
void ReturnAll(alien_table_t *table, int alien);

// Return true(1) if alien can be granted another utype utensil.
// Return false(0) otherwise
int BankerCheck(alien_table_t *table, int alien, int utype);
```

In the following problems, assume that you have `LockAcquire(lock_t *somelock)`, and `LockRelease(lock_t *somelock)` to manipulate locks. Also assume that condition variables are Mesa scheduled and that you can utilize condition variables via `CVWait(CV_t *someCV)`, `CVSignal(CV_t *someCV)`, and `CVbroadcast(CV_t *comeCV)`. You may also find `calloc()` useful:

```
// Return zeroed array of nmemb structures of given size.  This combines
// malloc() with a block zeroing process.
void *calloc(size_t nmemb, size_t size);
```

Problem 4a(2pts): Fill in missing lines of code at position (9), for data structure definitions. No blank line should have more than one semicolon (you do not need to fill all lines):

```
 1. typedef struct alien {
 2.    int utensils[2];
 3. } alien_t;

 4. typedef struct alien_table {
 5.    lock_t *mylock;
 6.    CV_t *myCV;
 7.    int numaliens, numarms;
 8.    int utensils[2];

 9.    alien_t *aliens;
       ____________________________________
10. } alien_table_t;
```

Problem 4b(2pts): Fill in missing lines of code for the `InitTable()` code at positions (13) and (20). No blank line should have more than one semicolon (you do not need to fill all lines).

```
11.  alien_table_t *InitTable(int numAliens, int numArms,
                              int numFoons, in numSporks)
12.  {

13.     alien_table_t *myTable = malloc(sizeof(alien_table_t));
14.     myTable->mylock = get_new_lock();            // Alloc lock
15.     myTable->myCV = get_new_CV(myTable->mylock); // Alloc CV
16.     myTable->numaliens = numAliens;
17.     myTable->numarms = numArms;    // number of arms on one side
18.     myTable->utensils[0] = numFoons;
19.     myTable->utensils[1] = numSporks;

20.     myTable->aliens = (alien_t*)calloc(numAliens,sizeof(alien_t));



21.     return (myTable);
22.  }
```

Problem 4c(4pts): Fill in missing code for the `GrabOne()` routine at position (25). No blank line should have more than one semicolon (you do not need to fill all lines). Do not forget to sleep if you have insufficient resources. *Solutions that do not use the* `BankerCheck()` *routine will receive no credit (you will design* `BankerCheck()` *in problem 4e).*

```
23.  void GrabOne(alien_table_t *table, int alien, int utype) {
24.     LockAcquire(table->mylock);

25.     while (!BankerCheck(table, alien, utype))

           CVWait(table->myCV);

        table->aliens[alien].utensils[utype]++;

        table->utensils[utype]--;



26.     LockRelease(table->mylock);
27.  }
```

Problem 4d(4pts): Fill in missing code for the `ReturnAll()` routine at position (30). No blank line should have more than one semicolon (you do not need to fill all lines).

```
28. void ReturnAll(alien_table_t *table, int alien) {
29.     LockAcquire(table->mylock);

30.     for (int j = 0; j < 2; j++) {

            table->utensils[j] += table->aliens[alien].utensils[j];

            table->aliens[alien].utensils[j] = 0;

        }

        CVbroadcast(table->myCV);

        ______________________________________________________________

31.     LockRelease(table->mylock);
32. }
```

Problem 4e[6pts]: Finally, implement `BankerCheck()` by filling out missing code at positions (37), (39), and (41). No blank line should have more than one semicolon (you do not need to fill all lines). This routine should return true (1) if the given Alien can be granted an additional resource of type utype. Return false(0) otherwise. Do not blindly implement the Banker's algorithm: this method only needs to have the same external behavior as the Banker's algorithm for this application and can be implemented with a single pass through the Aliens (since all aliens are identical). This method can be written with as few as 5 semicolons.

```
33. int BankerCheck(alien_table_t *table, int alien, int utype) {
34.     int result = 0;
35.     if (table->utensils[utype] <= 0)
36.        return result;

37.     table->aliens[alien].utensils[utype]++;

        table->utensils[utype]--;

38.     for (int i = 0; i < table->numaliens; i++) {

39.        if ((table->aliens[i].utensils[0]+table->utensils[0]>=table->numarms)&&

               (table->aliens[i].utensils[1]+table->utensils[1]>=table->numarms))

             result = 1;
40.     }

41.     table->utensils[utype]++;

        table->aliens[alien].utensils[utype]--;

42.     return (result);
43. }
```

[This page intentionally left blank!]

Problem 5: Virtual Memory [22 pts]

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

Virtual Page #1 (10 bits)	Virtual Page #2 (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	0	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).

Problem 5a[2pts]: What is the total maximum size of a page table (in bytes) for this scheme? Explain. *Note: the phrase "page table" in this question means the multi-level data structure that maps virtual addresses to physical addresses.*

*Since each Virtual Page # is 10 bits and the PTE is 4 bytes, this means that each level of the page table consists of 2^{10}x4 bytes = 4096 bytes = 1 page. Thus, the top-level will have one page, and there will be $2^{10=}$1024 next level pages for a total of (1 + 1024) * 4096 = 4096+2^{22} = 4,198,400 bytes.*

Problem 5b[3pts]: Suppose that you were interested in supporting more than 4GB (i.e. 2^{32} bytes) of physical memory, while still supporting a 32-bit virtual address space. Also suppose that you wanted to stick with 4-byte PTEs with the same *hardware-defined* page control and status bits as defined above. What is the maximum amount of physical memory you could easily support? Explain.

Looking at the PTE, there are 3 OS defined bits and 2 bits that are stuck at zero, leaving 5 bits for expansion. Thus, if we made the Physical Page number consist of 25 bits instead of 20 bits, we could get 32 times as much physical memory => 128GB of physical memory.

Problem 5c[2pts]: When a modern x86 processor translates a virtual address, where does the segment identifier come from?

The segment identifier comes from the instruction opcode bits. Thus, for instance, a "push" instruction implicitly references the stack segment, SS.

Problem 5d[5pts]: Assume the memory translation scheme given above (5a). Suppose that the base table pointer for the current ***user level process*** is `0x00200000`. Translate the following virtual addresses to physical addresses, using the memory contents given on the next page (page 19). We have filled in some of the boxes for you. Please fill in the empty/missing information. Table entries are in Hexadecimal (*Hint: don't forget that hexadecimal digits contain 4 bits!*)

Starting Virtual Address	Virtual Page #1	Virtual Page #2	First-Level PTE	Next-Level Page-Table Address	Second-Level PTE	Final Physical Address
0x00001047	0x0	*0x1*	*0x001FF007*	0x001FF000	0x00001065	0x00001047
0x00C07665	0x3	0x7	*0x00103007*	*0x00103000*	*0xEEFF0067*	*0xEEFF0665*
0xFFFFF555	*0x3FF*	0x3FF	*0x001FF007*	*0x001FF000*	0x00004065	*0x00004555*

Problem 5e[10pts]: Using same assumptions and memory contents as in (5d), predict results for the following instructions. Addresses are virtual. 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.**

Instruction	Result
`Load [0x00001047]`	`0x04`
`Store [0x00C07665]`	`ok`
`Store [0x00C005FF]`	`ERROR: read-only`
`Load [0x00003042]`	*`0x03`*

Instruction	Result
`Test-And-Set [0x02006047]`	*`0x50`*
`Load [0xFF80078F]`	*`ERROR: invalid`*
`Load [0xFFFFF005]`	*`0x01`*
`Store [0xFFFFF006]`	*`ERROR: read-only`*

Physical Memory in Bytes [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	02	31	03	13	04	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	00	00	11	01	11	01	33	03	34	01	35	00	43	38	32	79
00004010	50	28	84	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	65	00	00	20	67	00	00	30	00	00	00	40	07
00100010	00	00	50	03	00	00	00	00	00	00	20	67	00	00	00	00
…																
00102000	00	00	20	01	00	00	00	00	00	00	30	03	00	00	40	07
…																
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	20	00	00	00	10	65	00	00	10	67	00	00	20	65
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	00	40	65
…																
00200000	00	1F	F0	07	00	10	10	07	00	10	20	07	00	10	30	07
00200010	00	10	20	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
…																

[Draw a Logo for CS162 Here!]

[Scratch Page: Do not put answers here!]

[Scratch Page: Do not put answers here!]