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!

<table>
<thead>
<tr>
<th>Problem</th>
<th>Possible</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
[ This page left for $\pi$ ]

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899
Problem 1: True/False [18 pts]
Please \textit{EXPLAIN} your answer in TWO SENTENCES OR LESS (Answers longer than this may not get credit!). Also, answers without an explanation \textit{GET NO CREDIT}.

\textbf{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).

\[ \square \text{True} \quad \square \text{False} \]
Explain:

\textbf{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.

\[ \square \text{True} \quad \square \text{False} \]
Explain:

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

\[ \square \text{True} \quad \square \text{False} \]
Explain:

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

\[ \square \text{True} \quad \square \text{False} \]
Explain:

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

\[ \square \text{True} \quad \square \text{False} \]
Explain:
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:

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:

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

☐ True  ☐ False

Explain:

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:
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 widespread 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:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Measurement</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{TLB}$</td>
<td>Probability of TLB miss</td>
<td>0.1</td>
</tr>
<tr>
<td>$P_F$</td>
<td>Probability of a page fault when a TLB miss occurs on user pages</td>
<td>0.0002</td>
</tr>
<tr>
<td>$P_{L1}$</td>
<td>Probability of a first-level cache miss for all accesses</td>
<td>0.1</td>
</tr>
<tr>
<td>$P_{L2}$</td>
<td>Probability of a second-level cache miss for all accesses</td>
<td>0.0004</td>
</tr>
<tr>
<td>$T_{TLB}$</td>
<td>Time to access TLB (hit)</td>
<td>1 ns</td>
</tr>
<tr>
<td>$T_{L1}$</td>
<td>Time to access L1 cache (hit)</td>
<td>5 ns</td>
</tr>
<tr>
<td>$T_{L2}$</td>
<td>Time to access L2 cache (hit)</td>
<td>20ns</td>
</tr>
<tr>
<td>$T_M$</td>
<td>Time to access DRAM</td>
<td>100ns</td>
</tr>
<tr>
<td>$T_D$</td>
<td>Time to transfer a page to/from disk</td>
<td>10 ms  = 10,000,000 ns</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Access→</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>A</th>
<th>A</th>
<th>E</th>
<th>C</th>
<th>F</th>
<th>F</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>D</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO</td>
<td>1</td>
<td>A</td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td>A</td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td></td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIN</td>
<td>3</td>
<td></td>
<td>C</td>
<td></td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td>D</td>
<td></td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LRU</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

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:

_________________________

_________________________

_________________________

_________________________
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:

```c
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).*

```c
// Ensure memory between ptr and ptr+size-1 is valid user memory
void validate_buffer(void *ptr, size_t size) {
    while (size > 0) {
        if (__________________________)
            thread_exit(); // Invalid buffer

        size_t bytes_validated;

        size_t page_remaining = __________________________________

        if (page_remaining > size)
            bytes_validated = size;
        else
            __________________________
            __________________________
            __________________________

        size -= bytes_validated;
    }
}
```
**Problem 3e[6pts]:**
Here is a table of processes and their associated arrival and running times.

<table>
<thead>
<tr>
<th>Process ID</th>
<th>Arrival Time</th>
<th>CPU Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process A</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Process B</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Process C</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Process D</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>Process E</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

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.*

<table>
<thead>
<tr>
<th>Time Slot</th>
<th>FCFS</th>
<th>SRTF</th>
<th>RR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
[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:

```c
// 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:

```c
// 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):

```c
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.   ________________________________________________

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 = ______________________________________________
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.    ______________________________________________

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

25.    ______________________________________________

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).

```c
28. void ReturnAll(alien_table_t *table, int alien) {
29.   LockAcquire(table->mylock);
30.____________________________________________________________________
31.____________________________________________________________________
32.____________________________________________________________________
33.____________________________________________________________________
34.____________________________________________________________________
35.____________________________________________________________________
36.____________________________________________________________________
37.____________________________________________________________________
38.____________________________________________________________________
39.____________________________________________________________________
40. }  
41. }
```

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.

```c
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.____________________________________________________________________
38.____________________________________________________________________
39.____________________________________________________________________
40. for (int i = 0; i < table->numaliens; i++) {
41.____________________________________________________________________
42.   return (result);
43. }
```
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.*

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.

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

<table>
<thead>
<tr>
<th>Starting Virtual Address</th>
<th>Virtual Page #1</th>
<th>Virtual Page #2</th>
<th>First-Level PTE</th>
<th>Next-Level Page-Table Address</th>
<th>Second-Level PTE</th>
<th>Final Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00001047</td>
<td>0x0</td>
<td></td>
<td></td>
<td>0x001FF000</td>
<td>0x00001065</td>
<td>0x00001047</td>
</tr>
<tr>
<td>0x00C07665</td>
<td>0x3</td>
<td>0x7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFFFFF555</td>
<td>0x3FF</td>
<td></td>
<td></td>
<td>0x000004065</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result</th>
<th>Instruction</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load [0x00001047]</td>
<td>0x04</td>
<td>Test-And-Set [0x02006047]</td>
<td></td>
</tr>
<tr>
<td>Store [0x00C07665]</td>
<td>ok</td>
<td>Load [0xFF80078F]</td>
<td></td>
</tr>
<tr>
<td>Store [0x00C005FF]</td>
<td>ERROR: read-only</td>
<td>Load [0xFFFFF005]</td>
<td></td>
</tr>
<tr>
<td>Load [0x00003042]</td>
<td></td>
<td>Store [0xFFFFF006]</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>+0</td>
<td>+1</td>
<td>+2</td>
</tr>
<tr>
<td>-----------</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>000000000</td>
<td>0E</td>
<td>0F</td>
<td>10</td>
</tr>
<tr>
<td>00000010</td>
<td>1E</td>
<td>1F</td>
<td>20</td>
</tr>
<tr>
<td>00001010</td>
<td>40</td>
<td>41</td>
<td>42</td>
</tr>
<tr>
<td>00001020</td>
<td>40</td>
<td>03</td>
<td>41</td>
</tr>
<tr>
<td>00001030</td>
<td>00</td>
<td>11</td>
<td>22</td>
</tr>
<tr>
<td>00001040</td>
<td>10</td>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>00002030</td>
<td>10</td>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>00002040</td>
<td>02</td>
<td>20</td>
<td>03</td>
</tr>
<tr>
<td>00002050</td>
<td>10</td>
<td>00</td>
<td>31</td>
</tr>
<tr>
<td>00004000</td>
<td>00</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>00004010</td>
<td>50</td>
<td>28</td>
<td>84</td>
</tr>
<tr>
<td>00004020</td>
<td>23</td>
<td>03</td>
<td>20</td>
</tr>
<tr>
<td>00100000</td>
<td>00</td>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td>00100010</td>
<td>00</td>
<td>00</td>
<td>50</td>
</tr>
<tr>
<td>00102000</td>
<td>00</td>
<td>00</td>
<td>20</td>
</tr>
<tr>
<td>00103000</td>
<td>11</td>
<td>22</td>
<td>00</td>
</tr>
<tr>
<td>00103010</td>
<td>22</td>
<td>33</td>
<td>44</td>
</tr>
<tr>
<td>001FE0000</td>
<td>04</td>
<td>15</td>
<td>00</td>
</tr>
<tr>
<td>001FE0100</td>
<td>10</td>
<td>15</td>
<td>00</td>
</tr>
<tr>
<td>001FF0000</td>
<td>00</td>
<td>00</td>
<td>20</td>
</tr>
<tr>
<td>001FF0100</td>
<td>00</td>
<td>00</td>
<td>20</td>
</tr>
<tr>
<td>001FFFF0</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00200000</td>
<td>00</td>
<td>1F</td>
<td>F0</td>
</tr>
<tr>
<td>00200010</td>
<td>00</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>00200020</td>
<td>00</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>00200FF0</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>
[Draw a Logo for CS162 Here!]
[Scratch Page: Do not put answers here!]
[Scratch Page: Do not put answers here!]