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

Spring 2023 Kubiatowicz

Midterm II

March 15th, 2023

CS162: Operating Systems and Systems Programming

Your Name:	
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 110 minutes 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.

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	20	
2	20	
3	22	
4	18	
5	20	
Total	100	

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899

Problem 1: True/False and Why[20 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]: In systems which provide multiple segments at a time, the segment identifier is always taken from a portion of the virtual address.

□ True □ False

Explain:

Problem 1b[2pts]: A system that is in a SAFE state is guaranteed to eventually complete execution of all threads.

□ True □ False

Explain:

Problem 1c[2pts]: The CFS scheduler makes sure that every runnable thread gets the CPU for a minimum amount of time.

□ True □ False

Explain:

Problem 1d[2pts]: The `fork()` system call must allocate new physical memory for the child process and copy all of the contents of the parent's address space into this new memory in order ensure that parent and child can subsequently modify memory contents without affecting each other.

□ True □ False

Explain:

Problem 1e[2pts]: A user process could generate a page fault while performing a load or store operation to a memory address that it has access to.

□ True □ False

Explain:

Problem 1f[2pts]: In PintOS, there is a single kernel thread that handles all of the system calls made by user threads in a process.

□ True □ False

Explain:

Problem 1g[2pts]: In PintOS, the timer_interrupt function (which handles timer interrupts) does not need to be reentrant (i.e. handle the case in which more than one timer interrupt event is being handled at the same time).

□ True □ False

Explain:

Problem 1h[2pts]: Priority donation is a mechanism whereby a non-interactive thread gives its priority to an interactive thread, thereby raising the priority of the interactive thread and making it more responsive.

□ True □ False

Explain:

Problem 1i[2pts]: A system which will be used to perform hard realtime scheduling needs to have an accurate measure for the worst-case execution time (WCET) of every task that will be run.

□ True □ False

Explain:

Problem 1j[2pts]: A system with an inverted page-table uses an amount of memory for the page table that scales with the size of the physical memory.

□ True □ False

Explain:

Problem 2: Multiple Choice [20pts]

Problem 2a[2pts]: Consider the following pseudocode for three threads (T1, T2, T3) that try to acquire three locks (A, B, C):

```
   T1 {                  T2 {                  T3 {
L1.   acquire A       L2.   acquire B       L4.   acquire C
L3.   acquire B       L5.   acquire C       L6.   acquire A
      acquire C             acquire A             acquire B
      release C             release A             release B
      release B             release C             release A
      release A             release B             release C
   }                     }                     }
```

Assume the code executes lines L1, L2, L3, L4, L5, L6, in that order. After line L6, the system is deadlocked. After executing which line did the system enter an unsafe state? *(Select one):*

A: ☐ L1
B: ☐ L2
C: ☐ L3
D: ☐ L4
E: ☐ L5

Problem 2b[2pts]: Select all of the following that are true of page tables (*choose all that apply*):

A: ☐ Each thread within a process has its own page table.

B: ☐ The number of PTEs in a simple, single-level page table is proportional to the size of virtual memory.

C: ☐ Multi-level page tables are more memory-efficient than single-level page tables for sparse address spaces.

D: ☐ In a system with 2^{16} physical pages, a page table entry will consist of exactly 16 bits.

E: ☐ Two different page tables can reference the same physical page.

Problem 2c[2pts]: Which of the following is true about a multi-level feedback queue (MLFQ)? (*choose all that apply*):

A: ☐ Starvation is never a problem for a MLFQ.

B: ☐ Short-running tasks are given higher priority.

C: ☐ A MLFQ is an approximation of shortest remaining time first.

D: ☐ A MLFQ is less fair than round robin.

E: ☐ One way to keep a job's priority high in a MLFQ is to insert a bunch of short sleep operations.

Problem 2d[2pts]: Which of the following are true about CFS? (*Choose all that apply*):

A: ☐ CFS attempts to equalize the virtual CPU time for all jobs.
B: ☐ Giving a thread a lower nice value results in CFS allocating a lower share of physical CPU time to that thread.
C: ☐ Adding new thread to the run queue in CFS takes *O(1)* time.
D: ☐ The target latency in CFS is the period of time over which every job should get service.
E: ☐ In CFS, higher-weight threads accumulate less virtual CPU time per physical CPU cycle than lower-weight threads.

Problem 2e[2pts]: One of the following statements about schedulers is false. Which one is it? (*choose the false statement)*:

A: ☐ FCFS maximizes throughput of threads (where throughput represents the average amount of actual work being done by threads).
B: ☐ SRTF minimizes average completion time of threads.
C: ☐ The average throughput of threads running under a RR scheduler (regardless of quanta) is always greater than or equal to the average throughput of threads running with FCFS.
D: ☐ The average throughput of threads running under a RR scheduler with quantum q=2 is always greater than or equal to the average throughput of threads running under RR with q=1.
E: ☐ If threads arrive in increasing order of runtime, then the FCFS and SRTF schedulers will have the same average completion times.

Problem 2f[2pts]: Which of the following is true about TLBs? (*choose all that apply):*

A: ☐ For physically addressed caches, TLB access can occur in parallel with cache lookup, even though the physical address is required to complete cache access.
B: ☐ The TLB is always structured as a single-level, fully-associative caching structure for performance reasons.
C: ☐ During a context switch from a thread in one process to a thread in another, the contents of the TLB must saved into the PCB of the first process and loaded from the PCB of the second.
D: ☐ During a TLB miss on an Intel x86 processor, the OS must walk the page table to find the missing translation.
E: ☐ System with inverted page tables cannot benefit from the addition of a TLB.

Problem 2g[2pts]: Which of the following schedulers attempts to automatically give priority to interactive processes (those getting input from users) over long-running ones and can be actually implemented in a laptop? (*Choose all that apply):*

A: ☐ Lottery Scheduler
B: ☐ SRTF
C: ☐ Linux CFS
D: ☐ Round-Robin
E: ☐ MLFQ

Problem 2h[2pts]: Consider a system with a priority-based scheduler that provides priority donation. Suppose that the following events occur in this order:

1. Thread A is created with priority 3 and calls acquire on locks lock_1 and lock_2.
2. Thread B is created with priority 6 and calls acquire on lock_1.
3. Thread C is created with priority 10 and calls acquire on lock_2.
4. Thread A releases lock_1.
5. Thread A releases lock_2.

Which of the following sequences represent the evolution of Thread A's effective priority? (*Choose one):*

A: ☐ 3, 6, 10, 10, 3

B: ☐ 3, 6, 10, 6, 3

C: ☐ 3, 3, 10, 6, 3

D: ☐ 3, 6, 10, 3, 3

E: ☐ 3, 6, 6, 3, 3

Problem 2i[2pts]: Which of the following is *not true* about the EDF scheduler? (*Choose the incorrect statement):*

A: ☐ It is specified for periodic tasks with well-defined worse-case execution time (WCET).

B: ☐ It can allocate up to 100% of processor cycles while still meeting deadlines.

C: ☐ It is a real-time scheduler.

D: ☐ It can run a non-realtime tasks in the background as long as the total utilization of the realtime tasks is < 1.

E: ☐ It operates by selecting the task with the shortest remaining computation time to run next.

Problem 2j[2pts]: Which of the following are potential ways to avoid (reduce) the total number of cache misses in a particular situation? *(Choose all that apply):*

A: ☐ Increase cache size.

B: ☐ Increase cache associativity.

C: ☐ Decrease cache associativity.

D: ☐ Prefetching.

E: ☐ Rearrange the layout of data structures.

Problem 3: Deadlock and the Cephalopod Banquet [22pts]

Problem 3a[4pts]: Name and define the four conditions for deadlock ***(One sentence each):***

Problem 3b[2pts]: Suppose that we utilize the Banker's algorithm to determine whether or not to grant resource requests to threads. The job of the Banker's algorithm is to keep the system in a "SAFE" state. What is a SAFE state? ***(No more than two sentences):***

Problem 3c[2pts]: Explain how the Banker's algorithm prevents deadlock by removing one or more of the conditions of deadlock from (3a). Be explicit. ***(No more than two sentences):***

Problem 3d[4pts]: Suppose that we wish to evaluate the current state of the system and declare whether or not it is in a SAFE state. If there were only two types of resource, we could describe the state of the system with the following data structures:

```
typedef struct FreeRes {     // Track system-wide resources
   int FreeResA, FreeResB;   // Number of copies ResA and ResB free
} FRes_t;

typedef struct ThreadRes {   // Track resources for one thread
   int MaxNeedA, MaxNeedB;   // Max number each resource needed
   int CurHeldA, CurHeldB;   // Current number resources held
} TRes_t;

FRes_t curFree;              // Structure of all free resources
TRes_t curThreadRes[];       // Array of all thread resources
```

Assume that curFreeRes and curThreadRes[] have been initialized to reflect the current state of the system. Fill in the missing lines to complete our check for safety. ***Only one expression per line, no comma expressions or additional semicolons.***

```
 1. bool IsSAFE(FRes_t *curFreeRes, TRes_t curThreadRes[], int numThreads) {
 2.    int FreeA = curFreeRes->FreeResA;
 3.    int FreeB = curFreeRes->FreeResB;
 4.    bool ThreadDone[numThreads] = {false}; // init array to all false
 5.    int Remaining = numThreads; // Number threads not finished
 6.    bool needPass = true;       // Completion flag
 7.    while (needPass) {
 8.       needPass = false;  // No threads completed yet this iteration
 9.       for (int i = 0; i < numThreads; i++) {
10.          if (!ThreadDone[i]) { // Thread hasn't completed, thus check

11.             if ((_____________________________________________ <= FreeA) &&

12.                 (_____________________________________________ <= FreeB)) {

13.                _____________________________________________________;

14.                _____________________________________________________;

15.                ThreadDone[i] = _____________________________;

16.                needPass = _________________________________;

17.                Remaining = _________________________________;
                }
12.          }
13.       }
14.    }
15.    return (_____________________________________________);
16. }
```

The Cephalopod Diners Problem: Consider a large table with ***identical*** multi-even-armed cephalopods (e.g. octopuses). ***In the center is a pile of forks and knives.*** Before eating, each diner must have an equal number of forks and knives, one in each arm (e.g. if octopuses are eating, they would each need four forks and four knives). The creatures can only grab one utensil at a time. They grab utensils in a random order until they have enough utensils to eat. After they finish eating, they return all of their utensils at once. Diners are implemented as threads. Consider the following sketch for a Solution to the Cephalopod Diners problem using ***Mesa monitor synchronization:***

```
 1. typedef struct DinerUtensils {
 2.    int forks,knives;       // utensils held by creature
 3. } Diner_t;                 // clearly forks+Knives <= NumArms
 4. typedef struct CephTable {
 5.    struct lock lock;    // lock_init, lock_acquire, lock_release
 6.    struct condition CV; // cond_init, cond_wait, cond_signal, cond_broadcast
 7.    Diner_t *diners;     // Utensils for each diner
 8.    int numDiners, numArms;    // Number of diners and arms for each diner
 9.    int idleForks, idleKnives; // Number of forks/knives on table
10. } Table_t;
11. Table_t myTable;           // the current table!

12. // Initialize table. Arms must be even, both Forks and Knives >= Arms/2
13. void Init(Table_t *myTab, int Diners, int Arms, int Forks, int Knives) {
14.    lock_init(&(myTab->lock));
15.    cond_init(&(myTab->CV));
16.    // void *calloc(int n, size_t size) allocates num*size bytes set to 0
17.    myTab->diners = (Diner_t *)calloc(Diners, sizeof(Diner_t));
18.    myTab->numDiners = Diners;       // Number of Diners
18.    myTab->numArms = Arms;           // Number of arms per Diner
19.    myTab->idleForks = Forks;        // Number Forks on table initially
20.    myTab->idleKnives = Knives;      // Number Knives on table initially
21. }

22. // Called by diner "CephID" to return all their utensils to table
22. void DoneEating(Table_t *myTab, int CephID) {
23.    /* Return all utensils to the pile */
24.    lock_acquire(&(myTab->lock));
25.    myTab->idleForks += (myTab->diners)[CephID].forks;
26.    myTab->idleKnives += (myTab->diners)[CephID].knives;
27.    (myTab->diners)[CephID].forks = 0;
28.    (myTab->diners)[CephID].knives = 0;
29.    cond_broadcast(&(myTab->CV),&(myTab->lock));
30.    lock_release(&(myTab->lock));
31. }

32. // Called by diner "CephID" to grab a single utensil (fork or knife)
33. void GrabUtensil(Table_t *myTab, int CephID, boolean wantFork) {
33.    /* Implementation in Problem (3e) */
34. }

35. // Safety Check: Ok to give numforks forks and numknives knives to caller?
36. bool CephCheck(Table_t *myTab, int CephID, int numforks, int numknives) {
37.    /* Implementation in Problem (3g) */
38. }
```

Problem 3e[4pts]: Implement the code for the `GrabUtensil()` routine. It should return only after the request has been satisfied and sleep until it is ok. Assume that a Cephalopod will never request more resources than it needs. You can also assume that the `CephCheck()` routine returns true only if the requested number of forks and/or knives can be given to the particular Cephalopod without taking the system out of a safe state. `GrabUtensil()` should be able to deal with multiple threads accessing the state; implement this routine as a monitor and assume Mesa scheduling. Make sure your implementation is compatible with the provided `DoneEating()` routine. However, it is up to the Cephalopod thread to eat and subsequently call `DoneEating()`; you should not do that in `GrabUtensil()`. ***Only one expression per line, no comma expressions or additional semicolons. Do not worry about error handling code here.***

```
1.  // Called by diner "CephID" to grab a single utensil (fork or knife)
2.  void GrabUtensil(Table_t *myTab, int CephID, boolean wantFork) {
3.     int numforks = (wantFork?1:0);
4.     int numKnives = (wantFork?0:1);

5.     ____________________________________________________________;

6.     while (____________________________________________________) {

7.        ________________________________________________;
8.     }

9.     ____________________________________________________________;

10.    ____________________________________________________________;

11.    ____________________________________________________________;

12.    ____________________________________________________________;

13.    ____________________________________________________________;
14. }
```

Problem 3f[2pts]: In its general form, the Banker's algorithm makes a decision about whether or not to allow an allocation request by making multiple passes through the set of resource holders (threads). See, for instance, the nested loops in (3d). Explain why a Banker's algorithm *dedicated* to the Cephalopod Diners problem, namely the CephCheck() routine, could operate with a single pass through the resources holders. ***Provide two sentences or less.***

Problem 3g[4pts]: Finally, implement the `CephCheck()` method. This method should implement the Banker's algorithm: return true if the given Cephalopod can be granted '`numforks`' forks and '`numknives`' knives without taking the system out of a SAFE state. Assume that a Cephalopod will never request more resources than it needs. Do not worry about making this routing threadsafe; it will be called with a lock held by `GrabUtensil()`.

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. Your answer to **(3f)** is relevant here. ***This code should not permanently alter the local variables of the Table_t (although it can do so temporarily). Only one expression per line, no comma expressions, and no additional semicolons.*** *Hint: check the requesting Cephalopod, then the rest.*

```
1.  // Safety Check: Ok to give numforks forks and numknives knives to caller?
2.  bool CephCheck(Table_t *myTab, int CephID, int numforks, int numknives) {
3.     // Easy case first
4.     if (numforks > myTab->idleForks || numknives > myTab->idleKnives)
5.        return false;

6.     if (_____________________________________________________________ &&

7.         _____________________________________________________________ ) {

8.        return ____________________________________;

9.     }
10.    for (___________________; ___________________; ___________________) {

11.       if (_____________________________________________________________ &&

12.             _____________________________________________________________ ) {

13.            return ____________________________;
14.       }
15.    }
16.    return ____________________________________;
17. }
```

Problem 4: Short Answer Potpourri [18pts]

Problem 4a[3pts]: Rohan finished implementing priority scheduling, but he isn't sure if his code handles priority scheduling for waiters on a condition variable correctly. Specifically, condition variables in the starter code wake up waiters in FIFO order, and he forgot if he modified them to wake up the highest priority waiter first. Since he is too lazy to look at his code, he instead decides to write a test to prove that he made the necessary changes. Fill in the following code such that it prints a different output in the following two scenarios:

1. The condition variable wakes up waiters in FIFO order.
2. The condition variable wakes up the highest priority waiter first.

In both cases, you should assume that the preemptive priority scheduler is implemented correctly: at any given time, the highest priority thread that is awake is running. You should also assume that other synchronization primitives (i.e. locks and semaphores) correctly wake up waiters in priority order. You do not need to test priority donation. ***While you do not have to fill in every line, place no more than one expression per line, no comma expressions or additional semicolons:***

```
1. static thread_func priority_condvar_thread;
2. static struct lock lock;
3. static struct condition condition;

4. void test_priority_condvar(void) {
5.   lock_init(&lock);
6.   cond_init(&condition);
7.   thread_set_priority(0);
8.   for (int i = 0; i < 3; i++) {
9.     // Priority varies from 0 to 63, higher value=>higher priority

10.    int priority = ____________________________________________;
11.    char name[16];
12.    snprintf(name, sizeof name, "%d", i);
13.    // NOTE: `thread_create` yields if new thread has higher priority.
14.    thread_create(name, priority, priority_condvar_thread, NULL);
15.  }
16.  for (int i = 0; i < 3; i++) {
17.    lock_acquire(&lock);
18.    cond_signal(&condition, &lock);
19.    lock_release(&lock);
20.  }
21. }
22. static void priority_condvar_thread(void* aux UNUSED) {

23.  ____________________________________________;

24.  ____________________________________________;
25.  msg("Thread %s woke up.", thread_name());

36.  ____________________________________________;

37.  ____________________________________________;
38. }
```

Problem 4b[2pts]: Assuming the code of (**4a**), show how the two scenarios would differ:
What would be the output in Scenario 1?

What would be the output in Scenario 2?

Problem 4c[2pts]: Rohan thinks he can get away with removing the "`for`" loop for calling `cond_signal` (lines 16-20 of code in (**4a**)) and instead acquire the lock, call `cond_broadcast` once, then release the lock. Would the test still work? *Explain in two sentences or less.*

Problem 4d[4pts]: Consider a computer system with a two-level hardware cache. Ignore any virtual to physical translation or concerns about cache size or transfer time. Also, assume there are no page faults. Possibly useful parameters are as follows:

Variable	Measurement	Value
P_{L1H}	Probability of cache hit when going to the first level of the cache (L1).	90%
P_{L2H}	Probability of a cache hit when going to second level of the cache (L2).	95%
S_{L1}	Size of L1 cache.	64 KB
S_{L2}	Size of L2 cache.	2 MB
S_M	Size of DRAM	4 GB
S_D	Size of Disk	2 TB
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

Compute the Average Memory Access time (AMAT) and show your work. Give a symbolic expression followed by an actual value for AMAT. You should be able to compute this value without needing a calculator, however you can leave an unsimplified expression if necessary (Show your work!):

Problem 4e[2pts]: Provide one difference between how PintOS handles the main thread of a process calling pthread_exit() vs exit(). *Explain in two sentences or less.*

Problem 4f[2pts]: In PintOS, the `sema_init()` system call returns an integer. Why can't the `sema_init()` system call return the actual initialized `struct semaphore` to the user? *Explain in two sentences or less.*

Problem 4g[3pts]:
Five jobs are waiting to be run. Their expected running times are 10, 8, 3, 1, and X. In what order should they be run to minimize average completion time? State the scheduling algorithm that should be used AND the order in which the jobs should be run. *HINT: Your answer will explicitly depend on X.*

[This Page Intentionally Left Blank]

Problem 5: Address Translation [20 pts]

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

Problem 5a[2pts]: What is the size of a page in this machine? *Show your work.*

Problem 5b[2pts]: What is "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. *Explain in two sentences or less.*

Problem 5c[2pts]: Assume that we have a 64-bit processor which has the same page size as you gave in problem (5a) and the same 12 access control bits as given in the above PTE. Now, if 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 (5b) and that all levels of the page table are the same size—with the possible exception of the top-level. How many levels of page table would this imply? *Explain in two sentences or less.*

Problem 5d[2pts]: Returning to a 32-bit processor with the virtual address format and PTE as shown above (in **5a)**, 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 much memory would be devoted to the page table for this mapping (in bytes)? *Explain in two sentences or less.*

Problem 5e[2pts]: Given a 32-bit processor with the virtual address format and PTE as shown above (in **5a)**, and assuming that you can have as many processes (and page tables) as you like, what is the maximum amount of physical memory that this machine could divide up among these processes? *Explain in two sentences or less.*

Problem 5f[12pts]: Assume the translation scheme from (**5a**). Use the Physical Memory table given on the next page to predict what will happen with the following byte-oriented memory instructions. Assume a big-endian processor (i.e. the most significant byte of a 4-byte integer is stored first in memory). Assume that the base table pointer for the current ***user level process*** is `0x00200000`.

Fill in the missing entries in the following table. 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 hexadecimal digits contain 4 bits, so 10-bits is slightly more than two hexadecimal bytes! (2) PTEs are 4 bytes! (3) Make sure to look for duplicate lookup at (say) the first-level PTE!*

Instruction	First-Level PTE	Second-Level PTE	Physical Address	Result
`Load [0x00001047]`	`0x00100007`	`0x00002067`	`0x00002047`	
`Store [0x00C07665]`	`0x00103007`	`0xEEFF0067`	`0xEEFF0665`	
`Store [0x00C005FF]`	`0x00103007`	`0x11220005`		`ERROR: read-only`
`Load [0x00003012]`			`0x00004012`	`0x36`
`Store [0x02001345]`			`0x00002345`	`ok`
`Load [0xFF80078F]`		`0x04150000`		
`Test-And-Set [0xFFFFF005]`	`0x001FF007`		`0x00103005`	

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
…																

Function Reference Sheet

Here are a few function signatures for you

```
/* Pintos locks */
void lock_init(struct lock *lock);
void lock_acquire(struct lock *lock);
void lock_release(struct lock *lock);

/* Pintos semaphore interface */
void sema_init(struct semaphore *sema, unsigned value);
void sema_down(struct semaphore *sema);
void sema_up(struct semaphore *sema);

/* Pintos condition variables */
void cond_init(struct condition *cond);
void cond_wait(struct condition *cond, struct lock *lock);
void cond_signal(struct condition *cond, struct lock *lock);
void cond_broadcase(struct condition *cond, struct lock *lock);

/* Pintos Readers/Writers Locks */
void rw_lock_init(struct rw_lock*);
void rw_lock_acquire(struct rw_lock*, bool reader);
void rw_lock_release(struct rw_lock*, bool reader);

/* Pintos List */
void list_init(struct list *list);
struct list_elem *list_head(struct list *list);
struct list_elem *list_tail(struct list *list);
struct list_elem *list_begin(struct list *list);
struct list_elem *list_next(struct list_elem *elem);
struct list_elem *list_end(struct list *list);
struct list_elem *list_remove(struct list_elem *elem);
bool list_empty(struct list *list);
#define list_entry(LIST_ELEM, STRUCT, MEMBER) ...
void list_insert(struct list_elem *before, struct list_elem *elem);
void list_push_front(struct list *list, struct list_elem *elem);
void list_push_back(struct list *list, struct list_elem *elem);
```

[Scratch Page: Do not put answers here!]

[Scratch Page: Do not put answers here!]