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

Spring 2022 Joseph & Kubiatowicz

Midterm II
SOLUTION
March 17th, 2022
CS162: Operating Systems and Systems Programming

Your Name:	
SID AND Autograder Login (e.g. student042):	
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.

WRITE ALL OF YOUR ANSWERS IN YOUR ANSWER BOOK, NOT IN THIS BOOKLET! 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	18	
3	30	
4	12	
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]: Round Robin scheduling always results in a lower average response time than First Come First Serve scheduling.

□ True ☑ False

Explain: *If jobs happen to arrive in sorted order, smallest burst time first, then FCFS will produce the lowest average response time.*

Problem 1b[2pts]: The Banker's algorithm can be used to help avoid deadlocks (caused by resource cycles) from occurring between a set of threads.

☑ True □ False

Explain: *The Banker's algorithm double-checks each resource acquisition request to see if it might have the potential to cause deadlock, and delays such acquisitions until they can be done safely.*

Problem 1c[2pts]: Lottery scheduling is always more accurate at providing the intended proportional CPU shares than Stride scheduling.

□True ☑ False

Explain: *Lottery Scheduling relies on randomness to choose the next thread to run, so it provides exact results only on average. Stride scheduling provides a deterministic scheduling algorithm, that is more accurate over short periods of time.*

Problem 1d[2pts]: Memory management systems that use page-based memory allocation (for the lowest level) do not suffer from external fragmentation.

☑ True □ False

Explain: *External fragmentation occurs when different sized blocks are allocated from a larger chunk of memory, leading to a challenge of finding new blocks after the system runs for a while. Page-based memory allocation avoids this problem because all pages are the same size.*

Problem 1e[2pts]: A memory fault will always cause the faulting process to be terminated by the operating system.

□ True ☑ False

Explain: *If the OS can fixup the fault, then it can restart the process. A good example of this is a page-fault, in which the OS can load the missing page from disk, fix the page table, and restart the process.*

Problem 1f[2pts]: Priority donation temporarily decreases the priority of a thread.

□ True ☑ False

Explain: *Priority donation avoids priority inversion by temporarily* increasing *the priority of a thread holding a lock, not by decreasing it.*

Problem 1g[2pts]: Page faults are handled in hardware by the MMU.

□ True ☑ False

Explain: *Page faults are handled in software by the OS.*

Problem 1h[2pts]: Compulsory cache misses can never be avoided.

□ True ☑ False

Explain: *Compulsory cache misses can be avoided by prefetching.*

Problem 1i[2pts]: A good scheduling algorithm can achieve high throughput and low latency for any mix of jobs.

□ True ☑ False

Explain: *High throughput is achieved by switching infrequently, whereas low latency is best achieved by interrupting long-running threads and switching to threads with the shortest burst.*

Problem 1j[2pts]: A system that uses priority donation cannot deadlock and guarantees that all threads will eventually complete.

□ True ☑ False

Explain: *Priority donation helps prevent priority inversion. However, it will not prevent resource cycles involving threads running at the same priority.*

Problem 2: Multiple Choice [18pts]

Problem 2a[2pts]: Which of the following are true about priority inversion? (*choose all that apply)*:

A: ☐ Priority inversion could not occur if there were only two possible priorities for threads.

B: ☑ Priority donation is a solution to priority inversion.

C: ☑ Priority inversion could result in starvation of high priority threads.

D: ☐ Priority inversion is only a problem on a multi-core system.

E: ☑ Priority inversion would not happen in a first come first serve scheduler.

Problem 2b[2pts]: What are the conditions necessary for deadlock? (*choose all that apply*):

A: ☑ Circular wait condition between threads and resources.

B: ☐ Infinite number of each resource

C: ☑ Resources cannot be taken away from a thread until it releases them

D: ☑ Each resource can be held by only one thread

E: ☑ Threads will hold resources and wait while acquiring new ones

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: ☐ Long-running tasks are given higher priority.

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

D: ☐ A MLFQ is more 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 the Linux Completely Fair Scheduler (CFS) (*choose all that apply*):

A: ☐ CFS attempts to equalize the response time for all jobs.

B: ☑ A higher nice value results in CFS allocating a lower share of CPU time.

C: ☐ Finding a new thread to run in CFS takes *O(log n)* time.

D: ☑ The target latency in CFS is the period of time over which every job should get service.

E: ☐ Minimum granularity in CFS is the total number of nice values that are allowed.

Problem 2e[2pts]: Which of the following are potential ways to avoid cache misses? (*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 2f[2pts]: Which of the following are true about deadlocks? (*choose all that apply):*

A: ☑ Acquiring resources in a predetermined order is one way to avoid deadlock.

B: ☐ Starvation is a form of deadlock.

C: ☑ Rolling back threads is one way to address deadlock once it occurs.

D: ☐ Deadlock means that no threads are able to make progress.

E: ☐ Banker's algorithm is one way to recover from deadlock once it occurs.

Problem 2g[2pts]: Which of the following statements about memory management are true? (*choose all that apply):*

A: ☐ In general, MIN is the best page replacement policy that can be implemented.

B: ☐ A cache of 512 bytes always outperforms a cache of 256 bytes.

C: ☐ Adding more physical memory to a system will always reduce the page miss rate.

D: ☑ The Clock algorithm is an approximation of LRU.

E: ☑ A hardware implemented modified bit is optional.

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

Variable	Measurement	Value
P_{TLB}	Probability of TLB miss	App Dependent
P_F	Probability of a page fault when a TLB miss occurs on user pages (assume page faults do not occur on page tables).	App Dependent
P_{L1}	Probability of a first-level cache miss for all accesses	App Dependent
P_{L2}	Probability of a second-level cache miss for all accesses	App Dependent
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 operates in parallel with the cache lookup (at level 1) when there is a TLB hit.. Further, 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.

If the *full working set of the application fits into DRAM,* what is the effective access time (the time for an application program to do one memory reference) on this computer?

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

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

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

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

E: ☐ $T_{TLB} + T_{L1} + T_{L2} + T_M + T_D$

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

A: ☐ It can hand out more total processor cycles to 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.

E: ☐ None of the above

Problem 3: Short Answer [30pts]

Please write your answer in THREE SENTENCES OR LESS (Answers longer than this may not get credit!).

Problem 3a[4pts]: Suppose there are n jobs $J_1, J_2 \ldots J_n$ ready to run, with no other jobs in the system. Assume job J_i will take T_i time to finish. Furthermore, suppose $T_1 < T_2 < \ldots < T_n$. What order should the jobs be run to minimize average completion time? Provide a brief mathematical proof (*Hint: start by writing down an expression for average completion time).*

The jobs should be run with an FCFS scheduler in the order: $J_1, J_2 \ldots J_n$ to minimize average completion time.

Proof Sketch: Assume FCFS and that the final completion time for each of the jobs is $C_1, C_2 \ldots C_n$. If we schedule them in order $J_1, J_2 \ldots J_n$, then the, the average completion time is:

$$\frac{1}{n}\sum_j C_j = \frac{1}{n}\sum_{j=1}^{n}\sum_{k=1}^{j} T_k = \frac{1}{n}\sum_{j=n}^{1} j \times T_j$$

The middle term arises because, for instance, $C_1=T_1$, $C_2 = C_1+T_2=T_1+T_2$, etc..

Notice that this is the best order for FCFS, because any swapping of the order of the T_j in the final expression will increase the value (if we swap T_x and T_y then the sum will change by):

$$(n-x+1)\times(T_y-T_x)+(n-y+1)\times(T_x-T_y)=(y-x)\times(T_y-T_x)\geq 0$$

Further, anything other than FCFS (like RR) will only make the individual C_j larger by putting pieces of later threads before the completion.

Problem 3b[3pts]: In a typical cache, the tag-bits are pulled from the top (most significant bits) of the address, while index and offset bits are pulled from less significant bits. Why wouldn't it make sense to take the cache index from the most significant bits? Explain.

Because the resulting cache would not work well for systems with spatial locality that extended beyond the cache-line. For instance, with the index in the normal place, a set of accesses to an array that was bigger than a cache-line size could still fit into the cache (many different index values would be in use). In contrast, if the index were placed at the top of the address, the index bits would tend to stay constant when accessing such an array. Thus, ever array access that changed cache lines would automatically conflict with every other cache line.

Problem 3c[3pts]: You just brought an expensive gaming PC with a 64-bit virtual address space, 32 GiBs of RAM, an NVidia 3090ti GPU, and 16 TiB M.2 SSD. You decided to write a custom 64-bit OS for it. To reduce memory translation latency, you decided to implement a single-level page table based scheme, but will use a standard memory layout (stack grows down from top of address space, heap grows up). Assume each page is 4KiB. Will this approach work or not? Explain.

No, this won't work. To handle the sparse address space, we will need a full page table. The page table will need $2^{64}/2^{12}=2^{52}$ entries, one for each page in the address space. Further, the PTE will need 52 bits of physical page number, so will need to be at least 52/8=6.5 bytes. Rounding up to 8 bytes (for access control bits), we will need:
$2^{52}\times 8 = 2^{55}$bytes for the page table, which is 32 PiB (peta bytes!) of physical storage just for the page table of one process!

Problem 3d[3pts]: Before implementing a virtual memory system, you measure that accessing a word that is resident in physical memory takes about 10ns (with the hardware cache and TLB taken into account). You measure the time it takes the OS to service an access to an on-disk page (including page fault overhead, fetching the page from disk, and re-executing the instruction) to be about 10ms. What is the highest page fault rate we can tolerate to prevent the average memory access time from exceeding 20 ns? Explain how you calculated your answer.

Don't forget to equalize units by converting ms to ns:
AMAT = 10ns + $p_{fault} \times 10ms \times (10^6 ns/ms) = (10 + p_{fault} \times 10^7)$ ns ≤ 20 ns $\Rightarrow p_{fault} \leq 10^{-6}$
So, we want to keep our page fault rate under 0.0001%

Problem 3e[2pts]: In PINTOS, how do you calculate effective priority?

$$T_{Priority_{Eff}} = max\left(T_{Priority_{base}}, max\left(\left\{C_{Priority_{Eff}} \mid C \in T_{children}\right\}\right)\right)$$

Must mention effective priorities in PINTOS are computed as the max of the thread's base priority and the max of each child's effective priority.

Problem 3f[2pts]: For the PINTOS alarm clock, in **`timer_sleep()`** how is synchronization enforced on the list of waiting threads?

*Synchronization is enforced because **`timer_sleep()`** operates in a disabled interrupt context. No credit for any other answer.*

Problem 3g[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. You may break ties in any order you wish.

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						
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*		

Any one of these

Problem 3h[5pts]: 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	5
Process C	4	1
Process D	6	4

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*	*D*
7	*C*	*B*	*B*
8	*D*	*D*	*D*
9	*D*	*D*	*B*
10	*D*	*D*	*D*
11	*D*	*D*	*D*

Problem 4: Pintos Scheduling [12pts]

Marcus, inspired from reading through the Linux scheduler, decides to add in a simplified version of scheduling classes into Pintos. He decided that each thread will have a priority ranging from 0-139. Furthermore, the scheduler will be divided into two scheduling classes: Threads with priorities 0-99 will be scheduled FIFO within a priority, whereas threads with priorities 100-139 will utilize lottery scheduling.

Marcus decides that threads scheduled with FIFO will only be run after there are no more threads to be run in the lottery class. Assume threads with a larger priority number indicates a higher priority. Below are struct and function definitions within the kernel which may be helpful for this problem:

```
struct thread {
  ...
  int priority;
  struct list_elem elem;
  ...
}
static struct list ready_list_lottery;
static struct list ready_list_fifo[100];

// Calculate (and return) the number of tickets a thread with base priority
// 'priority' should receive. Don't worry about how this is implemented
int tickets_from_priority(int priority);

/* List operations */
#define list_entry(LIST_ELEM, STRUCT, MEMBER)
void list_push_back (struct list *, struct list_elem *);
struct list_elem *list_pop_front (struct list *);
struct list_elem *list_remove (struct list_elem *);
bool list_empty (struct list *);
```

Problems 4A-4L[1pt each]: Complete the implementation for the scheduler. Only one piece of code should be written per blank (i.e. no multiple statements with semicolons and no blank lines). Use proper C syntax with the given APIs. Pseudocode or comments will not receive any credit. You may only use methods and data structures given in this question as well as built-in ones. Finally, assume that calls to any given method will succeed and that **`ready_list_lottery`** and every sub-list of **`ready_list_fifo[]`** are initialized:

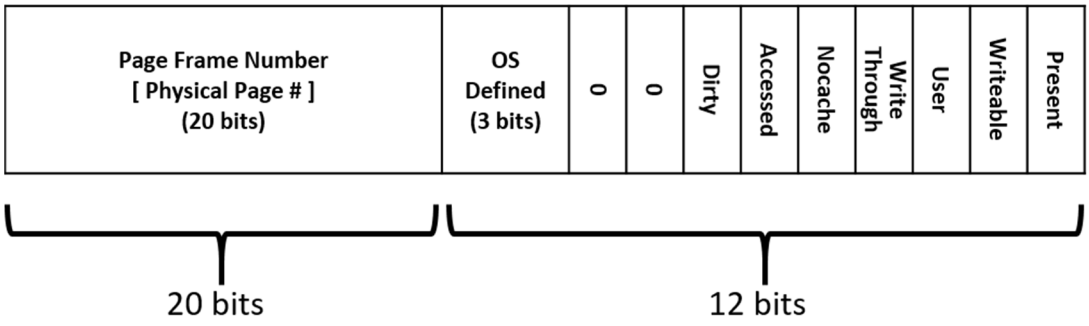
```
// Place a thread on the ready structure appropriate for the
// current active scheduling policy.
void thread_enqueue(struct thread *t) {

Line 4A:    if (t->priority < 100                                    ) {
              // Handle FIFO scheduling with priority

Line 4B:      list_push_back(&ready_list_fifo[t->priority], &t->elem)    ;
            } else {

Line 4C:      list_push_back(&ready_list_lottery, &t->elem)              ;
            }
          }
```

```
        // This is the primary scheduling function.
        // Choose (and return) the next thread to execute.
        // Make sure to remove thread from list; will be added back after run
        static struct thread* thread_schedule (void) {

Line 4D:    if (!list_empty(&ready_list_lottery)                                ) {
               int total_tickets = 0;

               // compute total_tickets in use
               for (struct list_elem* e = list_begin(&ready_list_lottery);
                    e != list_end(&ready_list_lottery);
                    e = list_next(e)) {
                  struct thread *t = list_entry(e, struct thread, elem);

Line 4E:          total_tickets += tickets_from_priority(t->priority)           ;
               }

               // Random number between 0 and total_tickets - 1, inclusive
               int chosen_ticket = random_ulong() % total_tickets;

               // Find winning thread (from list of threads)
               for (struct list_elem* e = list_begin(&ready_list_lottery);
                    e != list_end(&ready_list_lottery);
                    e = list_next(e)) {
                  struct thread *t = list_entry(e, struct thread, elem);

Line 4F:          chosen_ticket -= tickets_from_priority(t->priority)           ;

Line 4G:          if (chosen_ticket < 0                                         ) {

Line 4H:             list_remove(e)                                             ;

Line 4I:             return t                                                   ;
                  }
               }
            } else {
               for (int p = 99; p >= 0; p--) {

Line 4J:          if (!list_empty(&ready_list_fifo[p])                          ) {

Line 4K:             struct list_elem *e = list_pop_front(&ready_list_fifo[p]);

Line 4L:             return list_entry(e, struct thread, elem)                  ;
                  }
               }
               return idle_thread;
            }
        }
```

Problem 5: Virtual Memory [20 pts]

Using his newly acquired knowledge after taking 162, Yuwen is fiddling with the memory scheme in x86-64, which is a 64-bit processor. Currently, he is working on a new memory scheme with 48-bit virtual address spaces and 32-bit physical address spaces. The 48-bit virtual addresses occupy the lower 48 bits of the 64-bit address (i.e. the top 16 bits are zero). Yuwen has set up three-level page tables with equal size at each level and page table entry (PTE) size of 4 bytes. Each page is 4 KiB. Each page table starts at a page boundary.

Recall that the x86 PTE looks like the following:



Problem 5a[3pts]: What is the minimum number of pages needed to contain a complete, multi-level page table mapping *one* data page? *Express your answer as a single decimal integer AND explain your reasoning.*

Each sub-component of the page table holds 2^{12} entries of 4 bytes each = 2^{14} bytes = 4 pages (since each page is 4 KiB in size). Since this is a three-level page table, we need 3×4 = 12 total pages for a minimum (non-empty) page table.

Problem 5b[3pts]: To reduce the size of each page table, Yuwen is looking to reduce the PTE size by half. He is willing to restrict the page tables to reside in lower physical memory and restrict the access control bits to just "Present" and "Writeable". Will his memory scheme still be able to work properly? *Express your answer as either "yes" or "no" AND explain your reasoning.*

No. Even if the address of the components of the page table+access bits could fit in 16 bits (because we put page tables in lower memory), the final PTE has to map all 32-bits worth of physical memory, which requires a physical page number of 32-12 = 20 bits.

Problem 5c[14pts]: Yuwen is currently running a program and wants to analyze the physical memory accesses. Assume the next assembly instruction to execute is going to be:

`movl (%rbx), %rax`.

Some of the register contents are as follows:

CR3	0x 0000 0000 1000 0000
RIP	0x 0000 1B42 4370 A304
RAX	0x A107 C9F2 FDDD A2C0
RBX	0x 0000 B8F0 0C45 7234

*You may assume that the system is running in kernel mode. The contents of physical memory and **RAX** and **RBX** will not change during the execution of the current instruction. Further, **CR3** is the page-table base register.*

Using the memory contents displayed on the next page, fill out the following table of physical memory accesses performed by the MMU for the above instruction (assume no TLB). *Hint: the first four rows correspond to fetching the instruction, while the last four correspond to memory access from the instruction.* We have given you the first two accesses:

Access #	Address	Value
1	0x 1000 06D0	0x 1304 B00F
2	[A]: *0x 1304 B90C*	[B]: *0x 18B7 600F*
3	[C]: *0x 18B7 7C28*	[D]: *0x 2810 50FF*
4	[E]: *0x 2810 5304*	[F]: *0x 0067 8B03*
5	[G]: *0x 1000 2E3C*	[H]: *0x 1331 D007*
6	[I]: *0x 1331 D030*	[J]: *0x 1912 307F*
7	[K]: *0x 1912 415C*	[L]: *0x A516 8007*
8	[M]: *0x A516 8234*	[N]: *0x FEA1 447E*

The address column should contain the physical address **in 8 digits of hex, with 0 padding if necessary**, and prefixed with "0x". No credit will be given for nonconforming answer formats. The value column should be formatted similarly and contain the value read in from that memory address for reads and value written to that memory address for writes.

Memory access need to be written in the exact row it belongs in. For instance, writing the 4th memory access in the 2nd row will not receive any credit.

[CONTENTS OF MEMORY IN BIG-ENDIAN ORDER]

	+ 0x0	+ 0x4	+ 0x8	+ 0xC
0x 1000 0000	0x 1D64 D34E	0x BB6E 7D7B	0x 5E97 4DF2	0x 995A 665E
0x 1000 0010	0x 29B4 8FE6	0x E296 F5F7	0x 2BE5 135A	0x 0E66 14C3
...				
0x 1000 2E20	0x D097 46CB	0x 2248 775D	0x B7F9 4FF5	0x 7381 E345
0x 1000 2E30	0x C611 A9E7	0x E267 C733	0x 7F44 9FE9	0x 1331 D007
...				
0x 1000 06C0	0x 6955 3648	0x 9097 03F4	0x 0DD0 559B	0x B449 EC76
0x 1000 06D0	0x 1304 B00F	0x 2CA5 07A0	0x 1F47 06D1	0x 047C 606B
...				
0x 1304 B900	0x 8F64 496B	0x 6401 E365	0x 51D0 A006	0x 18B7 600F
0x 1304 B910	0x 2CB6 CE8D	0x 7F5D A0A8	0x E30A C1F6	0x A720 5900
...				
0x 1331 D020	0x F190 3367	0x BA83 B42B	0x 45CB E334	0x 7CBF 4D0C
0x 1331 D030	0x 1912 307F	0x D283 3085	0x 30D3 1755	0x 9FDC 644E
...				
0x 18B7 7C20	0x 8667 76D5	0x 0EC4 EA9E	0x 2810 50FF	0x 9887 B2DB
0x 18B7 7C30	0x 774D 35E8	0x 528E C666	0x 1467 70DE	0x 50D1 0AC8
...				
0x 1912 4140	0x C76F 9A9C	0x A9CD 6A7A	0x 0FCD F392	0x 766B 7380
0x 1912 4150	0x 87C3 D9B6	0x 5BC9 A256	0x 244D D109	0x A516 8007
...				
0x 2810 5300	0x 668C ED35	0x 0067 8B03	0x 58B0 8C20	0x BB06 7B72
0x 2810 5310	0x A668 1BE3	0x C27E 2836	0x C8B7 7BAA	0x ED25 5D4D
...				
0x A516 8220	0x 29F4 5DFD	0x 3169 45F2	0x 9E97 EDC1	0x A2A5 4A25
0x A516 8230	0x DC9A D3D9	0x FEA1 447E	0x 5A84 2347	0x AE5A EDCC

[This Page Intentionally Left Blank]

[Scratch Page: Do not put answers here!]

[Scratch Page: Do not put answers here!]