CS557 Midterm
Spring 2017

Please read carefully.
Please write clearly.

Definitely write your full name here:

________________SOLUTIONS________________

(Answers in <<< triple angle brackets >>>)

Exam length: 18 questions
Exam time: 2 hours
Grading (do not mark)
--------------------

Question 1: / 12
Question 2: / 8
Question 3: / 10
Question 4: / 4
Question 5: / 4
Question 6: / 5
Question 7: / 3
Question 8: / 9
Question 9: / 6
Question 10: / 6
Question 11: / 8
Question 12: / 18
Question 13: / 5
Question 14: / 4
Question 15: / 4
Question 16: / 17
Question 17: / 16
Question 18: / 14

-----------------------

Total: / 153
Question 1 (12 points)

The following (identical) pieces of pseudocode implement a hash table that uses chaining to resolve hash collisions. Given a type `spinlock_t` and the functions `spin_lock(spinlock_t*)` and `spin_unlock(spinlock_t*)`, insert data declarations and function calls as necessary to make this code safe for multithreaded execution, each in a different way.

/* Version one */
struct list_node {
    char* key;
    int value;
    list_node* next;
}

struct hashtable {
    list_node* buckets[NUMBUCKETS];
    <<< spinlock_t lock; >>>
};

void insert(hashtable* table, char* key, int value) {
    int h = hash(key) % NUMBUCKETS;
    list_node* new = new_node(key, value);
    list_node* node = table->buckets[h];
    while (node != NULL) {
        if (!strcmp(key, node->key)) {
            break;
        }
        node = node->next;
    }
    table->buckets[h] = new;
    <<< spin_unlock(&table->lock); >>>
}

int lookup(hashtable* table, char* key) {
    int tmp = -1; /* return -1 if not found */
    int h = hash(key) % NUMBUCKETS;
    list_node* node = table->buckets[h];
    while (node != NULL) {
        if (!strcmp(key, node->key)) {
            tmp = node->value;
            break;
        }
        node = node->next;
    }
    <<< spin_unlock(&table->lock); >>>
    return tmp;
}

/* Version two */
struct list_node {
    char* key;
    int value;
    list_node* next;
}

struct hashtable {
    list_node* buckets[NUMBUCKETS];
    <<< spinlock_t locks[NUMBUCKETS]; >>>
};

void insert(hashtable* table, char* key, int value) {
    int h = hash(key) % NUMBUCKETS;
    list_node* new = new_node(key, value);
    list_node* node = table->buckets[h];
    while (node != NULL) {
        if (!strcmp(key, node->key)) {
            break;
        }
        node = node->next;
    }
    table->buckets[h] = new;
    <<< spin_unlock(&table->locks[h]); >>>
}

int lookup(hashtable* table, char* key) {
    int tmp = -1; /* return -1 if not found */
    int h = hash(key) % NUMBUCKETS;
    list_node* node = table->buckets[h];
    while (node != NULL) {
        if (!strcmp(key, node->key)) {
            tmp = node->value;
            break;
        }
        node = node->next;
    }
    <<< spin_unlock(&table->locks[h]); >>>
    return tmp;
}

<<< One coarse-grained (single lock); one fine-grained (lock per bucket) >>>

How will (or won’t?) their performance characteristics differ in a program with many threads?

<<< Fine-grained will perform better with multiple threads (scales with >>>
<<< CPU/thread count) >>>

What about with a single thread?

<<< Single-threaded performance should be `equal [Trap: coarse-grained faster >>>
<<< single-threaded (not the case; # of lock acquisitions/releases the same)] >>>
Question 2 (8 points)

Imagine a malicious process is trying to do evil things to your system. For each of the Very Bad Things™ listed below that it could try to do, say which one of the following mechanisms (A, B, C, or D) would prevent it from succeeding.

(A) Instruction only allowed in privileged mode
(B) Interrupts
(C) MMU
(D) Careful checking by the kernel

- directly altering another process’s memory: <<< C >>>
- directly altering the kernel’s memory: <<< C >>>
- denying other processes CPU time by going into an infinite loop: <<< B >>>
- denying the kernel control by disabling interrupts: <<< A >>>
- gaining access to another process’s memory by altering a segment base or bound register: <<< A >>>
- gaining access to kernel memory by altering page table contents: <<< C >>>
- gaining access to kernel memory by altering the PTBR: <<< A >>>
- tricking the kernel into accessing an invalid memory address passed as a syscall parameter: <<< D >>>

Question 3 (10 points)

In operating systems in general there are three basic states in which the scheduler can put a given task: ready, running, and blocked.

For each of the following state transitions, give an example of something that would cause the transition to occur, or write "N/A" if it is not a transition that would legitimately occur.

- Ready -> Running: <<< Timer interrupt/scheduled on CPU >>>
- Running -> Ready: <<< Timer interrupt/descheduled from CPU >>>
- Ready -> Blocked: <<< N/A >>>
- Blocked -> Ready: <<< I/O operation completes, condvar signaled... >>>
- Running -> Blocked: <<< I/O operation requested, page fault, mutex wait... >>>
- Blocked -> Running: <<< N/A, or same set as Blocked->Ready >>>

On Unix-like systems there is a slightly special additional state we’ve discussed that a process can be in.

- What is the name of this state? <<< zombie >>>
- When does it arise? <<< process has exited, but parent has not waited for it >>>
- Which of the three "basic" task states does it most closely resemble as far as the scheduler is concerned? <<< blocked >>>
Question 4 (4 points)

Complete the analogies:

- In terms of virtualizing computing resources, as a process is to a whole computer, a ______ is to a CPU core.

<<< thread >>>

- As a CPU cache is to memory, a _____ is to a page table.

<<< TLB >>>

Question 5 (4 points)

Implementing virtual memory via paging offers a great deal of flexibility to the operating system. In its most basic, simplistic form, however, paging suffers two major drawbacks.

What are these disadvantages?

<<< slowed by additional access to page table on each mem access >>>
<<< page tables are too big >>>

For each drawback, what is the primary means by which we work around it?

<<< slowness: TLBs >>>
<<< page table size: multi-level page tables >>>

Question 6 (5 points)

When a page is selected to be evicted from memory to make room for another page (perhaps due to a memory allocation request or another page getting swapped in), the victim page must usually be written out to the swap device (the hard disk). There are some cases, however, when the write-out can be skipped, saving work. One such situation is when the page holds file contents (such as code from an executable) that can be read back in from the filesystem if they are needed. Describe another set of circumstances under which this optimization could be made.

<<< Both the page in memory and the location on disk at which it last >>>
<<< resided are unmodified since the last time it was paged in. >>>

What hardware-supported MMU/PTE feature would be very important in allowing the OS to do this?

<<< dirty bit >>>
Question 7 (3 points)

There are three main types of memory access that can be allowed to a given address -- read (R), write (W), and execute (X). Circle the THREE (3) entries in the following list that are MOST likely to be used in common, realistic situations (a letter being present means that type of access is permitted; a "-" means it is prohibited).

---  --X  -W-  -WX  R--  R-X  RW-  RWX

<<< RW-, R-X, R-- >>>

Question 8 (9 points)

Consider a system with a 36-bit virtual address space and a 8KB page size.

For each of the following questions, either provide an answer or write "N.E.I." to indicate that there is Not Enough Information provided to determine the answer.

- What is the maximum amount of memory a process running on this system would be able to access?  <<< 2^36 == 64GB; N.E.I. also accepted. >>>

- How many entries does the TLB have?  <<< N.E.I. >>>

- How large (in bits) is the offset portion of a virtual address?  <<< 13 >>>

- How large (in bits) is the VPN (virtual page number)?  <<< 23 >>>

- How large (in bits) is the PFN (physical frame number)?  <<< N.E.I. >>>

- With a 4-byte PTE, how much space would a linear page table require?  <<< 2^25 = 32MB >>>

- Completely ignoring all concerns about sanity or practical utility (really, go nuts here), what is the ABSOLUTE MAXIMUM number of levels a multi-level page table for this system could theoretically have?  <<< 23 >>>

- If it actually did have this many levels in its page table, how large would (or, at least, COULD) each entry in its page tables be?  <<< 4KB >>>

- OK, now drifting slightly back toward reality: why wouldn’t you want to actually have that many levels in your page tables?

<<< TLB misses very slow (lots of memory accesses) >>>
<<< Also acceptable: inefficient use of page table memory assuming one >>>
<<< page for each level of the table >>>

Question 9 (6 points)

With a round-robin CPU scheduling algorithm, a key parameter is the length of the CPU timeslice. Briefly (in a sentence or two each) explain one advantage of a shorter timeslice and one advantage of a longer timeslice. Include the "why", not just the "what".

- Shorter:  <<< better response time due to processes not having to wait >>>
<<< as long for an opportunity to run after they become ready. >>>

- Longer:  <<< better throughput due to wasting less time in context >>>
<<< switches (and also better use of caches & branch predictors). >>>
Question 10 (6 points)

Four key system calls in Unix process management are fork(), exec(), wait(), and exit(). Assuming successful operation (i.e. that no errors occur), how many times does each return after a single call?

- fork: <<< 2 >>>
- exec: <<< 0 >>>
- wait: <<< 1 >>>
- exit: <<< 0 >>>

Each time a process calls exit(), there should be a matching wait() call by its parent process.
- What happens when the parent calls wait() before the child calls exit()?

<<< Parent blocks until the child exits >>>

- What happens when the child calls exit() before the parent calls wait()?

<<< Child lingers as a zombie until the parent calls wait() >>>

================================================================================================

Question 11 (8 points)

For each of the following items that may or may not be found in a TLB entry, say whether it is Necessary, Optional, or Irrelevant (optional meaning that it might be useful but is not critical):

- PFN: <<< Necessary >>>
- ASID (a.k.a. PID): <<< Optional >>>
- PTBR: <<< Irrelevant [or Optional, if explained as equivalent to ASID] >>>
- Valid bit: <<< Necessary >>>
- Protection bits: <<< Necessary >>>
- Offset: <<< Irrelevant >>>

For each item you marked Optional, briefly (in a sentence or two each) describe how it *could* be useful but is not strictly necessary:

<<< ASID/PID: can be used to skip TLB flush on context switches >>>
<<< without it just always flush TLB on ctx switch. >>>
Below is a memory dump of the entire contents of a 10-bit physical address space (in hex, so "e" = 1110, "a" = 1010, "4" = 0100, and so forth).

The system uses a 16-byte page size, a 13-bit virtual address, and a 3-level page table with (rightmost) four bits the PFN of the page that holds the next level of the page table or (in the last level) the page that holds the byte that the virtual address being translated refers to; the remaining bits are unused.

For each of the following virtual addresses, write down, in hex, the addresses of the page table entries involved in translating it, in order (one per level of the page table), and whether it is ultimately valid or invalid. If the address is invalid, you may end up listing fewer than three PTE addresses. If the address is valid, write the (hex) value of the byte in memory at the physical address that it translates to, and what types of access (R/W/X) would be permitted to that page.

- 0x03a9 <<< 000_111_010_1001 >>>
- 0x0340: 15 3c (valid, pfn 0x15) >>>
- 0x015e: dc a8 (valid, pfn 0x1c) >>>
- 0x01c4: fc ad (valid, pfn 0x3c, PA=0x3c9, R-X) >>>
- 0x01f7: cce 111_110_1100 >>>
- 0x33e4: 9b cb (valid, pfn 0x1b) >>>
- 0x1b0c: 91 0d (valid, pfn 0x11) >>>
- 0x11c8: c1 3e (valid, pfn 0x01c, R-W-) >>>
- 0x3340: 9d ff (valid, pfn 0x1d) >>>
- 0x11d4: c2 77 (invalid) >>>
- 0x3344, 0x1d4: Invalid >>>

---

Question 12 (18 points)

Below is a memory dump of the entire contents of a 10-bit physical address space (in hex, so "e" = 1110, "a" = 1010, "4" = 0100, and so forth).

The first byte of each entry (the one at the lower address) stores in its least significant (rightmost) bits the PTE of the page that holds the next level of the page table or (in the last level) the page that the byte that the virtual address being translated refers to; the remaining bits are unused.

In the second byte of each entry (the one at the higher address), the least significant (rightmost) four bits are, in order from left to right (most to least significant): Valid, Readable, Writeable, and eXecutable. (The R/W/X bits are only used in the last level; they are ignored in the first two levels.)

For each of the following virtual addresses, write down, in hex, the addresses of the page table entries involved in translating it, in order (one per level of the page table), and whether it is ultimately valid or invalid. If the address is invalid, you may end up listing fewer than three PTE addresses. If the address is valid, write the (hex) value of the byte in memory at the physical address that it translates to, and what types of access (R/W/X) would be permitted to that page.

- 0x03a9 <<< 000_111_010_1001 >>>
- 0x0340: 15 3c (valid, pfn 0x15) >>>
- 0x015e: dc a8 (valid, pfn 0x1c) >>>
- 0x01c4: fc ad (valid, pfn 0x3c, PA=0x3c9, R-X) >>>
- 0x01f7: cce 111_110_1100 >>>
- 0x33e4: 9b cb (valid, pfn 0x1b) >>>
- 0x1b0c: 91 0d (valid, pfn 0x11) >>>
- 0x11c8: c1 3e (valid, pfn 0x01c, R-W-) >>>
- 0x3340: 9d ff (valid, pfn 0x1d) >>>
- 0x11d4: c2 77 (invalid) >>>
- 0x3344, 0x1d4: Invalid >>>
Question 13 (5 points)

For this question, assume the TLB is used only for data accesses (i.e. that instruction fetches do not use or affect the TLB).

If a system with a 32-entry TLB is able to access every element of an array of 8,192 eight-byte doubles without incurring any TLB misses (after an initial pass to fill the TLB), what is the smallest its page size could be?

<<< 2KB >>>

With a 1KB page size and a TLB using an LRU replacement policy, for each of the following TLB sizes, state how many TLB misses would be incurred by the SECOND iteration of a loop that sequentially accesses the same array described above.

- 4 entries: <<< 64 >>>

- 16 entries: <<< 64 >>>

- 128 entries: <<< 0 >>>

Question 14 (4 points)

Page size is an important parameter in a page-based virtual memory system. Briefly (in a sentence of two each) describe an advantage of making the page size smaller and an advantage of making it larger.

- Smaller: <<< Reduced space waste due to internal fragmentation >>>

- Larger: <<< Smaller page tables (also OK: reduced TLB miss rate) >>>

Question 15 (4 points)

In an MLFQ CPU scheduler, higher priority tasks always run in preference to lower priority tasks. What happens when multiple tasks are "tied" at the highest priority?

<<< Round-robin >>>

An MLFQ scheduler will reduce the priority of a task when it consumes an entire timeslice. What problem would arise without this "demotion"?

<<< Starvation by a high-prio task >>>

<<< [OR: degradation to round-robin since all tasks start at max prio] >>>
Question 16 (17 points)

Consider four tasks:
- Task A arrives at time 0 and requires 6 units of CPU time
- Task B arrives at time 3 and requires 10 units of CPU time
- Task C arrives at time 7 and requires 4 units of CPU time
- Task D arrives at time 8 and requires 2 units of CPU time

For each of the following scheduling algorithms, write the time at which each job first runs and the time at which it completes; then compute the average response time and average turnaround time across all four jobs.

- **FIFO/FCFS**:
  
  <<< Steps >>>
  <<< AAAAA|BBBBBBBBBB|CCCC|DD >>>
  <<< A: start=0, completion=6, resp=0, turnaround=6 >>>
  <<< B: start=6, completion=16, resp=3, turnaround=13 >>>
  <<< C: start=16, completion=20, resp=9, turnaround=13 >>>
  <<< D: start=20, completion=22, resp=12, turnaround=14 >>>
  <<< Avg resp: 0+3+9+12=24, 24/4 = 6 >>>
  <<< Avg turnaround: 6+13+13+14=46, 46/4 = 11.5 >>>

- **Shortest job first**:

  <<< Steps >>>
  <<< AAAAA|BBBBBBBBBB|DD|CCCC >>>
  <<< A: start=0, completion=6, resp=0, turnaround=6 >>>
  <<< B: start=6, completion=16, resp=3, turnaround=13 >>>
  <<< C: start=18, completion=22, resp=11, turnaround=15 >>>
  <<< D: start=16, completion=18, resp=8, turnaround=10 >>>
  <<< Avg resp: 0+3+11+8 = 22, 22/4 = 5.5 >>>
  <<< Avg turnaround: 6+13+15+10 = 44, 44/4 = 11 >>>

- **Shortest time-to-completion first**:

  <<< Steps >>>
  <<< AAAAA|B|C|DD|CCC|BBBBBBBBB >>>
  <<< A: start=0, completion=6, resp=0, turnaround=6 >>>
  <<< B: start=6, completion=22, resp=3, turnaround=19 >>>
  <<< C: start=7, completion=13, resp=0, turnaround=6 >>>
  <<< D: start=8, completion=10, resp=0, turnaround=2 >>>
  <<< Avg resp: 0+3+0+0=3, 3/4 = 0.75 >>>
  <<< Avg turnaround=2: 6+19+6+2 = 33, 33/4 = 8.25 >>>

What key feature differentiates STCF from SJF?

<<< Preemption >>>

Why are SJF and STCF not really implementable in practice?

<<< CPU time required for each job not known in advance >>>
Question 17 (16 points)

Different CPUs offer a variety of different hardware atomicity primitives; here are pseudocode
functions describing the behavior of some possibilities (all execute as atomic operations):

/* load-linked */                            | /* fetch-and-add, a.k.a. xadd */
unsigned LL(unsigned* addr)                  | unsigned XADD(unsigned* addr, unsigned val)
{                                            | {
    /* HW remembers addr as "special" */ |    unsigned tmp = *addr;
    return *addr;                        |    *addr += val;
}                                            |    return tmp;

/* store-conditional */                      |
int SC(unsigned* addr, unsigned val)         |
{                                            |
    if (*addr unmodified since LL) {     |
        *addr = val;                 |
        return 1;                    |
    } else {                             |
        return 0;                    |
    }
|

Here are a pair of data structures that might be used to implement spinlocks:

struct spinlock1 { unsigned x; }   | struct spinlock2 { unsigned x, y; }
}                                     |

Here are some functions to release spinlocks represented by the structs above:

void unlock1(struct spinlock1* lock)   | void unlock2(struct spinlock2* lock)
{                                       |
    lock->x = 0;                       |    XADD(&lock->y, 1);
}                                       |

Here are some possible implementations (not necessarily correct) of functions to acquire a
spinlock using the primitives and data structures above (assume all data starts out zeroed). A, B, and C would be used with 'unlock1' above; D, E, and F would be used with 'unlock2'.

void A(struct spinlock1* lock)                 | void D(struct spinlock2* lock)
{                                              |
    do {                                         |
        while (LL(&lock->x))                   |
            /* spin */;
    } while (!SC(&lock->x, 1));            |
}                                              |

void B(struct spinlock1* lock)                 | void E(struct spinlock2* lock)
{                                              |
    while (!LL(&lock->x)) {                |
        if (SC(&lock->x, 1)) {         |
            return;                |
        }
    }
}                                      |

void C(struct spinlock1* lock)                 | void F(struct spinlock2* lock)
{                                              |
    while (1) {                             |
        if (!LL(&lock->x)) {            |
            if (SC(&lock->x, 1)) { |
                return;        |
            }
        }
    }
}                                             |

[Tragically, this question did not fit on one page, so it is continued on the next one. You do
not need to write anything on this page. (You may if you want to, however.)]
Question 17 (16 points) (cont.)

Write the names (a single letter each) of each of the six lock-acquisition functions on the previous page that works correctly:

<<< A, C, F >>>

What advantage(s), if any, might a spinlock implemented using ‘spinlock1’ and LL/SC have over one using ‘spinlock2’ and XADD?

<<< Full points: Smaller (reduced memory consumption) >>>
<<< Half points: None >>>

What advantage(s), if any, might a spinlock implemented using ‘spinlock2’ and XADD have over one using ‘spinlock1’ and LL/SC?

<<< Fairness/FIFO ordering, increased efficiency >>>
Here is some pseudocode for a semaphore-based producer-consumer queue:

```c
int buffer[PC_QUEUE_SIZE];
int head = 0, tail = 0;
sem_t empty; /* initialized to PC_QUEUE_SIZE */
sem_t full; /* initialized to zero */
sem_t mutex; /* initialized to one */

void insert(int value)
{
    buffer[head] = value;
    head = (head + 1) % PC_QUEUE_SIZE;
}

int remove(void)
{
    int tmp = buffer[tail];
    tail = (tail + 1) % PC_QUEUE_SIZE;
    return tmp;
}

void producer(void)
{
    while (1) {
        wait(&empty);
        wait(&mutex);
        insert(random());
        post(&mutex);
        post(&full);
    }
}

void consumer(void)
{
    int tmp;
    while (1) {
        wait(&full);
        wait(&mutex);
        tmp = remove();
        post(&mutex);
        post(&empty);
        printf("%d\n", tmp);
    }
}
```

Consider a slightly altered version of a semaphore called a bsem whose initialization function looks like this:

```c
void bsem_init(bsem_t* bs, int initval, int max);
```

Like a normal semaphore, its internal counter will start out at 'initval' and be incremented and decremented by post() and wait() calls, respectively. Unlike a normal semaphore, however, with a bsem both wait() AND post() are potentially blocking operations: a post() call that increments the internal counter beyond 'max' blocks until there is a matching wait() call (much like how a wait() that decrements it below zero blocks until there is a matching post()).

Could you use this to eliminate the need for the separate 'full' and 'empty' semaphores in the pseudocode above and replace them both with a single bsem? If so, write a few sentences describing how it would work (make sure you explain what the bsem's counter value would represent, and what the 'initval' and 'max' arguments passed to bsem_init() would be). If not, explain why it wouldn't work.

<<< Partial-credit: Yes. EITHER: bsem_init(bs, 0, PC_QUEUE_SIZE) with bsem counter value representing full slots, producer post()s and consumer wait()s; OR bsem_init(bs, PC_QUEUE_SIZE, PC_QUEUE_SIZE) with bsem counter value representing empty slots, producer wait()s and consumer post()s. >>>
<<< Full-credit: No. Do you do the wait()/post() before or after accessing the queue? Neither works. If after, producers could overflow the queue, or consumers could underflow it. If before, producer and consumer could race with each other trying to insert() or remove() before the other has done the matching remove() or insert(). >>>
This page left intentionally (mostly) blank.  [Hint: scratch space!]

[Extra hint: you may carefully tear it off for reduced page-turning.]