CS 537: Introduction to Operating Systems
Fall 2019: Midterm Exam #2
November 6, 2019

This exam is closed book, closed notes.

All cell phones must be turned off and put away.

No calculators may be used.

You have two hours to complete this exam.

Write all of your answers on the accu-scan form with a #2 pencil.

These exam questions must be returned at the end of the exam, but we will not grade anything in this booklet.

Unless stated (or implied) otherwise, you should make the following assumptions:

- All memory is byte addressable
- The terminology lg means $\log_2$
- 2^{10} bytes = 1KB
- 2^{20} bytes = 1MB
- Page table entries require 4 bytes
- Data is allocated with optimal alignment, starting at the beginning of a page
- Assume leading zeros can be removed from numbers (e.g., 0x06 == 0x6).

There are 90 questions on this exam.

Good luck!

Virtualization Review [1 point each]
Designate if each statement is True (a) or False (b).

1) On a timer interrupt, the OS is responsible for transitioning from user to kernel mode.

False; on an interrupt, the hardware performs the transition from user to kernel mode

2) Increasing the time-slice of an RR scheduler is likely to decrease the overhead imposed by scheduling

True; fewer context switches.

3) Increasing the time-slice of an RR scheduler makes the scheduler behave more like FCFS.

True; in the extreme case, time-slice >= job length, then identical to FCFS

4) An MLFQ scheduler assumes the presence of an oracle that knows the length of a job's CPU burst in advance.

False; the schedule dynamically adjusts the time slice based on past behavior of that job

5) The fork() system call creates a child process whose execution begins at the entry point main().

False; child starts execution as if returning from fork().

6) A process can close() the stdout file descriptor.

True; this is how you can implement stdout redirection to a file

7) With dynamic relocation, the OS determines where the address space of a process is allocated in virtual memory.

False; OS picks location in physical memory; compiler generates all addresses in virtual memory

8) With dynamic relocation, the OS can move an address space after it has been placed in physical memory.

True; just copy data in physical memory and then change base register in MMU.

9) A Gantt chart illustrates how virtual pages are mapped to physical pages

False; Gantt charts show scheduling of jobs over time.

10) With pure segmentation (with no paging or TLBs), each segment of each process must be allocated contiguously in physical memory.

True; each segment must be contiguous with its own base and bounds register

11) With pure segmentation (with no paging or TLBs), fetching an instruction will involve exactly two memory references.

False; to fetch an instruction, there is one address translation lookup;with pure segmentation, the start of the segment is kept in a register; there are no page tables, so no extra memory references

12) Threads running within the same process are likely to share the same value for the base register, but not the bounds register, in the MMU.

False; share the same address space, so share same base and bounds

13) Given a constant number of bits in a virtual address, the size of a linear page table increases with more (virtual) pages.

True. More virtual pages → More PTEs → larger page table

14) Given a fixed number of virtual pages, the size of a linear page table decreases with a smaller address space.

False; fixed number of pages → Fixed number of PTEs → Fixed size of page table

15) Given a two-level page table, there may be two TLB misses when performing a virtual to physical address translation.

False; the TLB maps directly from VPN to PPN (nothing with inner/outer pages)

16) If the TLB does not support ASIDs, all page table entries (PTEs) are marked invalid on a context switch

False; PTEs aren't marked invalid, it is TLB entries that are marked invalid.

17) With a linear page table, all virtual pages within an address space must be allocated.

False; while the PTEs have to be allocated with a linear page table, the actual pages do not.

18) Given a 2-level page table (and no TLB), exactly 3 memory accesses are needed to fetch each instruction.

True; for the fetch, one to lookup page mapping in page directory, one to lookup in inner page table, one to fetch actual instruction

19) LRU with N+1 physical pages will contain (cache) the same virtual pages as LRU with N pages, plus the contents of one more virtual page.

True, adding 1 more frame to LRU means old content is cached plus the one page referenced longer ago…

20) Demand paging loads the address space of a process into physical memory when that process is started

False; demand paging waits until the process accesses a particular page to load it.

21) The clock algorithm must be able to determine if a page has been accessed in some particular time interval

True; the clock hand looks for a page that has not been accessed since the last time it was examined

Concurrency [2 points each]
Designate if the statement is **True (a)** or **False (b).**

22) The clock frequency of CPUs has been increasing exponentially each year since the VAX-11.

False; the increases have not been so dramatic somewhat recently

23) Threads that are part of the same process share the same page table.

True; they share the same address space, so same page table

24) Threads that are part of the same process share the same registers.

False; each thread is given its own virtualized set of registers (appear different across threads)

25) Context-switching between threads of the same process requires flushing the TLB

False; since they are part of the same address space, they have the same page translations

26) A problem with user-level thread implementations is that if one thread in the application process blocks, then all the threads in that application process are blocked as well.

True; since the threads appear to the OS scheduler all be the same (part of the same process), if one thread blocks, the scheduler considers the entire process blocked.

27) Locks prevent the OS scheduler from performing a context switch during a critical section.

False; context switches can still occur.

28) The **volatile** key word before a variable directs the compiler to protect that variable with locks.

False; volatile simply directly the compiler to keep the variable in memory (and move it back and forth to registers as needed to modify)

29) Peterson's algorithm is able to provide mutual exclusion using simple load and store instructions for two threads running on a modern multiprocessor.

False; modern multiprocessors don't provide the sequential consistency assumed by Peterson's algorithm

30) A context switch can occur in the middle of the atomic hardware instruction XCHG.

False; the hardware instruction is atomic – nothing else can happen during it.

31) Atomic hardware instructions can be used as building blocks for providing mutual exclusion to user applications on multiprocessor systems.

True; e.g., can build a lock out of test-and-set.

32) It is possible for a lock implementation to provide fairness across threads (i.e., threads receive the lock in the order they requested the lock) with spin-waiting.

True; can implement a ticket lock with spin-waiting

33) To minimize wasted CPU cycles on a multiprocessor system, a lock implementation should block instead of spin if the lock will held by another process for longer than the time required for a context-switch.

True; less time will be wasted with the context switch than would be spent spin-waiting.

34) A ticket lock must atomically increment the ticket number that a thread is given when it tries to acquire the lock.

True; the algorithm assumes two competing threads can't get the same ticket number

35) Two-phase waiting assumes that an oracle can predict the amount of time that the lock will be held by another process.

False; two-phase waiting bounds the worst-case wasted time given no knowledge of how long the lock will be held.

36) When a thread calls cond_wait() on a condition variable, cond_wait() returns immediately if a cooperating thread previously called cond_signal() on that condition variable.

False; condition variables do not track any state or knowledge that cond_signal was called in the past

37) The call cond_wait() must release a corresponding mutex.

True.

38) With producer/consumer relationships and a circular shared buffer containing N elements, producing threads must wait when there is exactly 1 empty element in the buffer.

False; producing threads must wait when there are no empty buffers.

39) The performance of a multi-threaded application that uses broadcasting to a condition variable is likely to decrease as the number of waiting threads increases.

True; more threads will wake-up even though only one is likely to be able to make progress; takes time for all to be scheduled, recheck condition, and then call wait() again.

40) Condition variables can provide both ordering and mutual exclusion.

False; just ordering

41) Semaphores can provide both ordering and mutual exclusion.

True

42) Semaphores can be built on top of condition variables and locks.

True;

43) As the amount of code a mutex protects increases, the amount of concurrency in the application increases.

False; more code that only one thread can execute at a time → less concurrency

44) To implement a thread_join() operation with a semaphore, the semaphore is initialized to the value of 0 and the thread_join() code calls sem_wait().

True

45) The safety property for dining philosophers states that no two adjacent philosophers are eating simultaneously.

True

46) With a reader/writer lock, multiple readers can hold the lock simultaneously with a single writer.

False; multiple readers can hold the lock simultaneously OR one single writer, but not both groups together

47) With a wait-free algorithm, a thread that cannot acquire a lock must release any other locks that the thread holds.

False; wait-free means not using locks (instead use atomic hardware instructions); releasing locks as described means we are ensuring no hold-and-wait.

48) Deadlock can be avoided if locks are always acquired in a well-defined, strict linear order.

True; no circular ordering $\rightarrow$ no deadlock

Processes and Fork [7 total points, 1 point each]

Assume the following code is compiled and run on a modern Linux machine. Assume the code compiles correctly and fork() never fails.

```
volatile int a = 0;
main() {
   int b = 0;                // 1
   a++;                      // 2
   int rc = fork();          // 3
   b++;                      // 4
   fork();                   // 5
   if (rc == 0) {            // 6
         fork();             // 7
         a++;                // 8
   }
   b++;                      // 9
}
```

49) How many processes will execute **line 2** of this program?
   a) 1
   b) 2
   c) 4
   d) 8
   e) None of the above

50) How many processes will execute **line 4** of this program?
   a) 1
   b) 2
   c) 4
   d) 8
   e) None of the above

51) How many processes will execute **line 6** of this program?
   a) 1
   b) 2
   c) 4
   d) 8
   e) None of the above

52) How many processes will execute **line 8** of this program?
   a) 2
   b) 4
   c) 6
   d) 8
   e) None of the above

Of the 4 processes that execute line 6, only half of them (2 processes) call fork at line 7, leading to 4 processes at line 8

53) What will be the value of variable **b** after **line 4** has executed?
   a) Known, but will be different for different processes
   b) Unknown; may have unexpected values due to race conditions
   c) 1
   d) 2
   e) None of the above

   Each process has their own copy of b (and all other variables whether allocated on stack or heap). B is incremented 1 time to 1.

54) What will be the value of variable **b** after **line 9** has executed?
   a) Known, but will be different for different processes
   b) Unknown; may have unexpected values due to race conditions
   c) 2
   d) 4
   e) None of the above

Each process has their own copy of b (and all other variables whether allocated on stack or heap). B is incremented 2 times to 2.

55) What will be the value of variable **a** after **line 9** has executed?

a) Known, but will be different for different processes
b) Unknown; may have unexpected values due to race conditions
c) 2
d) 4
e) None of the above

Each process has its own copy of a, and there aren't any race conditions where one process could overwrite another. However, some processes increment a 1 time and some 2 times

Threads [9 total points, 3 points each]

Assume the following code is compiled and run on a modern Linux machine. Assume the code compiles correctly and none of the system calls fail. Look at the code carefully; you probably haven't seen this exact version before…

```
volatile int balance = 0;
pthread_mutex_t add_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t sub_mutex = PTHREAD_MUTEX_INITIALIZER;
void *add(void *arg) {
   for (int i = 0; i < 200; i++) {
         pthread_mutex_lock(&add_mutex);
         balance++;
         pthread_mutex_unlock(&add_mutex);
   }
   printf("Balance is %d\n", balance);
   return NULL;
}
void *sub(void *arg) {
   for (int i = 0; i < 200; i++) {
         pthread_mutex_lock(&sub_mutex);
         balance--;
         pthread_mutex_unlock(&sub_mutex);
   }
   printf("Balance is %d\n", balance);
   return NULL;
}
int main(int argc, char *argv[]) {
   pthread_t p1, p2;

   pthread_create(&p1, NULL, add, "A");
   pthread_create(&p2, NULL, sub, "B");
   pthread_join(p1, NULL);
   pthread_join(p2, NULL);
   printf("Final Balance is %d\n", balance);
}
```

56) When thread p1 prints "Balance is %d\n", what will p1 say is the value of `balance`?

- a) Due to race conditions or other timing differences, "balance" may have different values at this point on different runs of the program.
- b) -200
- c) 0
- d) 200
- e) None of the above

The problem is that different locks are used to protect balance in the two different routines, thus one thread can be trying to increment balance while another tries to decrement it.

57) When thread p2 prints "Balance is %d\n", what will p2 say is the value of `balance`?

- a) Due to race conditions or other timing differences, "balance" may have different values at this point on different runs of the program.
- b) -200
- c) 0
- d) 200
- e) None of the above

58) When the main parent thread prints "Final Balance is %d\n", what will the parent thread say is the value of `balance`?

- a) Due to race conditions or other timing differences, "balance" may have different values at this point on different runs of the program.
- b) -200
- c) 0
- d) 200
- e) None of the above

Thread Join with Condition Variables [15 total points]
Your project partner wrote code to implement thread_join() and thread_exit() with locks and condition variables, but it is a little bit different than what you saw in lecture. Specifically, lines c2 and c3 have been swapped. Assume the code compiles correctly. It is your job to complete the initialization and to then determine whether or not this implementation is correct.

```
void thread_join() {
        Mutex_lock(&m);           // p1
        while (done == 0)         // p2
            Cond_wait(&c, &m);    // p3
        Mutex_unlock(&m);         // p4
}
void thread_exit() {
        Mutex_lock(&m);           // c1
        Cond_signal(&c);          // c2
        done = 1;                 // c3
        Mutex_unlock(&m);         // c4
}
```

Assume `thread_create()` which the parent thread invokes to create and start the child thread has the following signature:

```
int thread_create(pthread_t *thread,
                  void *(*start_routine)(void *), void *arg)
```

59) Ignoring any new problems your partner may have introduced, to what value should the **done** variable be initialized in order to join the parent and child? [2 points]
a) -1
b) 0
c) 1
d) 2
e) None of the above

Initialize so thread_join will call cond_wait() unless thread_exit() has run first.

60) Ignoring any new problems your partner may have introduced, to what value should the **condition variable, c**, be initialized? [2 points]
a) -1
b) 0
c) 1
d) 2
e) None of the above

Condition variables do not contain state; doesn't make sense to initialize to an integer value

Again, ignoring any problems your partner may have introduced, which of the following thread library routines could perform this initialization of **done** and the condition variable **c** (without causing any race conditions)? Assume that these variables are not initialized while a mutex lock is held. Note: **at least one** of these options is a correct possibility.

Answer **(a) yes, a correct possibility**, or **(b) no, could cause a race condition**

61) `thread_create()` in the parent thread before the `start_routine` of the child thread is called [2 points]

Yes. Child thread has not started yet.

62) `thread_create()` in the parent thread after the `start_routine` of the child thread is called [1 point]

No. child thread has started, so child could call thread_exit() before parent initializes done.

63) In the child thread before beginning its designated `start_routine [1 point]`

No. The parent thread could call thread_join() before child initializes done.

64) `thread_join()`: As the first line of thread_join [1 point]

No. Child thread could call thread_exit before parent calls thread_join and performs initialization.

65) `thread_exit()`: As the first line of thread_exit [1 point]

No. Parent thread could call thread_join before initialization in thread_Exit.

66) Now, assuming everything has been initialized correctly, does your project partner's version of `thread_join()` and `thread_exit()` work correctly? [5 points]

a) **Only** if the parent calls `thread_join()` and `cond_wait()` before the child calls thread_exit()

b) **Only** if the child completes `thread_exit()` before the parent calls `thread_join()`

c) **Only** if there is **not** a context switch between when the child calls `cond_signal()` and sets `done = 1`

d) Works correctly in all cases

e) None of the above

Swapping lines c2 and c3 has no impact on correctly because they are performed while the mutex is held; therefore, the ordering of those two statements is not visible to any other threads that also acquire that mutex.

Reader/Writer Locks with Semaphores [20 points total]
In this question, you will explore your understanding of the reader/writer lock code with semaphores. The code below is identical to that shown in class. No changes were made and no bugs were introduced (ignore any incorrect/inconsistent capitalization). Assume the code compiles correctly.

```
Acquire_readlock() {
   Sem_wait(&mutex);           // AR1 (line 1 of Acquire_readlock)
   If (ActiveWriters +         // AR2
        WaitingWriters==0) {   // AR3
         sem_post(OKToRead);   // AR4
         ActiveReaders++;      // AR5
   } else WaitingReaders++;    // AR6
   Sem_post(&mutex);           // AR7
   Sem_wait(OKToRead);         // AR8
}
Release_readlock() {
   Sem_wait(&mutex);           // RR1 (line 1 of Release_readlock
   ActiveReaders--;            // RR2
   If (ActiveReaders==0 &&     // RR3
     WaitingWriters > 0) {     // RR4
         ActiveWriters++;      // RR5
         WaitingWriters--;     // RR6
         Sem_post(OKToWrite);  // RR7
   }
   Sem_post(&mutex);           // RR8
}
Acquire_writelock() {
   Sem_wait(&mutex);           // AW1 (line 1 of Acquire_writelock)
   If (ActiveWriters + ActiveReaders + WaitingWriters==0) { // AW2
       ActiveWriters++;        // AW3
       sem_post(OKToWrite);    // AW4
   } else WaitingWriters++;    // AW5
   Sem_post(&mutex);           // AW6
   Sem_wait(OKToWrite);        // AW7
}
Release_writelock() {
   Sem_wait(&mutex);           // RW1
   ActiveWriters--;            // RW2
   If (WaitingWriters > 0) {   // RW3
      ActiveWriters++;         // RW4
      WaitingWriters--;        // RW5
      Sem_post(OKToWrite);     // RW6
   } else while(WaitingReaders>0) { // RW7
      ActiveReaders++;         // RW8
      WaitingReaders--;        // RW9
      sem_post(OKToRead);      // RW10
   }
   Sem_post(&mutex);           // RW11
```

67) Does this implementation of reader/writer locks give priority to readers or to writers? That is, if there is both a reader and a writer waiting to acquire this lock, which one will be given the lock next? [2 points]

a) The reader
b) The writer
c) The thread that called acquire_readlock() or acquire_writelock() first
d) Strictly alternates between readers and writers
e) Cannot determine because cannot make assumptions about behavior of scheduler

When release_lock() is called (for either read or write lock), if the number of waiting writers> 0, then sem_post(OKToWrite) is called, which lets a writer continue.

68) How should the semaphore mutex() be initialized? [2 points]

a) 0
b) 1
c) To the number of reader threads
d) To the number of writer threads
e) None of the above

1 thread should be able to acquire a mutex lock

69) How should the semaphore OKToRead be initialized? [2 points]

a) 0
b) 1
c) To the number of reader threads
d) To the number of writer threads
e) None of the above

A thread needs to call sem_post() on OKToRead to let a reader thread continue past sem_wait(OKToRead).

70) How should the semaphore OKToWrite be initialized? [2 points]

a) 0
b) 1
c) To the number of reader threads
d) To the number of writer threads
e) None of the above

A thread needs to call sem_post() on OKToWrite to let a writer thread continue past sem_wait(OKToWrite).

Assume the following calls are made by threads in the system and all three threads execute as far along as they can until they execute a statement that causes them to block (or wait). Assume that after a thread returns from one of these four functions, the thread executes other user code that does not involve any synchronization or blocking (i.e., some code beyond AR8, RR8, AW7, and RW11).

```
Reader Thread R0:     Acquire_readlock();
Writer Thread W0:     Acquire_writelock();
Reader Thread R1:     Acquire_readlock();
```

71) Where will reader thread R0 be in the code? [1 point]

a) AR1
b) AR4
c) AR8
d) Beyond AR8

e) None of the above (including non-deterministic locations)

RO acquires the reader lock and continues past the shown code

72) Where will thread W0 be in the code? [1 point]

a) AW1
b) AW4
c) AW7
d) Beyond AW7
e) None of the above (including non-deterministic locations)

W0 did not call sem_post(OKTOWrite) so it will be stuck on Sem_wait(OKToWrite);

73) Where will thread R1 be in the code? [2 points]

a) AR1
b) AR4
c) AR8
d) Beyond AR8
e) None of the above (including non-deterministic locations)

R0 did not call sem_post(OKTORead) so it will be stuck on Sem_wait(OKToRead);

f)

Continuing the same execution stream as above, now assume Thread R0 calls release_readlock() and the three threads again execute as far along as they can until they run into a statement that causes them to block (or wait).

74) Where will thread R0 be in the code? [1 point]

a) RR1
b) RR7
c) RR8
d) Beyond RR8
e) None of the above (including non-deterministic locations)

R0 will finish with release_readlock and continues past the shown code. As part of release_readlock() it will call sem_post(OKToWrite);

75) Where will thread W0 be in the code? [1 point]

a) AW1
b) AW4
c) AW7
d) Beyond AW7
e) None of the above (including non-deterministic locations)

Since R0 called sem_post(OKToWrite) W0 will finish acquire_writelock and continue past the shown code

76) Where will thread R1 be in the code? [2 points]

a) AR1
b) AR4
c) AR8
d) Beyond AR8
e) None of the above (including non-deterministic locations)

Nothing has changed for R1, it is still waiting to acquire the reader lock

77) Continuing the same execution stream, assume another read thread R2 begins and calls acquire_readlock() and executes as far along as it can before it must block. Where will thread R2 be in the code? [2 points]

a) AR1
b) AR7

c) AR8
d) Beyond AR8
e) None of the above (including non-deterministic locations)

R2 will also not call sem_post(OKToRead) and so will also be stuck at Sem_Wait(OKToRead)

78) Continuing the same execution stream, when writer thread W0 eventually calls release_writelock(), what will happen next before threads must block? [2 points]

a) Only Thread R1 will acquire the readlock, while R2 continues to wait for the readlock to be released
b) Thread R1 **or** R2 will acquire the readlock, but it is unknown which (the other waits for the readlock to be released)
c) First, Thread R1 will acquire the readlock, and then Thread R2 will acquire the readlock, while R1 still also has the readlock
d) Both Thread R1 and Thread R2 will acquire and concurrently hold the readlock, but the order in which each thread will return from acquire_readlock() is unknown
e) None of the above

When W0 calls release_writelock() it will call sem_post(OKToRead) two times. Therefore, both R1 and R2 will be able to continue with the readlock. It is unknown if R1 or R2 will be awoken from Sem_wait() first or which will return from acquire_readlock() first.

Wait-Free Algorithms [8 total points, 2 points each]

Your project partner has written the following correct thread-safe implementation of `add()` using traditional locks for mutual exclusion:

```
void add (int *val, int amt) {
   mutex_lock(&m);
   *val += amt;
   mutex_unlock(&m);
}
```

You decide you would like to replace the lock with calls to the atomic hardware instruction `CmpAndSwap(int *addr, int expect, int new)`, which returns 0 on failure and 1 on success. The exact behavior of atomic `CmpAndSwap()` is defined as in class.

You know that `add()` modified to use `CmpAndSwap()` looks something like the following:

```
void add (int *val, int amt) {
   do {
         int old = *val;
   } while (CmpAndSwap(<Q1>, <Q2>, <Q3>) == <Q4>);
}
```

What are the correct replacements for the symbols < Q1, Q2, Q3, Q4> in the code above?

79) <Q1> should be replaced with
- [✓ a) val]
- b) *val
- c) old + amt
- d) old
- e) None of the above

80) <Q2> should be replaced with
- a) val
- b) *val
- c) old +amt
- [✓ d) old]
- e) None of the above

81) <Q3> should be replaced with
- a) val
- b) *val
- [✓ c) old+amt]
- d) old
- e) None of the above

82) <Q4> should be replaced with
- [✓ a) 0]
- b) 1
- c) val
- d) *val
- e) None of the above

Scheduling without locks given assembly code[16 total points, 2 points each]
For the next questions, assume that two threads are running the following code on a uniprocessor (this is the same `looping-race-nolock.s` code from homework simulations).

```
# assumes %bx has loop count in it
.main
.top
mov 2000, %ax  # get the value at the address
add $1, %ax    # increment it
mov %ax, 2000  # store it back

# see if we're still looping
sub  $1, %bx
test $0, %bx
jgt .top

halt
```

This code is incrementing a variable (e.g., a shared balance) many times in a loop. Assume that the %bx register begins with the value 3, so that each thread performs the loop 3 times. Assume the code is loaded at address 1000 and that **memory address 2000 originally contains the value 0**. Assume that the scheduler runs the two threads producing the following order of instructions (the first column shows the address of the executed instruction).

For each of the questions designated below, determine the contents of the %ax register or the memory address 2000 AFTER that assembly instruction executes.

a) 1
b) 2
c) 3
d) 4
e) None of the above

```
Thread 0                 Thread 1
1000 mov 2000, %ax
1001 add $1, %ax
1002 mov %ax, 2000
------ Interrupt ------  ------ Interrupt ------
                         1000 mov 2000, %ax       83) Contents of register %ax? 1
                         1001 add $1, %ax
                         1002 mov %ax, 2000       84) Contents of addr 2000? 2
                         1003 sub  $1, %bx
                         1004 test $0, %bx
                         1005 jgt .top
                         1000 mov 2000, %ax
------ Interrupt ------  ------ Interrupt ------
1003 sub  $1, %bx
1004 test $0, %bx
1005 jgt .top
------ Interrupt ------  ------ Interrupt ------
                         1001 add $1, %ax         85) Contents of register %ax? 3
------ Interrupt ------  ------ Interrupt ------
1000 mov 2000, %ax                                86) Contents of register %ax? 2
1001 add $1, %ax
```

```
1002 mov %ax, 2000                                 87) Contents of addr 2000? 3
1003 sub  $1, %bx
1004 test $0, %bx
1005 jgt .top
1000 mov 2000, %ax
------ Interrupt ------  ------ Interrupt ------
                        1002 mov %ax, 2000         88) Contents of addr 2000? 3
                        1003 sub  $1, %bx
                        1004 test $0, %bx
                        1005 jgt .top
------ Interrupt ------  ------ Interrupt ------
1001 add $1, %ax
1002 mov %ax, 2000                                 89) Contents of addr 2000? 4
1003 sub  $1, %bx
1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
                        1000 mov 2000, %ax
                        1001 add $1, %ax
                        1002 mov %ax, 2000         90) Contents of addr 2000? 5
------ Interrupt ------  ------ Interrupt ------
1005 jgt .top
1006 halt
----- Halt;Switch -----  ----- Halt;Switch -----
                        1003 sub  $1, %bx
                        1004 test $0, %bx
                        1005 jgt .top
                        1006 halt
```

Congratulations on finishing!

See you in lecture tomorrow to talk more about file systems!