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:
- The OS manages a single uniprocessor
- All memory is byte addressable
- The terminology \( \log_2 \) 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).

This exam has multiple versions. To make sure you are graded with the correct answer key, you must identify this exam version with a Special Code in Column A on your accu-tron sheet. Be sure to fill in the corresponding bubble as well.

Your special code is

There are **107 questions on this exam.**

Good luck!
Part 1: Straight-forward True/False about Virtualization [2 points each]
Designate if each statement is True (a) or False (b).

1) Two processes reading from the same physical address will access the same contents.
2) Stacks are used for procedure call frames, which include local variables and parameters.
3) A SJF scheduler may preempt a previously running longer job.
4) If all jobs have identical run lengths, a RR scheduler (with a time-slice much shorter than the jobs’ run lengths) provides better average turnaround time than FIFO.
5) The longer the time slice, the more a RR scheduler gives similar results to a FIFO scheduler.
6) The OS provides the illusion to each thread that it has its own address space.
7) The OS may manipulate the contents of an MMU.
8) With a single-level page table (and no other support), fetching and executing an instruction that performs an add of a constant value to a register will involve exactly two memory references.
9) Given a constant number of bits in a virtual address, the size of a linear page table decreases with larger pages.
10) If a physical address is 32 bits and each page is 4KB, the top 18 bits exactly designate the physical page number.
11) A multi-level page table typically reduces the amount of memory needed to store page tables, compared to a linear page table.
12) Paging approaches suffer from internal fragmentation, which decreases as the size of a page decreases.
13) The size of a virtual page is identical to the size of a physical page.
14) TLB reach is defined as the number of TLB entries multiplied by the size of a page.
15) If the valid bit is clear (equals 0) in a PTE needed for a memory access, the desired page will be swapped in from the backing store.
16) TLBs are more beneficial with multi-level page tables than with linear (single-level) page tables.
17) When the dirty bit is clear (equals 0) in a PTE needed for a memory access, an identical copy of the desired page resides in the backing store.
18) LRU with N+1 pages of memory always performs better than LRU with N pages of memory.
19) FIFO with N+1 pages of memory always performs better than FIFO with N pages of memory.
Part 2: Process States [1 point each]
Assume you have a system with three processes (X, Y, and Z) and a single CPU. Process X has the highest priority, process Z has the lowest, and Y is in the middle. Assume a priority-based scheduler (i.e., the scheduler runs the highest priority job, performing preemption as necessary). Processes can be in one of five states: RUNNING, READY, BLOCKED, not yet created, or terminated. Given the following cumulative timeline of process behavior, indicate the state the specified process is in AFTER that step, and all preceding steps, have taken place. Assume the scheduler has reacted to the specified workload change.

For all questions in this Part, use the following options for each answer:
- a. RUNNING
- b. READY
- c. BLOCKED
- d. Process has not been created yet
- e. Not enough information to determine OR None of the above

Step 1: Process X is loaded into memory and begins; it is the only user-level process in the system.
20) Process X is in which state?

Step 2: Process X calls fork() and creates Process Y.
21) Process X is in which state?
22) Process Y is in which state?

Step 3: The running process issues an I/O request to the disk.
23) Process X is in which state?
24) Process Y is in which state?

Step 4: The running process calls fork() and creates process Z.
25) Process X is in which state?
26) Process Y is in which state?
27) Process Z is in which state?

Step 5: The previously issued I/O request completes.
28) Process X is in which state?
29) Process Y is in which state?
30) Process Z is in which state?

Step 6: The running process completes.
31) Process X is in which state?
32) Process Y is in which state?
33) Process Z is in which state?
Part 3: Straight-forward True/False about Concurrency [3 points each]
Designate if the statement is True (a) or False (b).

34) The clock frequency of CPUs has been increasing exponentially each year since 2005.
35) Threads that are part of the same process share the same stack.
36) Threads that are part of the same process can access the same TLB entries.
37) With kernel-level threads, multiple threads from the same process can be scheduled on multiple CPUs simultaneously.
38) Locks prevent the OS scheduler from performing a context switch during a critical section.
39) Peterson’s algorithm uses the atomic fetch-and-add instruction to provide mutual exclusion for two threads.
40) A lock that performs spin-waiting can provide fairness across threads (i.e., threads receive the lock in the order they requested the lock).
41) A lock implementation should block instead of spin if it will always be used only on a uniprocessor.
42) On a multiprocessor, a lock implementation should block instead of spin if it is known that the lock will available before the time required for a context-switch.
43) Periodically yielding the processor while spin waiting reduces the amount of wasted time to be proportional to the duration of a context switch.
44) When a thread returns from a call to cond_wait() it can safely assume that it holds the corresponding mutex.
45) When a thread returns from a call to cond_wait() it can safely assume that the situation it was waiting for is now true.
46) The call cond_signal() releases the corresponding mutex.
47) With producer/consumer relationships and a finite-sized circular shared buffer, producing threads must wait until there is an empty element of the buffer.
48) To implement a thread_join operation with a semaphore, the semaphore is initialized to the value of 0 and the thread_exit() code calls sem_wait().
49) The safety property for dining philosophers states that it is not the case that there exists a philosopher who is hungry and his/her neighbors are not eating.
50) With a reader/writer lock, either multiple readers can hold the lock or a single writer can hold the lock (or no one holds the lock).
51) A thread can hold only one lock at a time.
52) Deadlock can be avoided by using semaphores instead of locks for mutual exclusion.
53) Deadlock can be avoided if one thread does not acquire any locks. (REMOVED QUESTION)
Part 4: Straight-forward True/False about I/O Devices [2 points each]
Designate if each statement is True (a) or False (b).

54) The peripheral bus used by hard disk drives tends to provide lower bandwidth than the memory bus.
55) A device driver runs on the microcontroller that is part of the external device.
56) When interacting with a fast device, it can be better to spin wait than to use interrupts.
57) To use DMA, the OS must inform the device of the address for the relevant data in main memory.
58) The interface that modern hard disk drives expose to the OS is that of tracks on surfaces.
59) A hard disk drive can have more surfaces than platters.
60) A hard disk drive can have more r/w heads than surfaces.
61) A seek on a modern disk drive could take around 1 second.
62) With a 7200 RPM disk, the average rotation time for a random read is expected to be around 8.3 ms.
63) On a modern disk, a workload of sequential reads will be about twice as fast as a workload of random writes.
64) Track skew accounts for the amount a disk rotates while the r/w head seeks for one track.
65) On a disk with zones, large files that will be accessed sequentially should be placed on the outer tracks instead of the inner tracks.
66) Shortest-positioning-time-first can be implemented more accurately inside of a disk than within the OS.
67) Shortest-positioning-time-first treats I/O requests from different processes fairly.
68) The elevator algorithm takes rotational time into account when scheduling I/O requests.
69) An anticipatory scheduler may leave the disk idle even when there are waiting I/O requests.
Part 5. Fork and Thread_Create() [4 points each]

For the next two questions, assume the following code is compiled and run on a modern linux machine (assume any irrelevant details have been omitted):

```c
main() {
    int a = 0;
    int rc = fork();
    a++;
    if (rc == 0) {
        rc = fork();
        a++;
    } else {
        a++;
    }
    printf("Hello!\n");
    printf("a is %d\n", a);
}
```

70) Assuming `fork()` never fails, how many times will the message “Hello!\n” be displayed?
   a) 2
   b) 3
   c) 4
   d) 6
   e) None of the above

71) What will be the **largest** value of “a” displayed by the program?
   a) Due to race conditions, “a” may have different values on different runs of the program.
   b) 2
   c) 3
   d) 5
   e) None of the above
Part 5 continued.

For the next two questions, assume the following code is compiled and run on a modern linux machine (assume any irrelevant details have been omitted):

```c
volatile int balance = 0;

void *mythread(void *arg) {
  int i;
  for (i = 0; i < 200; i++) {
    balance++;
  }
  printf("Balance is %d\n", balance);
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p1, p2, p3;
  pthread_create(&p1, NULL, mythread, "A");
  pthread_join(p1, NULL);
  pthread_create(&p2, NULL, mythread, "B");
  pthread_join(p2, NULL);
  pthread_create(&p3, NULL, mythread, "C");
  pthread_join(p3, NULL);
  printf("Final Balance is %d\n", balance);
}
```

72) Assuming none of the system calls fail, when thread p1 prints “Balance is %d\n”, what will p1 say is the value of balance?
   a) Due to race conditions, “balance” may have different values on different runs of the program.
   b) 200
   c) 400
   d) 600
   e) None of the above

73) Assuming none of the system calls fail, 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, “balance” may have different values on different runs of the program.
   b) 200
   c) 400
   d) 600
   e) None of the above
Part 6. Impact of scheduling without locks (assembly code) [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).

```assembly
# 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 the 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). The code continues on the next page.

For each of the lines designated below with a question numbered 74-79, determine the contents of the memory address 2000 AFTER that assembly instruction executes.

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>74) Contents of addr 2000?</th>
<th>75) Contents of addr 2000?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000 mov 2000, %ax</td>
<td>1000 mov 2000, %ax</td>
<td>1002 mov %ax, 2000</td>
<td>1002 mov %ax, 2000</td>
</tr>
<tr>
<td>1001 add $1, %ax</td>
<td>1001 add $1, %ax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1002 mov %ax, 2000</td>
<td>1000 mov 2000, %ax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------ Interrupt ------</td>
<td>------ Interrupt ------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1003 sub $1, %bx</td>
<td>1003 sub $1, %bx</td>
<td>1004 test $0, %bx</td>
<td>1004 test $0, %bx</td>
</tr>
<tr>
<td>1005 jgt .top</td>
<td>1005 jgt .top</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------ Interrupt ------</td>
<td>------ Interrupt ------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000 mov 2000, %ax</td>
<td>1000 mov 2000, %ax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001 add $1, %ax</td>
<td>1001 add $1, %ax</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
1002 mov %ax, 2000
1003 sub $1, %bx

6) Contents of addr 2000?

6) Contents of addr 2000?

7) Contents of addr 2000?

7) Contents of addr 2000?

8) Contents of addr 2000?

8) Contents of addr 2000?

1000 mov 2000, %ax
1001 add $1, %ax
1002 mov %ax, 2000
1003 sub $1, %bx
1004 test $0, %bx

------ Interrupt ------
1002 mov %ax, 2000
1003 sub $1, %bx
1004 test $0, %bx

------ Interrupt ------

------ Interrupt ------

------ Interrupt ------

------ Interrupt ------

------ Interrupt ------

------ Interrupt ------

------ Interrupt ------

------ Interrupt ------

------ Interrupt ------

------ Interrupt ------

------ Interrupt ------

------- Halt;Switch -------
1005 jgt .top
1006 halt

------- Halt;Switch -------

------- Halt;Switch -------

------- Halt;Switch -------

------- Halt;Switch -------

------- Halt;Switch -------

------- Halt;Switch -------

------- Halt;Switch -------

80) Assume looping-race-nolock.s is run with an unknown scheduler and some random interleaving of instructions occurs across threads 1 and 2 (i.e., not just the interleaving shown above). For an arbitrary, unknown schedule, what contents of the memory address 2000 are possible when the two threads are done and the program is completed?

a) Any values >= 0 and <= 6
b) Any values >= 1 and <= 6
c) Any values >= 3 and <= 6
d) Any values >= 4 and <= 6
e) None of the above
**Part 6: Wait-Free Algorithms [12 total points]**

Your project partner has written the following correct implementation of `insert(int val)` using traditional locks for mutual exclusion:

```c
typedef struct {
    int val;
    node_t *next;
} node_t;

node_t *head; // assume points to an existing list
void insert (int val) {
    node_t *n = Malloc(sizeof(*n));
    n->val = val;
    lock(&m);
    n->next = head;
    head = n;
    unlock(&m);
}
```

You decide you would like to replace the locks 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 `insert()` modified to use `CmpAndSwap()` looks something like the following:

```c
node_t *head; // assume points to an existing list
void insert (int val) {
    node_t *n = Malloc(sizeof(*n));
    n->val = val;
    do {
        n->next = head;
    } while ( ??? );
}
```

Which of the following are correct replacements for `??` in the code above?

**For questions 81-88, mark the suggested replacement for `??` as Possible (a) or Not Possible (b).**

81) `CmpAndSwap(&n->next, head, n)`
82) `!CmpAndSwap(&n->next, head, n)`
83) `CmpAndSwap(&n->next, n, head)`
84) `!CmpAndSwap(&n->next, n, head)`
85) `CmpAndSwap(&head, n->next, n)`
86) `!CmpAndSwap(&head, n->next, n)`
87) `CmpAndSwap(&head, n, n->next)`
88) `!CmpAndSwap(&head, n, n->next)`
Part 7: Lock implementation with blocked threads [21 total points]
Your project partner started to write the code for implementing lock acquire() and lock release(), but the code isn't quite working yet. Your job is to finish it up. The following code is very similar to the code in lecture, except it has a problem.

typedef struct {
    bool lock = false;
    bool guard = false;
    queue_t q;
} LockT;

void acquire(LockT *l) {
    // A0
    while (TAS(&l->guard, true));  // A1
    if (l->lock) {  // A2
        qadd(l->q, tid);  // A3
        setpark(); // notify of plan  // A4
        park(); // unless unpark()  // A5
    } else {  // A6
        l->lock = true;  // A7
    }
}  // A8

void release(LockT *l) {
    while (TAS(&l->guard, true));
    if (qempty(l->q)) l->lock=false;
    else unpark(qremove(l->q));
    l->guard = false;
}

The problem is that acquire() does not set l->guard=false. Where in the code could the statement l->guard=false be correctly placed? Note that the statement may need, or may be able, to be placed in multiple locations. For questions 89-95, mark each location as Possible (a) or Not Possible (b) to provide a correct implementation for both lock acquire() and release().

89) Between A0 and A1
90) Between A1 and A2
91) Between A2 and A3
92) Between A3 and A4
93) Between A4 and A5
94) Between A6 and A7
95) Between A7 and A8

Although it was correct, you do not like the structure of two lines of code in release():
   if (qempty(l->q)) l->lock=false;
   else unpark(qremove(l->q));
and you decide to restructure those two lines to:
   if (!qempty(l->q)) unpark(qremove(l->q));
   l->lock=false;

96) What additional changes do you now need to make in order for all the code to be correct?
   a) No other changes are needed
   b) Place l->lock=true between A3 and A4
   c) Place l->lock=true between line A4 and A5
   d) Place l->lock=true after line A5
   e) Other changes beyond options b, c, and d are needed to make the code correct
Part 8. Scheduling multi-threaded C code [2 points each]
The following problem ask you to step through C code according to a specific schedule of threads. To understand how the scheduler switches between threads, you must understand the following model. This is identical to what was presented in previous exams and examples.

Imagine you have two threads, T and S. The scheduler runs T and S such that each statement in the C-language language code (or line of code as written in our examples) is atomic. We tell you which thread was scheduled by showing you either a "T" or a "S" to designate that one line of C-code was scheduled by the corresponding thread; for example, TTTSS means that 3 lines were run from thread T followed by 2 lines from thread S.

Assume the test for a while() loop or an if() statement corresponds to one line of C-code.

Function calls that may have to wait for something to happen (e.g., sem_wait()) are treated specially.

For sem_wait(), assume that the function call and return of sem_wait() requires exactly one scheduling interval if the calling process does not need to wait (i.e., it completes with just "T"). If the semaphore requires some other operation to occur in order for sem_wait() to complete, then assume the call spin-waits until that other operation occurs (e.g., you may see a long instruction stream "TTTTTTT" that causes no progress for this thread); once the other operation occurs, the next scheduling of the waiting thread causes that thread to finish the call to sem_wait() (i.e., it completes then with just "T").

Consider the following code for implementing rw locks; it continues on the next page. It is identical to the code presented in lecture.

typedef struct _rwlock_t {
    sem_t lock;
    sem_t writelock;
    int readers;
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;
    sem_init(&rw->lock, ???);
    sem_init(&rw->writelock, ???);
}
void rwlock_acquire_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock); // AR1
    rw->readers++; // AR2
    if (rw->readers == 1) // AR3
        sem_wait(&rw->writelock); // AR4
    sem_post(&rw->lock); // AR5
}

// 1 line of Critical Section C1 after acquire

void rwlock_release_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock); // RR1
    rw->readers--; // RR2
    if (rw->readers == 0) // RR3
        sem_post(&rw->writelock); // RR4
    sem_post(&rw->lock); // RR5
}

rwlock_acquire_writelock(rwlock_t *rw){sem_wait(&rw->writelock);} // AW1

// 1 line of Critical Section C1 after acquire

rwlock_release_writelock(rwlock_t *rw){sem_post(&rw->writelock);} // RW1

Assume rwlock_init() has already been called. Assume there are 3 threads, W, R, and S that want to execute the following requests:

W: rwlock_acquire_writelock(); 1 line of critical section C1 (not shown); rwlock_release_writelock();
R: rwlock_acquire_readlock(); 1 line of critical section C1 (not shown); rwlock_release_readlock();
S: rwlock_acquire_readlock(); 1 line of critical section C1 (not shown); rwlock_release_readlock();

For example, after thread R completes rwlock_acquire_readlock() (specifically, after it finishes line AR5), it will execute in its critical section, executing one line, C1; it will then call rwlock_release_readlock() (and the next line it will execute will be RR1).

The exact ordering of these operations across W, R, and S will depend upon the scheduler as described below.

97) To begin, how should the semaphore rw->lock be initialized in the statement
    sem_init(&rw->lock, ???);
    a) -1
    b) 0
    c) 1
    d) 2
    e) None of the above

98) How should rw->writelock be initialized in sem_init(&rw->writelock, ???);
    a) -1
    b) 0
    c) 1
    d) 2
    e) None of the above
99) Now let us consider what happens when the scheduler starts running these three threads. Assume the scheduler runs one line from W; that is, W calls sem_wait() within rwlock_acquire_writelock(). After the instruction stream “W” (i.e., after the scheduler runs one line), which line of W's will be run when W is scheduled again?

   a) AW1 (hint: here if the condition for sem_wait(&rw->writelock) was not satisfied)
   b) C1 (hint: here if the condition for sem_wait(&rw->writelock) was met)
   c) RW1 (hint: don't know how this would happen)
   d) Code beyond RW1 (hint: don't know how this would happen)
   e) Nothing else makes sense for W

100) Assume the scheduler continues on with RRRR (i.e., the scheduler runs 4 lines for one of the reader threads, and the full instruction stream is WRRRR). Which line will R execute when R is scheduled again?

   a) AR1
   b) AR3
   c) AR4
   d) AR5
   e) None of the above

101) Assume the scheduler continues on with SSSS (i.e., the scheduler runs 4 lines for another reader thread, S, and the full instruction stream is WRRRRSSSS). Which line will S execute when S is scheduled again?

   a) AR1
   b) AR3
   c) AR4
   d) AR5
   e) None of the above

102) Assume the scheduler continues on with WWW (i.e., the scheduler runs 3 lines for the original write thread, and the full instruction stream is WRRRRSSSSWW). Which line will W execute when W is scheduled again?

   a) AW1
   b) C1
   c) RW1
   d) Code beyond RW1
   e) Nothing else makes sense

103) Assume the scheduler continues on with SSSS (i.e., the scheduler runs 4 lines for reader thread, S, and the full instruction stream is WRRRRSSSSWWSSSS). Which line will S execute when S is scheduled again?

   a) AR1
   b) AR3
   c) AR4
   d) AR5
   e) None of the above
104) Assume the scheduler continues on with RRRRR (i.e., the scheduler runs 5 lines for reader thread, R, and the full instruction stream is WRRRRSSSSWWSSSSSSRRRR). Which line will R execute when R is scheduled again?
   a) AR1
   b) AR3
   c) AR4
   d) AR5
   e) None of the above

105) Assume the scheduler continues on with SS (i.e., the scheduler runs 2 lines for reader thread, S, and the full instruction stream is WRRRRSSSSWWSSSSSSRRRRSS). Which line will S execute when S is scheduled again?
   a) AR1
   b) AR3
   c) AR4
   d) AR5
   e) None of the above

106) Assume the scheduler continues on with RRR (i.e., the scheduler runs 3 lines for reader thread, R, and the full instruction stream is WRRRRSSSSWWSSSSSSRRRRSSRRR). Which line will R execute when R is scheduled again?
   a) RR1
   b) RR2
   c) RR3
   d) RR4
   e) None of the above

107) Assume the scheduler continues on with SSSS (i.e., the scheduler runs 4 lines for reader thread, S, and the full instruction stream is WRRRRSSSSWWSSSSSSRRRRSSRRRSSSS). Which line will S execute when S is scheduled again?
   a) RR1
   b) RR2
   c) RR3
   d) RR4
   e) None of the above

Congratulations on finishing! See you in lecture tomorrow to talk about file systems!