CS 537: Introduction to Operating Systems
Fall 2023: Midterm Exam #2

This exam is closed book, closed notes.

No calculators may be used. All cell phones must be turned off and put away.

You have 1 hour 30 minutes to complete this exam.

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

- CANVAS LAST NAME - fill in your last (family) name starting at the leftmost column
- CANVAS FIRST NAME - fill in the first five letters of your first (given) name
- IDENTIFICATION NUMBER - This is your UW Student WisCard ID number
- ABC of SPECIAL CODES - Write your lecture number as a three digit value:
    - 001 (Morning Lecture)
    - 002 (Afternoon Lecture)
    - 010 (Epic Lecture)

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

You may separate the pages of this exam if it helps you.

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

- The OS manages a single uniprocessor (single core)
- All memory is byte addressable
- The terminology lg means $\log_2$
- 2^{10} bytes = 1KB
- 2^{20} bytes = 1MB
- 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).
- Hex numbers are represented with a proceeding “0x”

This exam has 68 questions. Each question has the same number of points.

Good luck!

Part 1. Designate if the statement is True (A) or False (B).

1. Threads that are part of the same process share the same stack.

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

3. A context switch between threads of the same process requires flushing the TLB or tracking the ASID.

4. One advantage of multi-threaded programming is parallelism, allowing the program to take advantage of multiple CPUs.

5. Another advantage of multi-threaded programming is with IO-bound workloads since a thread can run the IO-bound instructions, not blocking the other threads.

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

7. A lock implementation should always spin instead of block if it is known that the lock will always be used only on a uniprocessor.

8. A lock implementation on a multiprocessor should always spin instead of blocking if it is known that the lock will be available before the time of a context-switch.

9. A spin-wait type lock can provide fairness across the threads (i.e. threads receive the lock in the order they requested it).

10. When performing thread traces, you can assume locks will be acquired in the order they are requested.

11. Failing to surround a critical section of code with a lock may result in a deadlock situation.

12. A condition variable contains a queue of waiting threads.

13. While not required, it is best practice to hold the mutex lock when calling pthread_cond_wait().

14. While not required, it is best practice to hold the mutex lock when calling pthread_cond_signal().

15. The performance of broadcasting to a condition variable decreases as the number of waiting threads increases.

16. One can safely modify the state associated with a condition variable without holding the mutex lock.

17. When condition variable signaling follows Mesa semantics it is not necessary to recheck the program state when a thread returns from a wait().

18. It is always correct to use broadcast instead of signal when using Mesa semantics.

19. When a thread returns from a cond_wait() it is safe to assume that it holds the corresponding mutex lock.

20. In the attempted solution to the producer/consumer problem using a single condition variable, a consumer thread may signal another consumer thread to awake which would cause corruption of the data in the buffer.

21. In the solution to the producer/consumer problem with a finite sized circular shared buffer, consuming threads must wait until there is an empty element of the buffer.

22. Semaphores can be used instead of mutex locks and instead of condition variables.

23. A semaphore used as a condition variable is called a binary semaphore.

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

25. Deadlock can be avoided by using semaphores instead of locks for mutual exclusion.

26. A semaphore used to create mutual exclusion is initialized to 1.

27. In the approximate counter algorithm, the threshold parameter that controls the transfer from local to global counter controls the tradeoff between accuracy and performance.

28. Increasing the number of locks that are used to access a thread-safe data structure (like the hashtable compared to the linked-list example from the book) is guaranteed to increase concurrency and performance.

29. With reader-writer locks as implemented in the textbook, a write thread must wait until all reader threads have released their lock. With many busy reader threads the writer thread may starve.

30. Livelock is when a thread holds a resource but can't acquire the remaining needed resources and so must release the resource it holds.

Part 2. Forks and Threads

Assume the following code is compiled and run on a modern Linux machine. Assume any irrelevant details have been omitted and that no routines, such as fork(), ever fail.

```
int a = 0;

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

31. How many times will the message "Hello!\n" be displayed?
A. 2
B. 3
C. 4
D. 6
E. None of the above

32. What will be the final value of "a" as displayed by the final line of the program?
A. The value of "a" may be printed multiple times in different orders with different values.
B. 4
C. 8
D. 13
E. None of the above

The next set of questions looks at creating new threads. For the next questions, assume the following code is compiled and run on a modern Linux machine. Assume irrelevant details have been omitted and that no routines, such as pthread_create() or pthread_join(), ever fail.

```
volatile int balance = 0;

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

int main() {
   pthread_t p1, p2;
   pthread_create(&p1, NULL, mythread, "A");
   pthread_create(&p2, NULL, mythread, "B");
   pthread_join(p1, NULL);
   pthread_join(p2, NULL);
   printf("Final Balance is %d\n", balance);
}
```

33. How many total threads are part of this process?
A. 1
B. 2
C. 3
D. 4
E. None of the above

34. When thread p1 prints "Result is %d\n", what value of result will be printed?
A. Due to race conditions, "result" may have different values on different runs of the program.
B. 0
C. 200
D. 400
E. A constant value, but none of the above

35. When thread p1 prints "Balance is %d\n", what value of balance will be printed?
A. Due to race conditions, "balance" may have different values on different runs of the program.
B. 0
C. 200
D. 400
E. A constant value, but none of the above

36. When "Final Balance is %d\n" is printed, what value of balance will be printed?
A. Due to race conditions, "balance" may have different values on different runs of the program.
B. 0
C. 200
D. 400
E. A constant value, but none of the above

Part 3. Scheduling Without Locks (assembly code)

For the next questions, assume that two threads are running the following code on a uniprocessor (This is the same looping-race-nolocks.s code from homework simulations). 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.

```
# assumes %bx has loop count in it
.main
.top
# critical section
mov 2000, %ax  # get 'value' at address 2000
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
```

Assume that the scheduler runs the two threads producing the following order of instructions (the first column shows the address of the executed instruction) on the next page.

```
       Thread 0                Thread 1
1000 mov 2000, %ax
1001 add $1, %ax
------ Interrupt ------  ------ Interrupt ------
                         1000 mov 2000, %ax
                         1001 add $1, %ax
------ Interrupt ------  ------ Interrupt ------
1002 mov %ax, 2000                                  37.
------ Interrupt ------  ------ Interrupt ------
                         1002 mov %ax, 2000         38.
------ Interrupt ------  ------ Interrupt ------
1003 sub  $1, %bx
1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
                         1003 sub  $1, %bx
------ Interrupt ------  ------ Interrupt ------
1005 jgt .top
1000 mov 2000, %ax
------ Interrupt ------  ------ Interrupt ------
                         1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
1001 add $1, %ax
------ Interrupt ------  ------ Interrupt ------
                         1005 jgt .top
                         1000 mov 2000, %ax
------ Interrupt ------  ------ Interrupt ------
1002 mov %ax, 2000                                  39.
1003 sub  $1, %bx
------ Interrupt ------  ------ Interrupt ------
                         1001 add $1, %ax
                         1002 mov %ax, 2000         40.
------ Interrupt ------  ------ Interrupt ------
1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
                         1003 sub  $1, %bx
                         1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
1005 jgt .top
1000 mov 2000, %ax
------ Interrupt ------  ------ Interrupt ------
                         1005 jgt .top
------ Interrupt ------  ------ Interrupt ------
1001 add $1, %ax
1002 mov %ax, 2000
------ Interrupt ------  ------ Interrupt ------
                         1000 mov 2000, %ax
                         1001 add $1, %ax
------ Interrupt ------  ------ Interrupt ------
1003 sub  $1, %bx
1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
                         1002 mov %ax, 2000
                         1003 sub  $1, %bx
------ Interrupt ------  ------ Interrupt ------
1005 jgt .top
------ Interrupt ------  ------ Interrupt ------
                         1004 test $0, %bx
                         1005 jgt .top
------ Interrupt ------  ------ Interrupt ------
1006 halt
----- Halt;Switch -----  ----- Halt;Switch -----
                         1006 halt
```

For each of the lines designated above with a question numbered 37-40, determine the contents of the memory address 2000 AFTER that assembly instruction executes. Use the following options for questions 37-40:

A. 1
B. 2
C. 3
D. 4
E. None of the above

41. What would be the expected value for the final content of address 2000 if there had been no race conditions between the two threads?
A. 1
B. 2
C. 3
D. 4
E. None of the above

42. Assume looping-race-nolocks.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 poss*ible 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 4. Non-Blocking Data Structures

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

```
typdef struct {
   int val;
   node_t *next;
} node_t;

node_t *head;
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 `CompareAndSwap(int *addr, int expect, int new)` which returns 0 on failure and 1 on success. Recall that it changes the value of *addr to new if *addr = expect. You know that insert() modified to use CompareAndSwap() looks something like the following:

```
node_t *head; //assume points to 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? Select (A) True, or (B) False.

43. CompareAndSwap(&n->next, head, n)

44. !CompareAndSwap(&n->next, head, n)

45. CompareAndSwap(&n->next, n, head)

46. !CompareAndSwap(&n->next, n, head)

47. CompareAndSwap(&head, n->next, n)

48. !CompareAndSwap(&head, n->next, n)

49. CompareAndSwap(&head, n, n->next)

50. !CompareAndSwap(&head, n, n->next)

Part 5. Spin-Locking with Atomic Hardware Instructions

Consider the following incomplete implementation of a spin-lock. Examine it carefully because it is not identical to what was shown in class.

```
typedef struct __lock_t {
   int flag;
} lock_t;

void init(lock_t *lock) {
   lock->flag = abc;
}

void acquire(lock_t *lock) {
   while(xchg(&lock->flag, xyz) == xyz)
      ; //spin-wait
}

void release(lock_t *lock) {
   lock->flag = 1;
}
```

51. To what value or values could lock->flag be initialized (shown as **abc** above) to obtain a correct implementation?
A. 0
B. 1
C. 2
D. 0 or 2
E. 0 or 1

52. To what value or values could lock->flag be compared to acquire() (shown as **xyz** above) to obtain a correct implementation?
A. 0
B. 1
C. 2
D. 0 or 2
E. 0 or 1

Part 6. Reusable Barrier Lock

Imagine a problem where many threads are executing the same piece of sequential code and all the threads need to complete the section of code before moving on to the next section of code. This rendezvousing needs to occur multiple times as the code is executed. So the sequential code looks something like this:

P1
barrier()
P2
barrier()
P3
...

All threads must complete P1 before any thread continues on to P2. All threads must then complete P2 before any thread continues on to P3. This process occurs over and over as the sequential code executes.

It is safe to assume you know the total number of running threads and that all threads will try and enter the barrier. You have created a partial implementation of a barrier lock that uses condition variables and mutex locks as shown below:

```
typedef struct __barrier_t {
   pthread_mutex_t lock;
   pthread_cond_t cond;
   int at_barrier;
   int total_threads;
} barrier_t;

void init(barrier_t *b, int num_threads) {
   pthread_mutex_init(&b->lock, NULL);
   pthread_cond_init(&b->cond, NULL);
   b->total_threads = num_threads;
   b->at_barrier = 0;
}
```

```
void barrier(barrier_t *b) {
   pthread_mutex_lock(&b->lock);
   b->at_barrier++;
   if (b->at_barrier < b->total_threads) {
      pthread_cond_wait(&b->cond, &b->lock);
      //FINISH POINT 1
   }
   //FINISH POINT 2
   pthread_mutex_unlock(&b->lock);
}
```

For each of the following attempted completions of the barrier() function select which is most accurate about the solution.

53. **FINISH POINT 1 contains:**

```
//no additional code
```

FINISH POINT 2 contains:

```
pthread_cond_signal(&b->cond);
b->at_barrier--;
```

A. Correct Implementation
B. Correct implementation but more signaling than necessary
C. Atomicity Violation
D. Order Violation
E. Deadlock

54. **FINISH POINT 1 contains:**

```
return;
```

FINISH POINT 2 contains:

```
pthread_cond_broadcast(&b->cond);
b->at_barrier=0;
```

A. Correct Implementation
B. Correct implementation but more signaling than necessary
C. Atomicity Violation
D. Order Violation
E. Deadlock

55. **FINISH POINT 1 contains:**

```
//no additional code
```

FINISH POINTS 2 contains:

```
pthread_cond_broadcast(&b->cond);
b->at_barrier=0;
```

A. Correct Implementation
B. Correct implementation but more signaling than necessary
C. Atomicity Violation
D. Order Violation
E. Deadlock

56. **FINISH POINT 1 contains:**

```
pthread_mutex_unlock(&b->lock);
return;
```

FINISH POINT 2 contains:

```
pthread_cond_broadcast(&b->cond);
```

A. Correct Implementation
B. Correct implementation but more signaling than necessary
C. Atomicity Violation
D. Order Violation
E. Deadlock

57. **FINISH POINT 1 contains:**

```
pthread_mutex_unlock(&b->lock);
return;
```

FINISH POINT 2 contains:

```
pthread_cond_broadcast(&b->cond);
b->at_barrier=0;
```

A. Correct Implementation
B. Correct implementation but more signaling than necessary
C. Atomicity Violation
D. Order Violation
E. Deadlock

Part 7. Reader/Writer Locks with Semaphores

The following is a correct implementation of reader/writer locks that uses semaphores and incorporates a technique so writer threads will not starve waiting for reader threads. Assume the code compiles and works 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 (line 1 of Release_writelock)
   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
}
```

58. What value should the semaphore mutex be initialized to?
A. 0
B. 1
C. The number of reader threads
D. The number of writer threads
E. None of the above

59. What value should the semaphore OKToRead be initialized to?
A. 0
B. 1
C. The number of reader threads
D. The number of writer threads
E. None of the above

60. What value should the semaphore OKToWrite be initialized to?
A. 0
B. 1
C. The number of reader threads
D. The number of writer threads
E. None of the above

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();
```

61. Where will reader thread R0 be in the code?
A. AR1
B. AR4
C. AR8
D. Beyond AR8
E. None of the above (including non-deterministic locations)

62. Where will thread W0 be in the code?
A. AW1
B. AW4
C. AW7
D. Beyond AW7
E. None of the above (including non-deterministic locations)

63. Where will thread R1 be in the code?
A. AR1
B. AR4
C. AR8
D. Beyond AR8
E. None of the above (including non-deterministic locations)

Continuing the same execution stream, 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).

64. Where will thread R0 be in the code?
A. RR1
B. RR7
C. RR8
D. Beyond RR8
E. None of the above (including non-deterministic locations)

65. Where will thread W0 be in the code?
A. AW1
B. AW4
C. AW7
D. Beyond AW7
E. None of the above (including non-deterministic locations)

66. Where will thread R1 be in the code?
A. AR1
B. AR4
C. AR8
D. Beyond AR8
E. None of the above (including non-deterministic locations)

67. 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?
A. AR1
B. AR7
C. AR8
D. Beyond AR8
E. None of the above (including non-deterministic locations)

68. Continuing the same execution stream, when writer thread W0 eventually calls release_writelock(), what will happen next before threads must block?
A. Only thread R1 will acquire the readlock, R2 will continue to wait
B. Thread R1 or R2 will acquire the readlock, but it is unknown which (the other waits)
C. First R1 will acquire the readlock, and then R2 will acquire the readlock, while R1 still also has the readlock
D. Both threads R1 and R2 will acquire the readlock concurrently, but the order the threads return from acquire_readlock() is unknown
E. None of the above (including non-deterministic locations)

You have reached the end of the exam!