CS 537: Introduction to Operating Systems
Fall 2024: Midterm Exam #2

This exam is closed book, but you may use 1 sheet of notes.

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

You have 1 hour and 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 – TuTh 9:30-10:45 (Louis)
    - 002 – TuTh 11:00-12:15 (Shivaram)

These exam questions, your sheet of notes, and the scantron must be returned at the end of the exam, but we will not grade anything in this booklet. You will not be getting your note sheet back.

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

- The OS manages a single uniprocessor (single core)
- All memory is byte addressable
- Page table and page directory 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).
- Hex numbers are represented with a proceeding "0x"

The following might help you with some calculations:

- 0x100 = 2^8 = 256
- 2^{10} = 1024
- 2^{12} = 4096
- 2^{10} bytes = 1KB
- 2^{20} bytes = 1MB

This exam has 50 questions. Each question has the same number of points.

Good luck!

True / False

1. Threading enables overlap of I/O with other activities within a single program. **T**
2. When a thread creates another thread, it waits until that other thread has finished executing. **F**
3. When a thread calls join(), it waits until all other threads finish executing. **F**
4. A context switch between threads of the same process requires restoring register values for the thread being switched to. **T**
5. A context switch between threads of different processes requires the flushing of the TLB or tracking of ASID. **T**
6. Spin locks should not be used if blocking locks are available. **T or F accepted, on a uniprocessor spinning is just wasting time, but on a multiprocessor sometimes spinning is preferred to blocking (when the wait time will not be long).**
7. A mutex lock is equivalent to a semaphore if the semaphore is initialized to any positive value. **F**
8. Disabling interrupts is a safe approach to guarantee mutual exclusion on a multiprocessor. **F**
9. If a thread holds a mutex lock then the thread prevents the OS from context switching to another thread. **F**
10. Adding one mutex lock around code accessing a shared data structure guarantees mutual exclusion and makes the data structure thread-safe. **T**
11. A thread must hold a mutex lock when calling either pthread_cond_wait() or pthread_cond_signal(). **T or F accepted (pthread_cond_wait() must hold lock, pthread_cond_signal() SHOULD hold lock).**
12. A call to pthread_cond_signal() will always wake up waiting threads in the order in which they called pthread_cond_wait() **F**
13. Calling pthread_cond_signal() when no threads are waiting will cause an error. **F**
14. A semaphore should be initialized to 0 to use it as a mutex lock. **F**
15. A thread calling sem_wait() will not block if the semaphore's value is not negative. **F (0 is not negative and a call to sem_wait() will block in this case.)**
16. A call to sem_post() always results in a thread being unblocked if there is any thread waiting. **T**
17. If a thread releases the resources it is holding because it cannot acquire the remaining resources it needs then deadlock cannot occur. **T**
18. One technique for preventing circular waits is to have a strict ordering on how resources are acquired. **T**
19. In the Stride scheduler, when a process enters or leaves the system, all processes `remain` values must be recalculated. **F**
20. The Stride scheduler allocates the CPU to processes in proportion to their assigned tickets. **T**

Multiple Choice

21. Which of the following is NOT shared between threads of the same process?
    A. Page Table
    B. Heap
    C. Program Counter Register
    D. Process ID
    E. Open file descriptors

22. We are given a binary semaphore, initialized with a value of 0. The following sequence of operations is performed on the semaphore: post, post, wait, post, wait, wait, wait, post. What is the final value of the semaphore?
    A. 1
    B. 0
    C. -1
    D. -2
    E. None of the above

23. Which of the following is NOT one of the properties that must be true in order for deadlock to occur?
    A. Mutual Exclusion
    B. Circular Wait
    C. No Preemption
    D. Hold and Wait
    E. **None of the above**

24. What is the primary purpose of a condition variable in thread synchronization?
    A. To allow threads to share a resource
    B. To ensure mutual exclusion
    C. To enable threads to wait for specific conditions to become true
    D. To prevent deadlock
    E. To allow a thread to preempt another

25. Failure to surround a critical section of code that accesses/changes a shared variable may result in what type of error?
    A. Deadlock
    B. Livelock
    C. Race condition
    D. Starvation
    E. None of the above

Threads

The next set of questions look 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 i;
   for (i = 0; i < 100; i++) {
      balance++;
   }
   printf("Balance is %d\n", balance);
   return NULL;
}

int main(int argc, char *argv[]) {
   pthread_t p1, p2, p3, p4;
   pthread_create(&p1, NULL, mythread, "A");
   pthread_create(&p2, NULL, mythread, "B");
   pthread_join(p1, NULL);
   pthread_join(p2, NULL);
   pthread_create(&p3, NULL, mythread, "C");
   pthread_create(&p4, NULL, mythread, "D");
   pthread_join(p3, NULL);
   pthread_join(p4, NULL);
   printf("Final Balance is %d\n", balance);
}
```

26. What is the maximum number of threads that could be executing at the same time?
    - A. 1
    - B. 2
    - **C. 3**
    - D. 4
    - E. None of the above

27. What will thread p1 print as the value of balance?
    - **A. Due to race conditions, balance may have different values on different runs of the program.**
    - B. 100
    - C. 200
    - D. 400
    - E. None of the above

28. What will the main parent thread print as the final value of balance?
    - **A. Due to race conditions, “balance” may have different values on different runs of the program.**
    - B. 100
    - C. 200
    - D. 400
    - E. None of the above

29. If the final pthread_join() call was not there, what is the smallest value that the main parent thread could print as the final value of balance?
    - A. 100
    - B. 200
    - C. 400
    - D. Not enough information
    - **E. None of the above**
      (this is a tricky one. Thread p1 and p2 are in a race condition as they run. The case that results in the smallest value as they run is that both threads copy the initial value of 0 into their register, then one thread does all iterations except for the last, then the other thread does one iteration, moving the value of 1 to memory; at this point the first thread grabs the value of 1 into its register, the second thread does all of its iterations and moves its value to memory, but then the first thread completes its last iteration (in which it adds one to 1 and moves the value 2 to memory). This same scenario can play out between threads p3 and p4. Even though the main thread is not required to wait, since the final pthread_join() was removed, it still could not be scheduled and the race condition could occur again resulting in the value of 4 in the balance variable.)

Locked Data Structures

Consider the following two pieces of code.

```
6 int Insert(list_t *L, int k) {
7   node_t *n = malloc(
8     sizeof(node_t));
9   if (n == NULL) {
10    perror("malloc");
11    return -1;
12  }
13  new->key = k;
14  pthread_mutex_lock(&L->lock);
15  new->next = L->head;
16  L->head = new;
```

```
6 int Insert(list_t *L, int k) {
7   pthread_mutex_lock(&L->lock);
8   node_t *new = malloc(
9    sizeof(node_t));
10  if (new == NULL) {
11    perror("malloc");
12    pthread_mutex_unlock(
13      &L->lock);
14    return -1; // fail
15  }
16  new->key = k;
```

```
  pthread_mutex_unlock(&L->lock);
  return 0; // success
}
```

```
  new->next = L->head;
  L->head = new;
pthread_mutex_unlock(&L->lock);
  return 0; // success
}
```

30. We use the implementation on the right to perform inserts on a shared linked list. As we increase the number of threads from 1 to 16 on a 4-core machine, we expect
    A. **The overall number of inserts per second will remain the same irrespective of the number of threads (because a mutex surrounds the entire code, only one thread will be able to do any work at a time)**
    B. We expect the number of inserts per second to linearly decrease as we increase the number of threads
    C. We expect the number of inserts per second to linearly increase as we increase the number of threads
    D. We expect the number of inserts per second to sub-linearly decrease as we increase the number of threads
    E. We expect the number of inserts per second to sub-linearly increase as we increase the number of threads

31. Comparing the two implementations of List insert, we expect that
    A. Both implementations scale similarly as we increase the number of threads
    B. **The implementation on the left scales better**
    C. The implementation on the right scales better
    D. Not enough information

Semaphores

The following program consists of 3 threads and 3 *binary* semaphores (denoted by S, A, B):

```
void thread_0() {
    while(1) {
        sem_wait(&S);
        printf("1");
        sem_post(&A);
```

```
void thread_A() {
    sem_wait(&A);
    sem_post(&S);
}
```

```
void thread_B() {
    sem_wait(&B);
    sem_post(&S);
}
```

`        sem_post(&B);`
`    }`
`}`		

32. If the initial values for the semaphore is **S = 0, A = 0, B = 0**, how many times “1” will be printed?
    A. Exactly 1
    B. Exactly 3
    C. At least 2
    D. Exactly 0
    E. None of the above

33. If the initial values for the semaphore is **S = 0, A = 1, B = 0**, how many times “1” will be printed?
    A. Exactly 1
    B. Exactly 3
    C. At least 2
    D. Exactly 0
    E. None of the above

34. If the initial values for the semaphore is **S = 1, A = 1, B = 0**, how many times “1” will be printed?
    A. Exactly 1
    B. Exactly 3
    C. At least 2
    D. Exactly 0
    E. None of the above

35. If the initial values for the semaphore is **S = 0, A = 0, B = 1**, how many times “1” will be printed?
    A. Exactly 1
    B. Exactly 3
    C. Exactly 2
    D. Exactly 0
    E. None of the above

Deadlock

Suppose there are N workers concurrently running the following worker function with arbitrary op_code parameters. As we can see, there are two different semaphores r0 and r1 declared in the following code. They are respectively initialized with value R0 and R1. N, R0, and R1 are all **positive** integers. do_task_A() and do_task_B() functions won't get stuck.

```
sem_t r0;   // Initialized with value R0
sem_t r1;   // Initialized with value R1

void *worker(void *op_code) {
   switch (*(int *)op_code) {
        case 0:
              sem_wait(&r0);
              sem_wait(&r1);

              do_task_A();

              sem_post(&r1);
              sem_post(&r0);
        case 1:
              sem_wait(&r1);
              sem_wait(&r0);

              do_task_B();

              sem_post(&r0);
              sem_post(&r1);
   }
}
```

Suppose the system is configured as described in each question below. For each question state if deadlock is possible (A) or NOT possible (B).

36. R0 = 3, R1 = 3, N = 6 - **A**
37. R0 = 1, R1 = 5, N = 4 - **B**
38. R0 = 2, R1 = 3, N = 6 - **A**

39. Which of the following statements is correct?
    A. As long as N, R0 and R1 satisfy the relation that N <= R0 + 1 and N <= R1 + 1, then a deadlock won't happen.
    B. A system is set up with specific values of N, R0 and R1, and one day a deadlock happens. We can infer that N should be greater or equal to R0 + R1.
    C. For any arbitrary values of R0, R1 and N, deadlock is always possible.
    D. Both A and B.
    E. None of the above (Option A is false. When N=2, r0=1, r1=1, the threads may deadlock if thread 1 takes the top option and context switch after getting r0 and thread 2 takes bottom option and gets r1 and then blocks trying to get r0. Context switch back and thread 1 tries to get r1 but it is held so blocks -- deadlock. Option B is false. This can be shown for N=2, r0=100, r1=0. A deadlock will always occur but you cannot infer N >= R0+R1.)

Consider a function of the form

```
      do_the_task (mutex_t *m1, mutext_t *m2)
```

In this function, both locks m1 and m2 should be grabbed (assume m1 and m2 can't be the same lock) before entering the critical section. Assume there are two mutexes L1 and L2 and multiple threads are concurrently calling either do_the_task(&L1, &L2) or do_the_task(&L2, &L1)

Suppose Bob and Alice come up with two solutions to grab the two locks. Bob's code looks like

```
If (m1 > m2) {
      pthread_mutex_lock(m1);
      pthread_mutex_lock(m2);
} else {
      pthread_mutex_lock(m2);
      pthread_mutex_lock(m1);
}
```

Alice's code looks like

```
// prevention is a mutex globally shared by all the threads
pthread_mutex_lock(prevention);
pthread_mutex_lock(m1);
pthread_mutex_lock(m2);
pthread_mutex_unlock(prevention);
```

40. Which of the two approaches above can prevent deadlock from happening?
    A. Only Alice can prevent deadlock

B. Only Bob can prevent deadlock
C. Both Alice and Bob can prevent deadlock
D. Neither can prevent deadlock

Now consider Sally's approach

```
while (1) {
     pthread_mutex_lock(m1);
     if (pthread_mutex_trylock(m2) !=0) {
          pthread_mutex_unlock(m1);
     } else {
          break;
     }
}
```

41. Which of the following statements is correct about this approach?
    A. Deadlocks are possible
    B. Both deadlocks and livelocks are possible
    C. Neither deadlocks nor livelocks are possible
    D. Livelocks are possible

Locks and Condition Variables

Four threads are started to work together to take turns printing out each integer once in sequential order from 1 up to some maximum. Two of the threads are executing printEven() and the other two threads are executing printOdd(). The following code is a partial implementation of the functions the threads are executing and the shared variables between them.

```
int MAX = 1000;
int counter = 1;              //the next number to be printed

//two threads execute this function
void* printEven(void* arg) {
   for (int i = 0; i < MAX; i++) {

      // COMPLETION BEFORE EVEN PRINT

      printf("Even: %d\n", counter);
      counter++;

      // COMPLETION AFTER EVEN PRINT
   }
   return NULL;
}




//two threads execute this function
void* printOdd(void* arg) {
   for (int i = 0; i < MAX; i++) {

      // COMPLETION BEFORE ODD PRINT

      printf("Odd: %d\n", counter);
      counter++;

      // COMPLETION AFTER ODD PRINT

   }
   return NULL;
}
```

For each of the following completions to the code, select the option that best describes this completed code.

42. `Create shared mutex lock and initialize it`

```
//COMPLETION BEFORE EVEN PRINT
pthread_mutex_lock(&mutex);

//COMPLETION AFTER EVEN PRINT
pthread_mutex_unlock(&mutex);

//COMPLETION BEFORE ODD PRINT
pthread_mutex_lock(&mutex);

//COMPLETION AFTER ODD PRINT
pthread_mutex_unlock(&mutex);
```

   A. **Race conditions will be prevented.**
   B. Deadlock may now occur.
   C. printEven() will only print even values and printOdd() will only print odd values.
   D. At most one thread running printEven() and one thread running printOdd() could be running simultaneously.

E. None of the above

43. Create shared mutex lock and cond condition variable and initialize

```
//COMPLETION BEFORE EVEN PRINT
pthread_mutex_lock(&mutex);
pthread_cond_wait(&cond,&mutex);

//COMPLETION AFTER EVEN PRINT
pthread_mutex_unlock(&mutex);

//COMPLETION BEFORE ODD PRINT
pthread_mutex_lock(&mutex);

//COMPLETION AFTER ODD PRINT
pthread_cond_wait(&cond,&mutex);
pthread_mutex_unlock(&mutex);
```

A. Just the value 1 will be printed.
B. Nothing will be printed.
C. Just the values 1 and 2 will be printed.
D. A or C
E. None of the above

44. Create shared mutex lock and cond condition variable and initialize

```
//COMPLETION BEFORE EVEN PRINT
pthread_mutex_lock(&mutex);
while (counter %2 != 0)
   pthread_cond_wait(&cond,&mutex);

//COMPLETION AFTER EVEN PRINT
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);

//COMPLETION BEFORE ODD PRINT
pthread_mutex_lock(&mutex);
while (counter %2 == 0)
```

```
   pthread_cond_wait(&cond,&mutex);

//COMPLETION AFTER ODD PRINT
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);
```

A. Deadlock may occur because a thread running printEven() may signal another thread running printEven()
B. Threads running printEven() will only print even values and threads running printOdd() will only print odd values
C. A race condition may occur
D. Both A and B
E. None of the above

45. `Create shared mutex lock and cond1 & cond2 condition variable and initialize`

```
//COMPLETION BEFORE EVEN PRINT
pthread_mutex_lock(&mutex);
while (counter %2 != 0)
   pthread_cond_wait(&cond1,&mutex);

//COMPLETION AFTER EVEN PRINT
pthread_cond_signal(&cond2);
pthread_mutex_unlock(&mutex);

//COMPLETION BEFORE ODD PRINT
pthread_mutex_lock(&mutex);
while (counter %2 == 0)
   pthread_cond_wait(&cond2,&mutex);

//COMPLETION AFTER ODD PRINT
pthread_cond_signal(&cond1);
pthread_mutex_unlock(&mutex);
```

A. Deadlock may occur because a thread running printEven() may signal another thread running printEven()
B. Threads running printEven() will only print even values and threads running printOdd() will only print odd values
C. A race condition may occur
D. Both A and B
E. None of the above

Lock Implementations, Hardware primitives, and Projects

46. The ticket lock implementation
    - **A. Provides fairness in that threads acquire the lock in the order they requested it.**
    - B. Is a blocking implementation
    - C. Uses spin-waiting when unlocking
    - D. Utilizes condition variables
    - E. A and D

47. Spinlocks are best to use if
    - A. The amount of time a thread will wait for another thread is short (less than the time for a context switch)
    - B. The system is a uniprocessor
    - C. The system is a multiprocessor
    - D. The amount of time a thread will wait for another thread is long (longer than the time for a context switch)
    - E. **A and C**

48. The Stride scheduler attempts to optimize
    - A. Turnaround time
    - B. Response time
    - **C. Fair sharing**
    - D. A and B
    - E. None of the above

49. The final value of the variable x after the following code is:
    ```
    int x = 2;
    int y = xchg(&x,3);
    int z = CompareAndSwap(&x,2,4);
    ```
    - A. 2
    - **B. 3**
    - C. 4
    - D. Insufficient information
    - E. None of the above

50. What is the primary purpose of a mutex lock in multithreaded programming?
    - A. To improve CPU performance

B. To speed up thread execution
C. To prevent memory leaks
D. To synchronize access to shared resources
E. To terminate threads