University of Waterloo
CS350 Midterm Examination

Winter 2019

Student Name: _____

Closed Book Exam
No Additional Materials Allowed
The marks for each question add up to a total of 74

1. (6 total marks)
For the following questions provide an answer and then justify your answer with a single sentence.

a. (2 marks) Efficiency
Which is typically faster and why:

i Printing the numbers from 1 to 1000000, one number at a time.

ii Creating a string with the numbers from 1 to 1000000 and printing that string.

Answer:

Justification:

(ii) is faster. (i) has 1 million system calls. (ii) has one.

b. (2 marks) Concurrency
Can you still have concurrency if you have a single processor with a single core and the degree of multithreading is one (i.e. P=1, C=1, M=1).

Answer:

Justification:

Answer: yes Justification: Preemption (or timesharing) will rapid.y switch between processes (or threads) so all make progress.

c. (2 marks) Synchronization
Can a lock be used anywhere a binary semaphore is used?

Answer:

Justification:

No. lock_acquire must be called before lock_release but semaphores allow P and V to be called in any order.

2. (10 total marks) For the following questions point form answers are preferred.

a. (2 marks) Concurrency
List two possible advantages of concurrency.

Improved CPU utilization Improved performance

b. (4 marks) Context Switching There are a number of ways that a context switch can occur.

i Which ones are prevented when interrupts are turned off.

preemption

ii Which ones are prevented when each process only has a single thread of execution.

none

iii Which ones are not prevented by either of the two previous conditions.

thread_exit thread_yield Thread block or sleep

c. (4 marks) cv_wait
List, in order, the four steps of `cv_wait`. Do not list any of the KASSERTs.

1.

2.

3.

4.

1. wchan_lock(cv->wchan) 2. lock_release(lk) 3. wchan_sleep(cv->wchan) 4. lock_acquire(lk)

3. (14 total marks) For the following questions a single sentence answer will suffice.

a. (1 mark)
Under what case (or cases) does disabling interrupts enforce mutual exclusion.

If there is only a single core in a single processor.

b. (1 mark)
Give one disadvantage of a scheduling quantum that is too short (i.e., 1ms or less).

Too little, if any of the original threads instructions may be executed before it is preempted again.

c. (1 mark)
Give one disadvantage of a scheduling quantum that is too long (i.e., 1s or more).

Threads respond to events too slowly. OR Threads appear stuck to user. OR Does not give user the impression that threads execute in parallel.

d. (1 mark)
What do exceptions and interrupts have in common?

The kernel handles both of these (or control is transferred to the kernel in both of these situations).

e. (1 mark)
What would be a scenario (if any) where a kernel stack would have a trapframe pushed and popped without a switchframe also being pushed and popped.

ANY OF: A timer interrupt before the quantum expired OR An interrupt by another device

f. (1 mark)
What would be a scenario (in any) where a kernel stack would have a switchframe pushed and popped without a trapframe also being pushed and popped.

thread_yield in kernel code

g. (1 mark)
Why cant you have a pid of 0 in os/161?

Because fork returns 0 to the child so it cannot be a valid pid.

h. (2 marks)
A programmer is writing a program that requires two major (but independent) tasks to be performed and is trying to decide between

- using `fork` and assigning one task to the parent and one to the child or
- using `thread_fork` and assigning one task to each thread.

i What would be one advantage of using `fork`?

A bug in one process would not affect the other

ii What would be one advantage of using `thread_fork`?

It uses less memory OR it is easier to exchange information between the two threads.

i. (3 total marks)

Draw the user and kernel stacks for a process that is executing `sys_waitpid`. Show the top of the stack (where items are push on or popped off) at the bottom of the diagram.

USER app frames waitpid KERNEL trapframe mipstrap syscall sys_waitpid

j. (2 total marks)

Explain how the following kernel stack could come to be.

trapframe
trapframe

A bug in common_exception allowed interrupts to be enabled before mips_trap was called.

4. (10 marks)

The following pseudocode makes use of a semaphore. Replace the semaphore based implementation with a condition variable based implementation that performs the same task. You may only add up to three additional variables. **Your cv may not be used with a loop.**

```
Semaphore barrier; // initialized to 0

int CS350( void * v, long n )
{
    WriteMidterm()
    V(barrier);
}

int main()
{
    for ( int i = 0; i <  NUMTHREADS; i ++ )
        thread_fork(‘‘student’’, null, CS350, null, i );

    for ( int i = 0; i < NUMTHREADS; i ++ )
        P( barrier );
    MarkMidterms()
}
```

```
int count = 0; [1 mark]
lock countLock; [1 mark]
cv barrier; [1 mark]

int KernelFunction( void * v, long n )
{

    acquire( countLock ); [1 mark]
    count ++; [1 mark]
    if ( count == NUMTHREADS ) cv_signal( countLock, barrier ); [1 mark]
   release( countLock ); [1 mark]
}

int main()
{
    for ( int i = 0; i <  NUMTHREADS; i ++ )
        thread_fork( , null, KernelFunction, null, i );

   acquire( countLock ); [1 mark]
      if ( count != NUMTHREADS ) cv_wait( countLock, barrier ); [1 mark]
   release( countLock ); [1 mark]
}
```

5. (10 marks)

Consider the following implementation of a lock. Assume that sem is created with an initial count of 1.

```
lock_acquire( lock * lk )
{
    KASSERT( lk != NULL );
    P( lk->sem );
        while ( lk->owner != NULL )
        {
             wchan_lock( lk->wchan );
             V( lk->sem );
             wchan_sleep( lk->wchan );
             P( lk->sem );
        }
        lk->owner = curthread;
    V( lk->sem );
}


lock_release( lock * lk )
{
    KASSERT( lk != NULL );
    V( lk->sem );
        lk->owner = NULL;
    V( lk->sem );
}
```

(a) If this lock was used to protect a critical section, would it guarantee mutual exclusion?

No. (2 marks)

(b) If you answered yes to (a), why? If you answered no to (a), correct the code (you may edit the code in-place).

(c) If there are any other issues with the lock not related to mutual exclusion, correct them. Otherwise, indicate the implementation is correct.

```
lock_acquire( lock * lk )
{
    KASSERT( lk->owner != curthread );    // add this [2 marks]
    KASSERT( lk != NULL );
    P( lk->sem );
        while ( lk->owner != NULL )
        {
             wchan_lock( lk->wchan );
             V( lk->sem );
             wchan_sleep( lk->wchan );
             P( lk->sem );
        }
        lk->owner = curthread;
    V( lk->sem );
}


lock_release( lock * lk )
{
    KASSERT( lk->owner == curthread );    // add this  [2 marks]
    KASSERT( lk != NULL );
    V( lk->sem );                         // change to P [2 marks]
        lk->owner = NULL;
        wchan_wakeone( lk->wchan );       // add this [2 marks]
    V( lk->sem );
}
```

6. (8 marks)

A system uses segmented address space for its implementation of virtual memory. Suppose a process initially uses 48KB of memory for its heap. The process then runs low on heap space and requests 16 KB more space. Assume you have access to a procedure sbrk and that sbrk(16*1024) will request that the heap's space be increase by 16 KB by finding a new location in RAM for the heap segment. If successful it returns the new address, otherwise it return NULL.

In roughly 4–6 steps, describe the process that would be required to increase the process's address space.

- Check if there is enough space to accommodate the new, larger heap (1 mark) or return ENOMEM (1 mark).
- Allocate the new heap (1 mark)
- Copy the contents of the old heap to the new one. (2 marks)
- Update the procs relocation and limit values for the heap in the kernel (1 mark) and on the MMU (1 mark).
- Delete the old heap (1 mark)

7. (8 total marks)

A system uses 24-bit virtual addresses, 24-bit physical addresses and memory segmentation. There are four segments and two bits are used for the segment number. The relocation and limit for each of the segments are as follows.

Segment Number	Limit Register	Relocation Register
0x0	0x2 0000	0x40 0000
0x1	0xC 0000	0x70 0000
0x2	0x9 0000	0x50 0000
0x3	0xA 0000	0x60 0000

Translate the following addresses from virtual to physical. Clearly indicate what segment each address belongs to.

a. (2 marks) `0x01 D0D4`

Segment Number:

Address Translation:

segment 0: 0x41 D0D4

b. (2 marks) `0x22 CA10`

Segment Number:

Address Translation:

segment 0: segmentation violation

c. (2 marks) `0x33 47B8`

Segment Number:

Address Translation:

segment 0: segmentation violation

d. (2 marks) `0x1C F008`

Segment Number:

Address Translation:

segment 0: segmentation violation

8. (8 total marks)
Ideally any scheme to enforce mutual exclusion should satisfy the following constraints.

i. Only one thread is allowed in a critical section at a time.

ii. No assumptions can be made about the order different threads will access the critical section.

iii. A thread that is outside the critical section cannot prevent another thread from entering the critical section.

iv. At least one thread should be making progress.

v. There should be a bound on the time a thread must wait.

For each of four cases listed below there are only two threads, T1 and T2. Rather than use locks, the threads use the code below to provide mutual exclusion to a critical region. Each thread will be trying to access the critical region multiple time (i.e. not just once).

a. (2 marks) T1 and T2 both execute the same function.

```
   #define CLOSED 0
   #define OPEN   1
   volatile int Lock = OPEN; // global variable

   AccessCriticalRegion() {
1    while (Lock == CLOSED) {;} // busy wait
2    Lock = CLOSED;
3    CriticalSection();
4    Lock = OPEN;
   }
```

Does this scheme meet all the requirements?

If not, which requirement is not satisified? Give a scenario where that requirement would not be satisfied.

Breaks (i)

lock open, thread 1 gets past line 1 but before 2 and gets preempted

lock still open, thread 2 enters critical section

b. (2 marks) Here T1 and T2 use different functions to access the critical region.

```
  volatile int Last = 2; // global variable

  // Code for T1                  |    // Code for T2
  T1AccessCriticalRegion() {      |    T2AccessCriticalRegion() {
1   while (Last == 1){;}          |  1   while (Last == 2){;}
2   CriticalSection();            |  2   CriticalSection();
3   Last = 1;                     |  3   Last = 2;
  }                               |    }
```

Does this scheme meet all the requirements?

If not, which requirement is not satisified? Give a scenario where that requirement would not be satisfied.

Breaks (iii)

when Last =2 (as in init) then T2 is prevented from running even if T1 is not in its critical section

or

Breaks (ii) sense last=2 so T1 runs first gets 1.5 marks.

c. (2 marks) Here T1 and T2 use different functions to access the critical region.

```
  #define WANT_IN 1
  volatile int T1 = ! WANT_IN; // global variables
  volatile int T2 = ! WANT_IN;

  // Code for T1                  |    // Code for T2
  T1AccessCriticalRegion() {      |    T2AccessCriticalRegion() {
1   T1 = WANT_IN;                 |  1   T2 = WANT_IN;
2   while (T2 == WANT_IN){;}      |  2   while (T1 == WANT_IN){;}
3   CriticalSection();            |  3   CriticalSection();
4   T1 = ! WANT_IN;               |  4   T2 = ! WANT_IN;
  }                               |    }
```

Does this scheme meet all the requirements?

If not, which requirement is not satisified? Give a scenario where that requirement would not be satisfied.

Breaks (iv)

T1 executes line 1 then T2 runs and executes line 1

deadlock on line 2, both are waiting for the other to change the value of T1/T2

d. (2 marks) Here T1 and T2 use different functions to access the critical region.

```
#define WANT_IN 1
volatile int T1 = ! WANT_IN; // global variables
volatile int T2 = ! WANT_IN;
volatile int Last = 1;
```

```
 // Code for T1
 T1AccessCriticalRegion() {
1  while (1) {
2     T1 = WANT_IN;
3     if (T2 == !WANT_IN) {
4        break;
5     }
6    if (Last == 1) {
7       T1 = !WANT_IN;
8       while (Last == 1){;}
9    }
10  }
11 CriticalSection();
12 Last = 1;
13 T1 = !WANT_IN;
  }
```

```
 // Code for T2
 T2AccessCriticalRegion() {
1  while (1) {
2    T2 = WANT_IN;
3    if (T1 == !WANT_IN) {
4      break;
5    }
6    if (Last == 2) {
7      T2 = !WANT_IN;
8      while (Last == 2){;}
9    }
10 }
11 CriticalSection();
12 Last = 2;
13 T2 = !WANT_IN;
  }
```

Does this scheme meet all the requirements?

If not, which requirement is not satisified? Give a scenario where that requirement would not be satisfied.

It works.