University of Waterloo
Midterm Examination
Term: Fall Year: 2010

Student Family Name _____

Student Given Name _____

Student ID Number _____

Section : Circle one (Brecht 10:00) (Brecht 11:30)

Course Abberviation and Number:	CS 350	[**!!! Circle Your Section Below !!!**]
Course Title:	Operating Systems	**10:00 RCH 305**
Section(s):	2	**11:30 RCH 307**

Date of Exam:	October 28, 2010
Time Period	Start time: 7:00 pm End time: 9:00 pm
Duration of Exam:	120 minutes
Number of Exam Pages:	11 (including cover sheet)
	NO CALCULATORS, NO ADDITIONAL MATERIAL

Problem	Mark	Score	Marker's Initials
1	10		
2(a)	10		
2(b)	10		
3	10		
4	10		
5	14		
6	10		
7	16		
Total	90		

Problem 1 (10 marks)

a. **(2 mark(s))** Under what circumstances would it be **NECESSARY** for a thread to call `thread_yield`? Explain why.

b. **(2 mark(s))** If a processor has 38-bit virtual addresses, 42-bit physical addresses and a page size of 16 KB. How many bits are needed to represent the physical frame? **Explain your answer.**

c. **(2 mark(s))** Describe two circumstances under which a variable needs to be declared `volatile`.

d. **(2 mark(s))** According to Andrew Birrell's "An Introduction to Programming with Threads", what is a spurious wake-up?

e. **(2 mark(s))** What key feature is present in the implementation of the condition variables described in Andrew Birrell's "An Introduction to Programming with Threads", that is not present in the condition variables in OS/161.

Problem 2 (20 marks)

The problem stated below will be solved in two different ways. In the first part it will be solved using system calls associated with processes (e.g., `fork()`). In the second part it will be solved using OS/161 kernel functions associated with threads and thread synchronization (e.g., `thread_fork()`). In both parts try to use as much C code as possible, but if necessary use pseudo-code.

The problem is that a parent wants to obtain and print a value that is computed by its grandchild (i.e., the child of its child). The grandchild computes the value by calling `compute_magic_num()`.

a. **(10 mark(s))** For this first part you must use the `fork()` system call to create the child and grandchild and you must assume (as is the case in UNIX/Linux) that `waitpid` can only be called using the pid of a child. **Be sure to include comments to explain what your code is doing.**

```
int main(int argc, char *argv[])
{
   int magic = 0;









   printf("Magic Value = %d\n", magic);

}
```

b. **(10 mark(s))** Now use OS/161 kernel level thread and synchronization functions available in OS/161 (assuming assignment 1 has been completed) to create the required child and grandchild and to solve the given problem. Assume that OS/161 will call `main_thread()`. Fill in the details for it and add any other required variables, functions and/or procedures needed to solve the problem. **Be sure to include comments to explain what your code is doing.**

```
void main_thread()
{
   int magic = 0;









   printf("Magic Value = %d\n", magic);


}
```

Problem 3 (10 marks)
Study the code below and answer the questions that follow.

```
#define N    (10)
#define M    (100000)
struct lock *locks[N+1];

void threadcode(void *unused, unsigned long i)
{
   int count;
   for (count=0; count<M; count++) {
      lock_acquire(locks[i+1]);
         function(i);
      lock_release(locks[i+1]);
   }
}

void function(unsigned long i)
{
   lock_acquire(locks[i]);
      do_something();       /* Does not do any synchronization */
   lock_release(locks[i]);
}

int main(int argc, char *argv[])
{
   for(i=0; i<=N; i++) {
     locks[i] = lock_create("lockname");
   }

   for(i=0; i<N; i++) {
     thread_fork("thread", threadcode, NULL, i, NULL);
   }

}
```

Circle ONE of the following statements that best represents the situation for the code above and **explain your answer in the space below**.

- Deadlock can occur.
- Deadlock can not occur.
- It is not possible to determine if deadlock can or can not occur.

Problem 4 (10 marks)
Study the monitor code below and answer the questions that follow.

```
monitor myMonitor
{
   conditionVariable cv;

   void proc1(int x) {
      printf("A ");
      wait(cv);
      printf("B ");
   }

   void proc2(int x) {
      printf("C ");
      signal(cv);
      printf("D ");
   }
}
```

Assume that thread T_1 calls proc1 and after that thread T_2 calls proc2.

a. **(5 mark(s))** If the system implements Mesa-style monitors what would the output be? **Show the output and explain your answer.**

b. **(5 mark(s))** If the system implements Hoare-style monitors what would the output be? **Show the output and explain your answer.**

Problem 5 (14 marks)

Assume that the code below is running on OS/161 after the synchronization code in assignment 1 has been correctly implemented. Assume that all synchronization mechanisms are starvation free and that `init()` is called before T threads are created. Answer the questions below assuming that the T threads only call the procedure in question and they don't call any of the other procedures. After the procedure is executed the kernel exits, the kernel is restarted and everything is reinitialized before the next procedure is called (i.e, the next part of the question is done). In other words, assume that each part of the question is independent of the other parts of the question.

In each part below fill in the `/* MAX = _____ */` comment to indicate the **maximum number of threads** that could be executing code in the procedure being called. In some cases you may wish to express the solution as a function of the max or min of two or more values.

```
struct lock *lock1 = 0;
struct lock *lock2 = 0; /* lock2 and cv2 are used together */
struct cv *cv2 = 0;
struct lock *lock3 = 0;
struct lock *locks[N];
struct semaphore *sem1 = 0;
struct semaphore *sem2 = 0;

void init()
{
   lock1 = lock_create("lock1");
   lock2 = lock_create("lock2");  /* lock2 and cv2 are used together */
   cv2 = cv_create("cv2");
   lock3 = lock_create("lock3");
   for (i=0; i<N; i++) {
      locks[i] = lock_create("lock");
   }
   sem1 = sem_create("sem1", 10);
   sem2 = sem_create("sem2", 1);
}
```

a. **(2 mark(s))**

```
proc1()
{
   lock_acquire(lock1);
     sub_proc1();              /* MAX = _________________ */
   lock_release(lock1);
}
```

b. **(2 mark(s))**

```
proc2(int i)         /* Calling code ensures i = 0 ... N-1 */
{
   lock_acquire(locks[i]);
     sub_proc2();             /* MAX = _________________ */
   lock_release(locks[i]);
}
```

c. **(2 mark(s))**

```
proc3()
{
   P(sem1);
     sub_proc3();              /* MAX = _________________ */
   V(sem1);
}
```

d. **(2 mark(s))**

```
proc4()
{
   V(sem2);
     sub_proc4();              /* MAX = _________________ */
   P(sem2);
}
```

e. **(2 mark(s))**

```
proc5()
{
   lock_acquire(lock2);
     while (need_to_wait() == TRUE) {
       cv_wait(lock2, cv2);
     }
     sub_proc5();              /* MAX = _________________ */
   lock_release(lock2);
}
```

f. **(2 mark(s))**

```
proc6()
{
   lock_acquire(lock2);
     if (need_to_broadcast == TRUE) {
        cv_broadcast(lock2, cv2);
     }
     sub_proc6();              /* MAX = _________________ */
   lock_release(lock2);
}
```

g. **(2 mark(s))**

```
proc7()
{
   lock_acquire(lock3);
     sub_proc7();              /* MAX = _________________ */
}
```

Problem 6 (10 marks)

Assume a new physical memory technology has been invented that allows the physical memory subsystem to detect problems when reading from or writing to memory. Further assume that this is implemented on the MIPS processor used in SYS/161 and OS/161. Instead of silently failing or crashing the system, this system generates an exception to let the operating system know that the write failed (the exception in this case is `EX_PHYS_WRITE_FAIL`). In the case of a read failure it generates an `EX_PHYS_READ_FAIL` exception. It is possible that a location in memory could fail to be written to but could be read, or that a read could fail while a write could succeed. The virtual address used in generating the exception is stored in the global variable `BadVaddr`.

Explain for each of these exceptions what the operating system could do to handle these exceptions. Ideally we want the operating system to completely hide these exceptions from the program and user so they never know a problem occurred (i.e., program execution should continue). If it is not possible to hide the problem you should explain why it is not possible and how the kernel would handle the exception. To simplify the problem, assume that one and only one program is ever running and that we are only concerned with trying to hide problems from the currently executing program. Assume that paging is used and that the page size is predefined as PageSize.

a. **(3 mark(s))** Explain **in detail** the steps you would take to handle `EX_PHYS_READ_FAIL` and why.

b. **(7 mark(s))** Explain **in detail** the steps you would take to handle `EX_PHYS_WRITE_FAIL` and why.

Problem 7 (16 marks)
Some useful info: 2^{10} = 1 KB, 2^{20} = 1 MB, 2^{30} = 1 GB

a. **(4 mark(s))** In this part of the question all addresses, virtual page numbers and physical frame numbers are represented in octal, recall that each octal character represents 3 bits. Consider a machine with *27-bit* virtual addresses and a page size of 4096 bytes. During a program execution the TLB contains the following entries (all in octal).

Virtual Page Num	Physical Frame Num	Valid	Dirty
6	20	1	1
6125	50	1	1
0	10	0	0
612	40	0	1
61	30	1	0

If possible, explain how the MMU will translate the following virtual address (in octal) into a *33-bit* physical address (in octal). If it is not possible, explain what will happen and why. Show and **explain how you derived your answer.** Express the final physical address using all 33-bits.
Load from virtual address = 612521276.

b. **(4 mark(s))** Assume the same scenario as above. If possible, explain how the MMU will translate the following virtual address (in octal) into a *33-bit* physical address (in octal). If it is not possible, explain what will happen and why. Show and **explain how you derived your answer.** Express the final physical address using all 33-bits.
Store to virtual address = 61252127.

c. **(4 mark(s))** In this part of the question all addresses, virtual page numbers and physical frame numbers are represented in decimal. Consider a machine with *36-bit* virtual and physical addresses, and a page size of 12304 bytes. During a program execution the TLB contains the following entries (all in decimal).

Virtual Page Num	Physical Frame Num	Valid	Dirty
891	100	0	0
8910	200	1	0
89	300	0	0
8	400	1	0
891000	500	1	0

If possible, explain how the MMU will translate the following virtual address (in decimal) into a physical address (in decimal). If it is not possible, explain what will happen and why. Show and **explain how you derived your answer.** Express the final physical address using decimal digits only.
Store to virtual address = 10963376.

d. **(4 mark(s))** In this part of the question all addresses, virtual page numbers and physical frame numbers are represented in hexidecimal, recall that each hexidecimal character represents 4 bits. Consider a machine with *40-bit* virtual addresses and a page size of 1 MB. During a program execution the TLB contains the following entries (in hexidecimal).

Virtual Page Num	Physical Frame Num	Valid	Dirty
0x AC	0x 2C	1	1
0x AC135	0x 5D	1	0
0x 0	0x 1F	1	0
0x AC13	0x 40	0	1
0x AC1	0x 3	1	0
0x AC1350	0x 6C	1	0

If possible, explain how the MMU will translate the following virtual address into a *48-bit* physical address (in hexidecimal). If it is not possible, explain what will happen and why. Show and **explain how you derived your answer**. Express the final physical address using all 48-bits.
Load from virtual address = 0x AC 1350 AC13.