University of Waterloo
Midterm Examination
Term: Fall Year: 2010

Solution

begin solution

Grade breakdown: Fall 2010

Unadjusted.

Question	Q1	Q2a	Q2b	Q3	Q4	Q5	Q6	Q7	Total
Avg	3.31	3.90	4.56	7.25	7.42	9.79	2.54	11.93	51
Avg %	33.11	39.01	45.56	72.52	74.24	69.91	25.43	74.54	56
Out of	10	10	10	10	10	14	10	16	90
Max	10	10	10	10	10	14	10	16	90

Add 8% of each student for adjusted class average = 64%.
Below shows adjusted stats.

90-108	10
80-89	22
70-79	20
60-69	41
50-59	28
40-49	16
30-39	10
20-29	2
10-19	2

end solution

Problem 1 (10 marks)

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

——————begin solution——————

There are no timer interrupts.
It is the only way to ensure that the kernel can get control of the CPU.

——————end solution——————

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.**

——————begin solution——————

```
16 KB = 2^14
So 42 - 14 = 28
```

——————end solution——————

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

——————begin solution——————

When it is shared by multiple threads.
When it is can be modified by a device.

——————end solution——————

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

——————begin solution——————

It is when threads are awaken when they cannot make useful progress.

——————end solution——————

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.

——————begin solution——————

Alerts.

——————end solution——————

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.**

begin solution

```
/* P = parent process */
/* C = child process */
/* G = grandchild process */

int main(int argc, char *argv[])
{
   int magic_num = 0;
   int rc, status;
   int child_pid, grandchild_pid;

   /* This is the parent (P) */
   child_pid = fork();

   if (child_pid == 0) {
      /* This is the child (C) */
      grandchild_pid = fork();
      if (grandchild_pid == 0) {
         /* This is the grandchild (G) */
         exit(compute_magic_num());          /* Return the exit value to C */
      }
      /* Get exit value from G */
      rc = waitpid(grandchild_pid, &status, 0);
      /* Pass it to P*/
      exit(status);
   }

   /* Only the parent executes this */
   rc = waitpid(child_pid, &magic, 0);        /* Get exit value from C */
   printf("Magic Value = %d\n", magic);
}
```

end solution

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.**

begin solution

```
struct semaphore *value_ready = 0;
volatile int shared_value = 0;

void grandchild(void *unused, unsigned long unused2)
{
   shared_value = get_magic_num();
   /* Let the grand parent know that the grandchild has the value */
   V(value_ready);
   thread_exit();                          /* Thread is done now */
}

void child(void *unused, unsigned long unused2)
{
   thread_fork("grandchild", grandchild, 0, 0, 0);
   thread_exit();      /* Just create the grandchild an we are done */
}

void main_thread()
{
   int magic = 0;
   /* Create a semaphore to use to signal when the magic value is available */
   value_ready = create_sem("value_ready", 0);

   thread_fork("child", child, 0, 0, 0);
   /* Block and wait for the grandchild to signal the value is ready */
   P(value_ready);

   magic = shared_value;                 /* Can now access the value */
   printf("Magic Value = %d\n", magic);
}
```

end solution

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.

begin solution

```
Deadlock is NOT possible.

Each thread always acquires and releases locks in the same
order from highest to lowest (first i+1 and then i).
As long as there is a consistent ordering to the locks and locks
are always acquired in the same order, there can be no deadlocks.
```

end solution

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.**

——————begin solution——————

A C D B
In Mesa-style monitors, the signaling thread continues execution until it clears the montor and releases the lock at which point the waiting thread executes. So:
T_1 calls proc1, prints A and waits to be signalled.
T_2 calls proc2, prints C signals T_1, continues executing and prints D it then leaves the monitor.
T_1 continues executing, prints B and then leaves the monitor.

——————end solution——————

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

——————begin solution——————

A C B D
In Hoare-style monitors, the signaling thread releases the lock so that waiting thread continues execution until it clears the montor and releases the lock. At which point the signalling thread can continue. So:
T_1 calls proc1, prints A and waits to be signalled.
T_2 calls proc2, prints C signals and T_1. At this point T_2 has release the lock to T_1.
T_1 continues executing and prints B, it then leaves the monitor, and releases the lock.
T_2 continues executing and prints D it then leaves the monitor.

——————end solution——————

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);
}
```

begin solution

```
/* MAX = ______ 1 ________ */
```

end solution

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]);
}
```

begin solution

```
                /* MAX = __ min(T,N)  _____ */  (If T < N)
```

end solution

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

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

begin solution

```
/* MAX = __ min(T,10)  _____ */
```

end solution

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

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

begin solution

```
/* MAX = _____ T ___________ */
```

end solution

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

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

begin solution

```
/* MAX = _____ 1 ___________ */
```

end solution

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

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

begin solution

```
/* MAX = _____ 1 __________ */
```

end solution

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

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

begin solution

```
/* MAX = _____ 1 __________ */
```

end solution

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.

————————begin solution————————

If the program is trying to read from a memory location and can not the program is unable to continue so I would just kill the program.

Determine the VPN of the faulting address VPN = BadVaddr / PageSize
If the page that is being read is available on disk.
Find a new unallocated frame, set up the page table and TLB so the VPN points to the new frame.
Copy the page from disk into the VPN.
If this generates a write fault try another frame if there is one available.
Keep track of bad pages.
Resume/continue execution of the faulting process, retries and hopefully continues.

————————end solution————————

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

————————begin solution————————

General idea is on a write failure, we try to remap the VPN to a different physical frame. This is done by first copying the original frame (accessed through the VPN) into the kernel (into tmppage), and then copying it to a new frame.

```
Determine the VPN of the faulting address    VPN = BadVaddr / PageSize
Copy that virtual page into kernel memory    copyin(ktmp_page, vpn, Pagesize)
If this fails with an exception kill the program because we can't
copy the page in order to continue executing.

look in the page table / TLB to figure out which page frame is bad
mark it is bad and don't use it anymore
```

Continued ...

```
while (1) {
   if (we can find a free phyiscal frame that isn't bad -> new_pfn) {
     found_page = 1;
     change mapping for VPN in the page table and TLB to new_pfn
     try to copy ktmp_page into the new_pfn    copyout(ktmp_page, vpn, PageSize)
     if (this fails) {
        mark new_pfn as bad and don't use it anymore
        continue  // go back to the top of the loop to look for another page
     } else {
        we've successfully copied the page from bad memory into good memory
        so we should break out of the loop and continue executing the program
        break;
     }
   } else {
     found_page = 0;
     break;
   }
}

if (!found_page) {
  kill the program because we can't find an available frame to map it to
} else {
  resume execution of the user program where it left off
}
```

end solution

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.

——————begin solution——————

2^{12} = 4096 so 12 bits for offset. 27-12 = 15 for VPN.
12 bits = 4 octal characters, 15 bits = 5 octal characters.
So the first 5 octal characters are the VPN and the last 4 are the offset

```
61252|1276.
There is no match for 61252 so no tranlation is possible and the
TLB generates a TLB exception.
```

——————end solution——————

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.

——————begin solution——————

2^{12} = 4096 so 12 bits for offset. 27-12 = 15 for VPN.
12 bits = 4 octal characters, 15 bits = 5 octal characters.
So the first 5 octal characters are the VPN and the last 12 are the offset

```
6125|2127.
There is a match for 6125 it is valid and it is writable so a tranlation is possible.
Replace the VPN with the PFN
50|2127.
Ensure that all 33-bits are used (i.e. 11 octal characters) and we have
000 0050 2127.
```

——————end solution——————

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.

begin solution

```
VPN = floor(virtual addr / pagesize) = 10963376 / 12304.
Could do:
 o long division = 891 remainder 512.
 o rough order of magnitude guestimate
   10,000,000 / 10,000 = 1000 (only close anwer is 891).
 o if want to avoid long division try multiplication (what * 12304 gives 10963376).

In the TLB VPN 891 has the valid bit set to 0 and not other entries
match the VPN so there is a TLB miss / exception generated.

Another reasonable but technically not quite correct answer.
The instruction is a store and all TLB entries have the Dirty bit
set to zero so this address will generate an write exception.
Technically, this wouldn't happen because the Valid bit would
need to be on for VPN 891 for this to happen.
```

end solution

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.

begin solution

1 MB page size = 2^{20} so 5 hex digits for offset.
40-20 = 20 bits (5 hex digits for virtual page number.

```
0x AC 135|0 AC13.
Lookup AC135 in TLB, match, is valid, and load means dirty bit doesn't matter.
Physical frame = 5D so
physical addr => 0x 00 0005D | 0AC13 (NOTE: 48-bit result).

NOTE: Someone correctly pointed out that since the VPN is 20 bits
the TLB could not contain the value 0xAC1350 -- because that is 24 bits.
```

end solution