**University of Waterloo**
**Midterm Examination**
**Term: Winter Year: 2012**

**Solution**

begin solution

Grade breakdown: Winter 2012

| Question | 1 | 2 | 3 | 4 | 5 | 6 | Total | |
|---|---|---|---|---|---|---|---|---|
| Avg | 5.3 | 11.1 | 9.7 | 15.2 | 6.5 | 8.4 | 56.1 | |
| Out of | 10.0 | 17.0 | 20.0 | 18.0 | 10.0 | 12.0 | 87.0 | |
| Maximum | 10.0 | 17.0 | 20.0 | 18.0 | 10.0 | 12.0 | 83.0 | (max grade not total) |
| Avg% | 52.8 | 65.4 | 48.3 | 84.3 | 64.7 | 69.7 | 67.5 | |

end solution

**Problem 1 (10 marks)**

a. **(2 mark(s))** Describe the different sources of overheads that are incurred when an application makes a BLOCKING system call in OS/161.

————————begin solution————————

Save user thread state on trap frame of calling thread.
Save the kernel state context of calling thread.
Flush the TLB.
Restore the kernel state context of new thread.
Restore the user thread state from the trap frame of the calling thread.

————————end solution————————

b. **(2 mark(s))** Explain what it means for a synchronization mechanism to be "starvation free".

————————begin solution————————

It means that every thread is guaranteed to eventually get access to the ***** critical section. ***** It DOES NOT mean that every thread will eventually get to run/execute/continue; a thread may get to run/execute/continue, try to get into the critical section and fail (and still starve).

————————end solution————————

c. **(2 mark(s))** Did you read the paper "An Introduction to Programming with Threads", or "An Introduction to Programming with C# Threads", by Andrew Birrell? If yes did you find the paper useful? Explain why or why not. If you did not read the paper, explain why you didn't.

————————begin solution————————

Some sensible answer. Basically 2 free marks.

This was asked to see if students are or are not reading the assigned required reading and to find out what those who did read the paper thought of it.

————————end solution————————

d. **(2 mark(s))** In whichever version of the Birrell paper you read what is the name of the function used to wait for another thread to finish?

————————begin solution————————

Join.

This is useful to know because if it is provided, it makes it easier for one thread to wait for the other threads it has created. This was done using P and V in the cat/mouse assignment and likely in solutions to fixing broken code in other parts of this exam. If `Join` is provided the thread can simply join with each created thread.

————————end solution————————

e. **(2 mark(s))** According to the Birrell paper you read, what are a spurious wake-ups and what is one example cause of spurious wake-ups?

————————begin solution————————

Spurious wake-ups occur when threads are waken that can not make progress.
In the Modula-2+ paper one example is if a Broadcast is used with a condition variable when a Signal should really be used instead.
In the C# paper this can happen if you use PulseAll when Pulse would have been sufficient.

————————end solution————————

**Problem 2 (17 marks)**
For the programs shown, fill in the blanks below each program to indicate how many characters of each letter will be printed in total when the program finishes running. If a range of values is possible, give the range. If it is not possible to determine the number or a range, state so and explain why. Assume that all function, library and system calls are successful. Use the space to the right of each program to draw a diagram of the process hierarchy that results during execution and to explain how you arrived at your answer. **NO MARKS WILL BE GIVEN UNLESS A PROPER DIAGRAM AND EXPLANATION ARE PROVIDED.**

a. **(9 mark(s))**

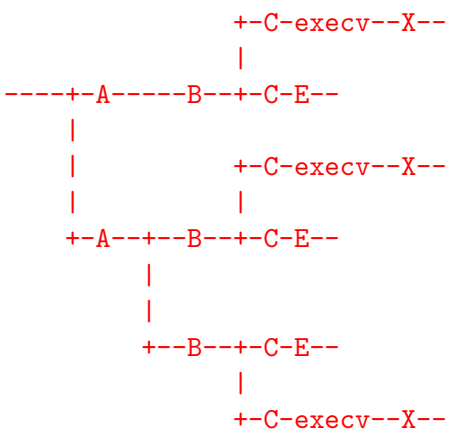
```
/* forkprint1.c : source code for the program forkprint1 */

int main(int argc, char *argv[])
{
  int rc;
  int status;

  char *args[3];
  args[0] = (char *) "pgm";
  args[1] = (char *) "X";
  args[2] = (char *) 0;

  rc = fork();
  printf("A");
  if (rc != 0) {
    rc = fork();
  }
  printf("B");

  rc = fork();
  printf("C");
  if (rc == 0) {
    /* exec the program shown below */
    rc = execv("pgm", args);
    printf("D");
  } else {
    waitpid(rc, &status, 0);
  }
  printf("E");
}
/*-------------------------------------------------------------------------------*/
/* pgm.c : source code for "pgm". Used above by forkprint1.c  */

int main(int argc, char *argv[])
{
  printf("%s", argv[1]);
}
```

Total number of printed A's _____, B's _____, C's _____, D's _____, E's _____, X's _____

begin solution

Total number of printed A's = 2, B's = 3, C's = 6 D's = 0, E's = 3, X's = 3

NOTE: D = 0 because execv doesn't return.




NOTE: If you compile and run these on a Linux/Unix machine you need to insert fflush(stdout); after every printf otherwise you can get strange output (e.g., more than 2 "A's" because the output may be buffered and copied at the time of a fork.

end solution

b. **(8 mark(s))**

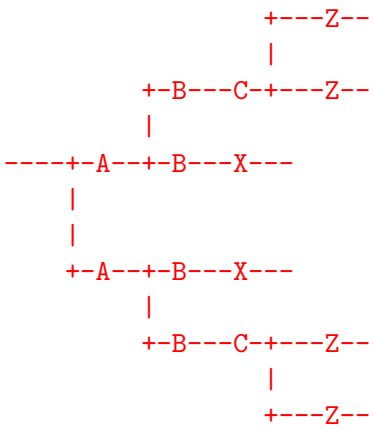
```
/* forkprint2.c : source code for the program forkprint2 */

char *str = "X";

main()
{
  int rc;

  rc = fork();
  printf("A");
  rc = fork();
  printf("B");
  if (rc == 0) {
    str = "Z";
    printf("C");
    rc = fork();
  }
  printf("%s", str);
}
```

Total number of printed A's _____, B's _____, C's _____, X's _____, Z's _____

begin solution

Total number of printed A's = 2, B's = 4, C's = 2, X's = 2, Z's = 4

NOTE: on fork the address space is copied so str = "Z" only affects the processes that execute that line (i.e., those two children).




end solution

**Problem 3 (20 marks)**

Below you will find some programs that may or may not be implemented correctly. Assume that the file `neededstuff.h` contains any definitions that are missing from the existing code (e.g., `T`, `DATA_SIZE`, etc.).

In each case below you are to look at the solution provided to the described problem and to determine whether or not the solution will always produce the correct result. If the solution will not produce the correct result add to and/or change the code (including necessary declarations and initialization code) so that it does produce the correct result. Use only those synchronization primitives available in OS/161 after assignment 1 has been completed. Do not disable and/or enable interrupts. If the solution is correct just write CORRECT on the solution. Assume that any function, library, or system calls always succeed.

a. **(8 mark(s))** This program should print the maximum value found in `array`.

```
#include "neededstuff.h"
unsigned int max = 0;
unsigned int array[DATA_SIZE];


main()
{
  int i = 0;

  for (i=0; i<T; i++) {

    thread_fork("findmax", findmax, UNUSED_PTR, i, NORETURN_VAL);

  }



  /* PRINT THE MAXIMUM VALUE FOUND IN ARRAY */
  printf("Maximum value = %u\n", max);
}

void findmax(void *unused, unsigned long threadnum)
{
  int i = 0; int size = DATA_SIZE/T; /* assume this divides evenly */
  int start = size * (int) threadnum;

  for (i=start; i<start+size; i++) {


     if (array[i] > max) {

         max = array[i];

     }


  }

}
```

begin solution

```
#include "neededstuff.h"
volatile unsigned int max = 0;                  /*** ADDED: volatile ***/
unsigned int array[DATA_SIZE];

struct lock *l;                                 /*** ADDED:          ***/
struct sem *s;                                  /*** ADDED:          ***/

main()
{
  int i = 0;

  l = lock_create("lock_max");                   /*** ADDED:          ***/
  s = sem_create("waiter", 0);                  /*** ADDED:          ***/
                                                /*** NOTE other possibilities exist ***/

  for (i=0; i<T; i++) {
    thread_fork("findmax", findmax, UNUSED_PTR, i, NORETURN_VAL);
  }

  /* Must wait for all of the threads to complete before printing */
  for (i=0; i<T; i++) {                         /*** ADDED:           ***/
    P(s);                                       /*** ADDED:           ***/
  }                                             /*** ADDED:           ***/

  /* PRINT THE MAXIMUM VALUE FOUND IN ARRAY */
  printf("Maximum value = %u\n", max);
}

void findmax(void *unused, unsigned long threadnum)
{
  int i = 0; int size = DATA_SIZE/T; /* assume this divides evenly */
  int start = size * (int) threadnum;

  for (i=start; i<start+size; i++) {

     lock_acquire(l);                           /*** ADDED: LOCK     ***/
     if (array[i] > max) {                      /*** MUST INCLUDE IF STMT ***/
         max = array[i];
     }
     lock_release(l);                           /*** ADDED:          ***/

  }
  V(s);                                         /*** ADDED: Signal thread is done ***/
}
```

end solution

b. **(12 mark(s))** This program should place values in an array (concurrently) and then compute the sum of those values (also concurrently). Once the sum has been computed it should be printed.

```
#include "neededstuff.h"
int array[DATA_SIZE];


main()
{
  int i = 0;
  int grand_total = 0;


  for (i=0; i<T; i++) {

    thread_fork("populate", populate, UNUSED_PTR, i, NORETURN_VAL);

  }



  for (i=0; i<T; i++) {

    thread_fork("dosum", dosum, UNUSED_PTR, i, NORETURN_VAL);

  }



  /* SHOULD PRINT SUM OF ALL ELEMENTS OF ARRAY */
  printf("Sum of all elements of array = %d\n", grand_total);

}

void populate(void *unused, unsigned long threadnum)
{
  int i = 0; int size = DATA_SIZE/T; /* assume this divides evenly */
  int start = size * (int) threadnum;


  for (i=start; i<start+size; i++) {


     array[i] = i;


  }


}
```

continued ...

```
void dosum(void *unused, unsigned long threadnum)
{
  int i = 0;
  int size = DATA_SIZE/T;       /* assume this divides evenly */
  int start = size * (int) threadnum;
  int sum = 0



  for (i=start; i<start+size; i++) {


     sum = sum + array[i];



  }



  grand_total = grand_total + sum;




}
```

begin solution

```
#include "neededstuff.h"
volatile int array[DATA_SIZE];                /*** ADDED volatile  ***/

struct lock *l;                               /*** ADDED:          ***/
struct sem *s;                                /*** ADDED:          ***/

volatile int grand_total = 0;                 /*** ADDED OR MOVED: MUST BE volatile ***/

main()
{
  int i = 0;
  /*** int grand_total = 0; ***/              /*** REMOVED OR MOVED TO GLOBAL ***/

  l = lock_create("lock_max");                 /*** ADDED:          ***/
  s = sem_create("waiter", 0);                 /*** ADDED:          ***/

  for (i=0; i<T; i++) {
    thread_fork("populate", populate, UNUSED_PTR, i, NORETURN_VAL);
  }

  /* Must wait for all of the threads to complete before starting dosum */
  for (i=0; i<T; i++) {                       /*** ADDED:           ***/
```

```
    P(s);                                       /*** ADDED:          ***/
  }                                             /*** ADDED:          ***/

  for (i=0; i<T; i++) {
    thread_fork("dosum", dosum, UNUSED_PTR, i, NORETURN_VAL);
  }

  /* Must wait for all of the threads to complete before printing */
  for (i=0; i<T; i++) {                         /*** ADDED:          ***/
    P(s);                                       /*** ADDED:          ***/
  }                                             /*** ADDED:          ***/

  /* SHOULD PRINT SUM OF ALL ELEMENTS OF ARRAY */
  printf("Sum of all elements of array = %d\n", grand_total);
}

void populate(void *unused, unsigned long threadnum)
{
  int i = 0; int size = DATA_SIZE/T; /* assume this divides evenly */
  int start = size * (int) threadnum;

  for (i=start; i<start+size; i++) {        /*** NO OVERLAPPING WRITES: NO SYNCH NEEDED ***/
     array[i] = i;
  }

  V(s);                                         /*** ADDED: Signal thread is done ***/
}

void dosum(void *unused, unsigned long threadnum)
{
  int i = 0;
  int size = DATA_SIZE/T;       /* assume this divides evenly */
  int start = size * (int) threadnum;
  int sum = 0

  for (i=start; i<start+size; i++) {            /*** NO CHANGES HERE ***/
     sum = sum + array[i];                      /*** sum is local, not shared ***/
  }

  lock_acquire(l);                              /*** ADDED: LOCK     ***/
    grand_total = grand_total + sum;
  lock_release(l);                              /*** ADDED:          ***/

  V(s);                                         /*** ADDED: Signal thread is done ***/
}
```

end solution

**Problem 4 (18 marks)**

Several different versions of a function to transfer funds from one bank account to another are shown below. Your manager wants you to evaluate each of them to see if any one of them could be used safely and correctly with multiple threads. When evaluating any one of them it would be the only one used and the others would not be called. Assume that all function, library and system calls succeed and that appropriate missing definitions and initializations are provided and/or called elsewhere. Your manager wants you to answer the following questions and to explain your answer:

- Can the use of the procedure result in a deadlock? Circle the answer and describe why or why not.
- Will the procedure transfer the specified amount between the two accounts correctly (aside from possible deadlock problems) and maintain correct balances? Circle the answer and describe why or why not.

**NOTE: several locks are declared and/or used differently in each of the different procedures so you need to carefully read the code to see which locks are being used.**

```
struct lock *from_lock;
struct lock *to_lock;

struct account {
  struct lock *lock;
  volatile int balance;        /* THIS WAS NOT BUT SHOULD HAVE BEEN DECLARED VOLATILE */
  int num;
};

init()
{
   from_lock = lock_create("from_lock");
   to_lock = lock_create("to_lock");
}
```

————————begin solution————————

Technically the correct answer is that none of the given procedures below is correct because `int balance` needed to be `volatile int balance`. But we were not intending to be that tricky in this question so also accepted the alternate solutions shown below (which are correct if we had used `volatile int balance`).

————————end solution————————

a. **(6 mark(s))**

```
void transfer1(struct account *from_account, struct account *to_account, int amount)
{
   lock_acquire(from_lock);
     lock_acquire(to_lock);
       from_account->balance -= amount;
       to_account->balance += amount;
     lock_release(to_lock);
   lock_release(from_lock);
}
```

Deadlock: YES NO

————————begin solution————————

Deadlock NO. The locks are always acquired in the order from_lock, to_lock which ensures that deadlock can not occur.
OR
The first lock ensures that there is atomic access to the rest of the code so only one thread is permitted to execute the function at a time so deadlock is not possible.

———————end solution———————

Correct: YES NO

———————begin solution———————

```
Correct YES. The use of the locks here ensures that the transfer is an atomic operation.
OR
Correct NO. int balance is not volatile
```

———————end solution———————

b. **(6 mark(s))**

```
void transfer2(stuct account *from_account, struct account *to_account, int amount)
{
   lock_acquire(from_account->lock);
     lock_acquire(to_account->lock);
       from_account->balance -= amount;
       to_account->balance += amount;
     lock_release(to_account->lock);
   lock_release(from_account->lock);
}
```

Deadlock: YES NO

begin solution

Deadlock YES. T1 transfer(a1,a2) concurrently with T2 transfer(a2,a1);
T1 lock_acquire(a1->lock), T2 lock_acquire(a2->lock),
T1 lock_acquire(a2->lock) [Now blocked waiting for a2->lock]
T2 lock_acquire(a1->lock) [Now blocked waiting for a1->lock]

end solution

Correct: YES NO

begin solution

Correct YES. The use of the locks here ensures that the transfer is an atomic operation.
OR
Correct NO. int balance is not volatile

end solution

c. **(6 mark(s))**

```
void transfer3(struct account *from_account, struct account *to_account, int amount)
{
   struct lock *from = create_lock("from");  /* typo in original was missing (*) ptr */
   struct lock *to = create_lock("to");      /* typo in original was missing (*) ptr */
   lock_acquire(from);                       /* announced correction during exam     */
     lock_acquire(to);
       from_account->balance -= amount;
       to_account->balance += amount;
     lock_release(to);
   lock_release(from);
   lock_destroy(from);
   lock_destroy(to);
}
```

Deadlock: YES NO

begin solution

Deadlock NO. Because the locks are local to transfer3 each thread
gets its own copy of the locks, the are not shared so they can't be
locked by any other thread and deadlock is not possible.

——————end solution——————

Correct: YES NO

——————begin solution——————

```
Correct NO. Because the locks are local to transfer3 each thread
gets its own copy of the locks, the are not shared so they can't be
locked by any other thread and there is no protection of the global variables.
OR
Correct NO. int balance is not volatile
```

——————end solution——————

**Problem 5 (10 marks)**

The structure `addrspace` shown below describes the address space of a running process on a 32-bit MIPS processor similar to that used in the assignments. The virtual page size is 4096 (`0x1000`) bytes. Assume that the operating system gives each process a stack segment consisting of 16 pages, starting at virtual address `0x7fff0000` and ending at virtual address `0x7fffffff`.

```
struct addrspace {
    vaddr_t as_vbase1 = 0x00900000;       /* text segment: virtual base address */
    paddr_t as_pbase1 = 0x10000000;       /* text segment: physical base address */
    size_t as_npages1 = 32;               /* text segment: number of pages in decimal */
    vaddr_t as_vbase2 = 0x30000000;       /* data segment: virtual base address */
    paddr_t as_pbase2 = 0x00400000;       /* data segment: physical base address */
    size_t as_npages2 = 512;              /* data segment: number of pages in decimal */
    paddr_t as_stackpbase = 0x80000000;   /* stack segment: physical base address */
};
```

For an application executing in user space that uses the address space defined above, assume that it is accessing the specified addresses below. When possible you are to translate the provided address. If the translation is not possible, explain why it is not possible and what would happen during translation. If the translation is possible indicate which segment the address belongs to. Use hexadecimal notation for all addresses.
Some possibly useful values:

```
  1 * 4096 =    0x1000     2 * 4096 =   0x2000    10 * 4096 =   0xA000
 16 * 4096 =   0x10000    32 * 4096 =  0x20000   100 * 4096 =  0x64000
128 * 4096 =   0x80000   256 * 4096 = 0x100000   512 * 4096 = 0x200000
```

a. **(2 mark(s))** Translate the **Virtual** Address `0x0090A8C0` to a **Physical** Address.

begin solution

`0x0090A8C0 - 0x0090 0000 = A8C0 + 0x1000000 = 0x1000A8C0: Part of the code segment`

end solution

b. **(2 mark(s))** Translate the **Virtual** Address `0x7FFFB495` to a **Physical** Address.

begin solution

`0x7FFFB495 - 0x7FFF0000 = B495 + 0x80000000 = 0x8000B495: Part of the stack segment`

end solution

c. **(2 mark(s))** Translate the **Physical** Address `0x00519BCF` to a **Virtual** Address.

begin solution

`0x0051 9BCF - 0x0040 0000 = 11 9BCF + 0x3000 0000 = 0x3011 9BCF: Part of the data segment`

end solution

d. **(2 mark(s))** Translate the **Physical** Address `0x8000100A` to a **Virtual** Address.

begin solution

0x8000100A - 0x8000 0000 = 0x100A which part of the stack segment. \\
So the tranlated virtual address is 0x7FFF0000 + 0x100A = 0x7FFF100A.

——————end solution——————

e. **(2 mark(s))** Translate the **Virtual** Address `0x8000100A` to a **Physical** Address.

——————begin solution——————

0x8000100A is part of the kernel's address space so translation can not \\
be performed while running a user application. An exception would be generated.

——————end solution——————

**Problem 6 (12 marks)**
Some possibly useful info: $2^{10}$ = 1 KB, $2^{20}$ = 1 MB, $2^{30}$ = 1 GB

In this 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 *33-bit* virtual addresses and a page size of 32768 bytes (32 KB). During a program's execution the TLB contains the following entries (all in octal).

| Virtual Page Num | Physical Frame Num | Valid | Dirty |
|---|---|---|---|
| 6125 | 1234567 | 1 | 0 |
| 61252 | 123456 | 0 | 0 |
| 612 | 3013 | 1 | 1 |
| 612521 | 765432 | 1 | 1 |

If possible, explain how the MMU will translate the virtual addresses given below (in octal) into a *36-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 36-bits**.

a. **(3 mark(s))** Load from virtual address = 61252127604.

———————begin solution———————

THIS DOES NOT NEED TO BE STATED OR REPEATED FOR EACH PART. $2^{15}$ = 32768 so 15 bits for offset. 33-15 = 18 for VPN.
15 bits = 5 octal characters for offset, 18 bits = 6 octal characters for VPN.
So the first 6 octal characters are the VPN and the last 5 are the offset

```
612521 | 27604 lookup 612521 in TLB valid bit is set so change virtual page
number to physical page number 612521 => 765432|27604
then make sure 36 bits = 0765432|27604 = 0765 4322 7604.
```

———————end solution———————

b. **(3 mark(s))** Store to virtual address = 61252127.

———————begin solution———————

```
612|52127.  There is a match in TLB for VPN = 612 it is valid and it is dirtyable (D=1)
so a tranlation is possible.  Replace the VPN with the PFN 3013|52127.
Ensure that all 36-bits are used (i.e. 12 octal characters) and we have 0003 0135 2127.
```

———————end solution———————

c. **(3 mark(s))** Store to virtual address = 612503714.

———————begin solution———————

```
6125|03714. There is a match for VPN = 6125 in TLB and it is valid
but it is not writable (D=0) so a tranlation is not possible.
An exception is generated because trying to write to a page that can not be dirtied.
```

———————end solution———————

d. **(3 mark(s))** Can a store be performed on the **physical** address = 12345671032? If yes, provide the virtual address used to access this physical address and if not explain precisely why not.

begin solution

123456|71032. There is a match for the frame number 123456. But the translation is not valid (V=0) and therefore we can't figure out what virtual address would correspond to the given physical address.

NOTE: stating we couldn't do a store to that page because the dirty bit is not set (D=0) is incorrect. Because the TLB entry is invalid we don't even know if it is referring to the same page (it may have been referring to a different address space (process)).

end solution