University of Waterloo
Midterm Examination
Term: Winter Year: 2015

Student Family Name ______________________________

Student Given Name ______________________________

Student ID Number ______________________________

Section : Circle one (Brecht 8:30) (Brecht 1:00)

Course Abbreviation and Number:	CS 350
Course Title:	Operating Systems
Section(s):	2
Instructors:	Tim Brecht

Date of Exam:	March 2, 2015
Time Period	Start time: 7:00 pm End time: 9:00 pm
Duration of Exam:	120 minutes
Number of Exam Pages:	9 (including cover sheet)
	NO CALCULATORS, NO ADDITIONAL MATERIAL

Problem	Topic	Marks	Score	Marker's Initials
1	Threads and thread_fork	9		
2	Processes and fork	14		
3	Context Switching	12		
4	Kernel Stacks	10		
5	Synchronization	12		
6	TLBs and Address Translation	9		
7	OS/161-"like" Address Translation	9		
Total		75		

Problem 1 (9 marks)

For the program shown below, assume that all function, library and system calls are successful. Recall that the prototype/signature for `thread_fork` is:

```
int thread_fork(const char *name, struct proc *proc,
                void (*func)(void *, unsigned long),
                void *data1, unsigned long data2);
```

```
volatile int x = 42;

main()
{
  /* name="1", no process, runs func1 */
  /* parameters 0 and 0, not used */
  thread_fork("1",NULL,func1,0,0);

  /* name="2", no process, runs func2 */
  /* parameters 0 and 0, not used */
  thread_fork("2",NULL,func2,0,0);

  func3(0,0);
}
```

```
void func1(unsigned long notused, void *notused2)
{
    kprintf("A: %d\n", x);
    x = 10;
}
void func2(unsigned long notused, void *notused2)
{
    kprintf("B: %d\n", x);
    x = 20;
}
void func3(unsigned long notused, void *notused2)
{
    kprintf("C: %d\n", x);
    x = 30;
}
```

When consdering each line of output produced by the program above, what would the output be when printing the value of the variable x? If more than one value or a range of values is possible, list all possible values or ranges.

```
/* From func1 */ A:


/* From func2 */ B:


/* From func3 */ C:
```

Problem 2 (14 marks)

For the program shown below, what output would be printed when it runs? If a range or multiple values are possible, give the range or possible values. If it is not possible to determine the value, posssible values or a range, state so and explain why. Assume that all function, library and system calls are successful. If more than one ordering of output is possible choose one of the possible orderings. Recall that `WEXITSTATUS(status)` just gets the exit code portion of the `status` variable.

```
int x = 42;

main()
{
  int rc, status;

  rc = fork();

  if (rc == 0) {
    func1();
    _exit(1);
  } else {
    rc = waitpid(rc, &status, 0);
    printf("R: %d", WEXITSTATUS(status));
    printf("M: %d\n", x);
    x = 100;
    printf("P: %d\n", x);
    _exit(2);
  }

  func2();
}
```

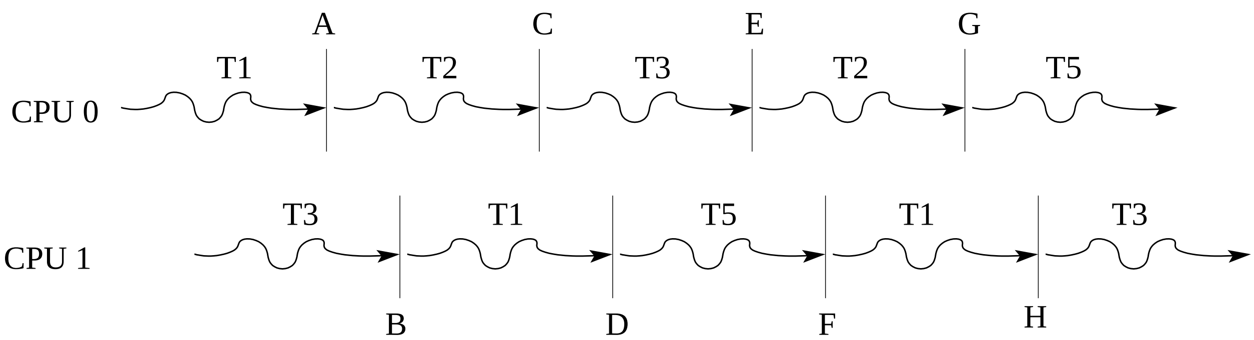
```
void func1()
{
    int rc, status;

    printf("T: %d\n", x);
    x = 10;
    rc = fork();
    if (rc == 0) {
      x = 50;
      printf("Q: %d\n", x);
      _exit(3);
    }
    rc = waitpid(rc, &status, 0)
    printf("A: %d\n", x);
    printf("D: %d", WEXITSTATUS(status));
    _exit(4);
}

void func2()
{
    printf("C: %d\n", x);
}
```

Problem 3 (12 marks)

The diagram below shows a number of threads executing on two different CPUs (the names of each thread are shown). The vertical lines indicate context switches between two threads and the labels at those vertical lines indicate the time at which the context switch occurred.



In OS/161 a context switch is initiated by a call to `thread_switch`. That function determines which thread to run next (pointed to by `next`) and calls `switchframe_switch` which performs the context switch from the current thread (pointed to by `cur`) to the next thread (pointed to by `next`). In the code below we have added two `kprintf` calls to print out the names of the `cur` and `next` thread before and after the context switch.

```
kprintf("Before: cur = %s next = %s\n", cur->t_name, next->t_name);

/* do the switch (in assembler in switch.S) */
switchframe_switch(&cur->t_context, &next->t_context);

kprintf("After: cur = %s next = %s\n", cur->t_name, next->t_name);
```

Using the diagram at the top of the page and the code above fill in the output that would be produced after each call to `switchframe_switch` (the output for the "Before" print statement has been provided). If it is not possible to determine the answer from the information provided in the diagram use the label "UN" (for unknown).

	Before		After	
Time	cur	next	cur	next
A	T1	T2		
E	T3	T2		
G	T2	T5		
H	T1	T3		

Problem 4 (10 marks)

Assume a **user-level process** (named P1) executes the code shown below on OS161.

```
main()          Q()                     R()                     S()
{               {                       {                       {  int i, x;
  Q();             int x = getpid();       int y = getpid();       for (i=0;i<N;i++) {
  R();                                                               x = x + i;
                   S();                                            }
}               }                       }                       }
```

In the rectangles shown for each part of this question below, fill in and label any information about the state of the **kernel stack** for the executing process (P1) as it would appear at the point in time stated in the question. Do not draw anything that has been popped from the stack (is no longer active) and use the same level of detail used in class and the course notes. If they are present, be sure to show trap frames, switch frames, and stack frames. (If you happen to remember some function names for stack frames use them but you don't have to name them or know how many there are just indicate where they are). Draw the stack so that the high addresses are at the top of the diagram and low addresses are at the bottom. Recall that the stack grows from high addresses to low.

a. **(5 mark(s))** The process P1 calls `int y = getpid()` in the funtion `R`. While in the kernel `syscall` function an interrupt occurs. Show what the contents of the stack look like just prior to returning from the interrupt handler and before popping the stack.

Kernel Stack

b. **(5 mark(s))** The process P1 is executing function `S`, an interrupt occurs, the kernel determines that the thread has reached its quantum of CPU time and it context switches to a thread of process P2. Show what the kernel stack looks like for P1 after the context switch to P2.

Kernel Stack

Problem 5 (12 marks)

Barrier synchronization can be used to prevent threads from proceeding until a specified number of threads have reached the barrier. Threads reaching the barrier block until the last of the specified number of threads has reached the barrier, at which point all threads can proceed. Below is a partial pseudocode example of how barrier synchronization might be used.

```
/* Used to wait for all mice to be ready to all attack together */
struct barrier *attack_barrier;
/* Used to wait for all mice and the main thread so they can all go to the bar together */
struct barrier *bar_barrier;

main()
{
  unsigned int i;
  attack_barrier = barrier_create(NUM_MICE);
  bar_barrier = barrier_create(NUM_MICE+1);

  for (i=0; i<NUM_MICE; i++) {
    thread_fork("MightyMouse", mouse, NULL, i);
  }

  /* Wait here until all threads are ready to go to the bar */
  barrier_wait(bar_barrier);
  go_to_bar();
}

void
mouse(void *unused, unsigned long mouse_num)
{
   unsigned int i;
   /* Attack the cats a number of times */
   for (i=0; i<ATTACK_COUNT; i++) {
     get_ammo(mouse_num);
     barrier_wait(attack_barrier); /* wait until all mice are ready to attack */
     attack_cats();
   }
   /* Wait here until all threads are ready to go to the bar */
   barrier_wait(bar_barrier);
   go_to_bar();
}
```

Fill in the spaces below (or on the next page) to complete the implementation of a barrier. (You will not implement `barrier_destroy`). You must only use `locks` and `condition variables` for synchronization (as they are defined in OS/161). To simplify the code, assume that all calls to `kmalloc` and to create any required objects always succeed.

```
struct barrier {









};
/* Create a barrier that can be used with thread_count threads */
struct barrier *barrier_create(unsigned int thread_count)
{









}
/* Callers wait here until the number of threads specified have */
/* reached this point, then they all proceed. */
void
barrier_wait(struct barrier *b)
{

















}
```

Problem 6 (9 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. Note: to make some numbers easier to read, spaces have been added between every 3 octal characters. Please also use this convention when providing your answers.

Consider a machine with *39-bit* virtual addresses, *33-bit* physical addresses and a page size of 262,144 bytes (256 KB). During a program's execution the TLB contains the following entries (all in **octal**). In this example **Dirty** means that the page can be dirtied (i.e., written to).

Virtual Page Num	Physical Frame Num	Valid Bit	Dirty Bit
0 061 252	06 125	1	0
6 125 273	01 234	0	0
0 000 061	30 130	1	1
0 000 612	61 252	1	1

If possible, explain how addresses given below (in **octal**) will be translated and provide the requested translated address. If a translation is not possible, explain what will happen and why. Show and **explain how you derived your answer.** Express the all physical address using **all 33-bits** and all virtual addresses using **all 39-bits**.

a. **(3 mark(s))** The physical address that results from a load from virtual address = 6 125 273 127 604

b. **(3 mark(s))** The physical address that results from a store to virtual address = 0 000 061 252 127.

c. **(3 mark(s))** Can a store be performed on the **physical** address = 61 252 612 522 If yes, provide the virtual address used to access this physical address and if not explain precisely why not.

Problem 7 (9 marks)

Note: to make some numbers easier to read, spaces have been added between every 4 hexidecimal characters. Please also use this convention when providing your answers.

The structure `addrspace` shown below describes the address space of a running process on a slightly modified MIPS processor. The `addrspace` and modified processor are similar to the `dumbvm` and MIPS processor provided in OS161/SYS161. The key differences are that this processor uses 36-bit virtual and physical addresses and a page size of 64 KB (`0x1 0000`). In a similar fashion to the 32-bit MIPS OS/161 processor the 36-bit virtual address space on this modified processor is divided into two halves. Virtual addresses from `0` to `0x7 FFFF FFFF` are for user programs and virtual address from `0x8 0000 0000` to `0xF FFFF FFFF` can not be accessed while in user mode. Fortunately, this new version of the OS161 kernel now explicitly represents the stack as segment 3 (note the stack size).

```
struct addrspace {
    vaddr_t as_vbase1 = 0x0 5000 0000;      /* text segment: virtual base address */
    paddr_t as_pbase1 = 0x0 0010 0000;      /* text segment: physical base address */
    size_t as_npages1 = 0x200;              /* text segment: number of pages */
    vaddr_t as_vbase2 = 0x3 0000 0000;      /* data segment: virtual base address */
    paddr_t as_pbase2 = 0x8 0000 0000;      /* data segment: physical base address */
    size_t as_npages2 = 0x137;              /* data segment: number of pages */
    vaddr_t as_vbase3 = 0x4 0000 0000;      /* stack segment: virtual base address */
    paddr_t as_pbase3 = 0x1 0000 0000;      /* stack segment: physical base address */
    size_t as_npages3 = 0x18;               /* stack segment: number of pages */
};
```

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 provide the requested translated address and indicate which segment the address belongs to. Use hexadecimal notation for all addresses and show all 36-bits. Show and explain how you arrived at your result.

Some possibly useful values:

```
 1 * 64 KB =  0x1 * 0x1 0000 =  0x1 0000    2 * 64 KB =  0x2 * 0x1 0000 =  0x2 0000
10 * 64 KB =  0xA * 0x1 0000 =  0xA 0000   16 * 64 KB = 0x10 * 0x1 0000 = 0x10 0000
32 * 64 KB = 0x20 * 0x1 0000 = 0x20 0000
```

a. **(3 mark(s))** Translate the **Virtual** Address `0x4 0017 6429` to a **Physical** Address.

b. **(3 mark(s))** Translate the **Virtual** Address `0x0 5200 AB25` to a **Physical** Address.

c. **(3 mark(s))** If possible, determine the user space **Virtual** Address that could be used to access the **Physical** Address `0x8 0128 95FA`.