CIS 520 - Final Exam – Fall 2010 Name: _____

Due Date: Wednesday, December 15, 2010, 1:40 pm (N122).

Short Answer (Problems 1-6: 15 points each, problem 7: 10 points):

1. Protection/Security Questions:

   a. If you ran the following program, test1.c, on a recent version of Linux:

```
// print address of argc
#include <stdio.h>
int main(int argc)
{
  printf("%p\n", &argc);
  return 0;
}
```

   Then you might see output such as the following:

```
$ gcc -o test1 test1.c
$ ./test1
0xbfac8d70
$ ./test1
0xbfcc0110
$ ./test1
0xbf86a050
```

   Which specific security-related technique explains this output?

   b. What type of attack are the Linux system designers trying to thwart or at least make more difficult by using this security-related technique?

   c. Is the following code segment vulnerable to a buffer-overflow attack? Explain briefly. If a vulnerability exists, how could it be eliminated?

```
char *f(void)
{
    static char buf[4];
    gets(buf);
    return(buf);
}
```

2. Assume you have a page reference string for a process running on a system with **m** physical pages available to the process. All of the process's pages are initially non-resident. The page reference string has length **p** with **n** distinct pages occurring in it. The process is running uninterrupted.

   a. For the LRU page replacement algorithm, suppose that the reference string is given by 1, 2, 3, … , **n**, 1, 2, 3, … , **n** , … (1 through **n** repeating to length **p**). If **m** = **n**, and **p** > **n**, how many page faults will occur? Explain briefly.

   b. For the same problem, but **m** = **n-1**, how many page faults will occur?

   c. For any reference string with **p** > **n** and any page replacement algorithm, what is a lower bound on the number of page faults? Explain briefly.

   d. For any reference string with **p** > **n** and any page replacement algorithm, what is an upper bound on the number of page faults? Explain briefly.

3. Suppose that a 32-bit virtual address is broken up into four fields, a, b, c, and d, to support a three-level page table.

   a. Explain the tradeoffs involved in deciding how many bits of the address should be assigned to each field. What type of information would you need about the expected workload on your system to make a good assignment?

   b. How many bits are used for each field in the 32-bit Intel Architecture (IA32)?

   c. The 64-bit architectures typically use five fields and four-level page tables. Why did AMD choose to not use the high-order sixteen bits and only use 48-bits for the five fields making up the virtual addresses in their AMD64 architecture; i.e., why not use all 64 bits?

4. Consider mapping a virtual memory of 1GB onto a physical memory organized into 256 page frames of 4096 byes each. Moreover, assume that the smallest addressable unit is 1 byte.

   a. Does the page table fit in main memory? Motivate your answer with a calculation.

   b. Does the frame table fit within a single page? Motivate your answer with a calculation.

   Given the same physical memory organization as above and a memory management strategy that always keeps at least 1 page frame per active process resident in main memory for paging purposes:

   c. What is the maximum size of the virtual address space given that any memory access may generate at most two page faults?

   d. Explain briefly how virtual addresses are mapped to the corresponding physical addresses.

   Furthermore, assume that there is hardware support in the form of a TLB (Translation Lookaside Buffer). Let p0, p1, and p2 be the probabilities that a memory access generates 0, 1, or 2 page faults, respectively. Let the time for a memory access without a page fault be t, the time for a memory access with a single page fault 4t, and the time for a memory access with two page faults 8t. Consider the following cases:

   Case 1: p0 = 3/4, p1= 3/16 and p2 = 1/16 with usage of the TLB.
   Case 2: p0 = 1, p1 = p2 = 0 without usage of the TLB.

   e. Determine the hit-ratio of the TLB when the two cases have the same performance. You may assume that there is no time penalty for accessing the TLB.

5. Process Model.

   a. Including the initial parent process, how many processes are created when the following program is executed?

   b. What will be printed?

   c. Draw the Process Model showing the parent-child hierarchy.

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    int i, retval;

    for (i=1; i<=3; i++)
    {
        retval = fork();  // fork off a child process
        if (retval == 0)
        {
          printf("In child %d\n", i);
          return 0;
        }
    }
    pause();  // wait for any signal to be received
}
```

   d. How many processes would be created if the "`return 0;`" statement was removed?

6. Given is the following set of jobs.

Job	Devices Allocated	Max R1 Required	Max R2 Required
A	0	2	3
B	0	1	2
C	0	3	3
D	0	2	2

a. Is deadlock possible if there are 6 type R1 resources and 8 type R2 resources? Explain briefly.

b. Suppose there are a total of 4 type R1 resources and 4 type R2 resources. Determine the situation if: A: req(R1, 2); A: acq(R1,2); B: req(R2, 2): B: acq(R2,2); i.e., A requests for and acquires 2 resources of type R1, and B requests for and acquires for 2 resources of type R2.

c. Starting from b, will the Banker's Algorithm grant the following subsequent requests? Explain your answer with a diagram.

   i. C: req(R1, 2)

   ii. D: req(R1, 2)

7. You have been given the task of synchronizing access to an array. Threads accessing the array are allowed to lock subsections of the array to increase the number of threads able to concurrently use the data structure.

   Your task is to design the control structures (locks and associated data structures) to make this time efficient (memory is cheap, so you don't have to worry about it as much, and if you lock a few extra elements, it's no big deal). E.g., you could use a bitmap like the previous problem to track which elements are currently locked, keep a list of ranges, or arrange the locks in a binary tree. The size of the array will be 2-3 million elements, is constant throughout the program run.

   Write the routines **`int getLockForRange(int first, int last),`** which returns an ID for the locked segment (locking the range within the array from first to last, inclusive) when appropriate entries is available, and **`bool releaseLock(int ID),`** which is called when the thread is done accessing that section of the array. Write the procedures using any combination of global variables, monitors, locks, semaphores, and condition variables as appropriate. You are not required to address starvation.