CSE 30321 – Computer Architecture I – Fall 2008
Homework 09 – Virtual Memory and I / O
Assigned: December 1, 2009 **– Due:** December 10, 2009
This assignment can be done in groups of 1, 2, or 3. The assignment is worth 60 points.

Problem 1: (40 points)

Background:
As discussed in class, virtual memory uses a page table and TLB to track the mapping of virtual addresses to physical addresses. This exercise shows how the TLB must be updated as addresses are accessed.

For this problem, you should assume the following:

1. Pages have 4 KB of addressable locations
2. There is a 4-entry, fully-associative TLB
3. The TLB uses a true, least-recently-used replacement policy

For the pattern of virtual addresses shown below, comment on whether the entry is:

a. Found in the TLB
b. Not found in the TLB but is found in the Page Table
c. Not found in the TLB or the Page Table (thus, there is a page fault)

The initial TLB and page table state is shown below.

If there is a page fault, and you need to replace a page from disk, assume the page number is one higher than the current highest page in the page table. (This is currently 1100_2 / 12_{10}. Thus, the next page would be 1101_2 / 13_{10}, the next would be 1110_2 / 14_{10}, etc.)

Stream of Virtual Addresses:

	MSB			LSB
1	0000	1111	1111	1111
2	0111	1010	0010	1000
3	0011	1101	1010	1101
4	0011	1010	1001	1000
5	0001	1100	0001	1001
6	0001	0000	0000	0000
7	0010	0010	1101	0000

Initial TLB State:
(Note that '1' = "Most Recently Used and '4' = "Least Recently Used")

Valid	LRU	Tag	Physical Page #
1	3	1011	1100
1	2	0111	0100
1	1	1001	0110
0	4	0000	-----

Initial Page Table State:

	Valid	Physical Page #
0000	1	0101
0001	0	Disk
0010	0	Disk
0011	1	0110
0100	1	1001
0101	1	1011
0110	0	Disk
0111	1	0100
1000	0	Disk
1001	0	Disk
1010	1	0011
1011	1	1100

Question A: (25 points)
Given the virtual addresses above, show how the state of the TLB changes by filling in the tables:

Address 1:

1	0000	1111	1111	1111

Valid	LRU	Tag	Physical Page #

This reference is a:

Address 2:

2	0111	1010	0010	1000

Valid	LRU	Tag	Physical Page #

This reference is a:

Address 3:

3	0011	1101	1010	1101

Valid	LRU	Tag	Physical Page #

This reference is a:

Address 4:

4	0011	1010	1001	1000

Valid	LRU	Tag	Physical Page #

This reference is a:

Address 5:

5	0001	1100	0001	1001

Valid	LRU	Tag	Physical Page #

This reference is a:

Address 6:

6	0001	0000	0000	0000

Valid	LRU	Tag	Physical Page #

This reference is a:

Address 7:

7	0010	0010	1101	0000

Valid	LRU	Tag	Physical Page #

This reference is a:

Question B: (10 points)
What would some of the advantages of having a larger page size be? What are some of the disadvantages?

Question C: (5 points)
Given the parameters in the table above, calculate the total page table size in a system running 5 applications.

Virtual Address Size	Page Size	Page Table Entry Size
64 bits	16 KB	8 bytes

Problem 2: (20 points)

Using a multi-core system to solve a problem is not simply a matter of moving the algorithm over from a single-threaded machine and splitting it into equal pieces. The multiple threads of execution must communicate and coordinate such that their intermediate results make sense relative to each other and when combined back into the whole.

You will learn much more about multi-threaded programming in your Operating Systems class next semester. For now, we are going to introduce a relatively simple problem called "resource contention" that becomes a problem when working on multi-core systems. Resource contention occurs when two processors try to access some hardware resource (say, an I/O device) concurrently when that device can only serve one processor at a given time. Usually, this is solved by "locking" the resource so that while one processor uses it, the other must simply wait and burn clock cycles until the resource is free.

Let us consider a machine that is a member of a high-performance supercomputer cluster working to find the Largest Prime Number[1] – the "LPN" program. The machine runs a tight loop that computes prime numbers in sequence and writes them to disk. Since the machine is built around a multi-core processor, the processors must be careful to access the disk one at a time.

Assume that each processor core produces prime numbers at a rate of 1 million per second. The disk runs fast enough to write 4 million primes per second to the output file. Thus, if we consider only the average data-flow rates given, we should use 4 cores to maximize prime number production.

Question A: (10 points)

Since the instantaneous data rate of the disk is faster than that of each core, the data must be buffered (stored up) by each core until it receives ownership of the disk, and then written in one burst. Assume that the core that currently owns the disk can tell the disk to start transferring from the buffer and then keep generating more primes. If the average generated prime is 512 bits long and the disk changes ownership once per millisecond, how much buffer memory is needed by each core?

Question B: (10 points)

Assume now that processor cores can communicate by sending an interrupt to another core. This will cause the interrupted core to jump to a special "interrupt handler" which can do whatever is necessary. (In the Intel architecture, this is called an IPI, or Inter-Processor Interrupt.)

In our application, a core passes the disk to the next core by sending an interrupt. Assume that it takes 1 ms for an IPI to be sent and received, and during that time, the receiving core cannot compute primes. We have 128 KB (2^{17} = 131,072 bytes) of buffer memory available per core.

What is the maximum rate of prime generation we can achieve with this system?

[1] (since no such number exists, this computer will run forever and so we can dispense with dealing with pesky edge-cases associated with program termination)