Flavio dos Santos – Ross
CSE 460 – Homework 5

1. (10 points)

   Consider a disk with the following characteristics:

Rotation speed	10,000 rpm
Number of surfaces	8
Sector size	512
Sectors per track	750
Tracks per surface	100,000
Storage capacity	307.2 billion bytes
Average seek time	4 ms
One-track seek time	0.2 ms
Maximum seek time	10 ms

a) What is the maximum transfer rate (bytes/sec) for a single track?

b) What is the maximum transfer rate for accessing five tracks consecutively?

a. maximum transfer rate = (sectors per track) * (sector size) * (rotation speed/sec)

rotation speed/sec = 10,000rpm / 60 = 166.666666667 rotations/sec

sectors per track = 750

sector size = 512

maximum transfer rate = (750) * (512) * (166.666666667)

= (384,000) * (166.666666667)

= 64,000,000 bytes / sec

b. maximum transfer rate for accessing five tracks consecutively =

(sector size) * (sectors per track) * (rotation speed / (60 * average seek time * 5)

= (512) * (750) * (10,000rpm / (60 + 4ms * 5)

= (512) * (750) * (10,000rpm / (60 + 0.02)

= (512) * (750) * (10,000rpm / (60.02)

= (512) * (750) * (166.611129623)

= 63,978,674 bytes / sec

Flavio dos Santos – Ross
CSE 460 – Homework 5

2. (10 points)
   Consider the following page-reference string:
   1,2,3,1,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,7
   How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, or seven frames? Remember that all frames are initially empty, so your first unique pages will all cost one fault each.
   - LRU replacement
   - FIFO replacement
   - Optimal replacement

LRU Replacement:

For Least-Recently-Used (LRU) Replacement, we get the following number of page faults:

Frames	Page Faults
1	21
2	19
3	15
4	12
5	8
7	7

FIFO Replacement:

For First-In First-Out (FIFO) Replacement, we would get the following number of page faults:

Frames	Page Faults
1	21
2	19
3	16
4	15
5	10
7	7

Flavio dos Santos – Ross
CSE 460 – Homework 5

<u>Optimal Replacement:</u>

For Optimal Replacement, we would have the following number of page faults:

Frames	Page Faults
1	21
2	15
3	11
4	9
5	7
7	7

3. (10 points)
Consider a file currently consisting of 200 blocks. Assume that the FCB (and the index block, in the case of indexed allocation) is already in memory. Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies if, for one block, the following conditions hold. In the contiguous-allocation case, assume that there is no room to grow in the beginning, but room to grow in the end. Assume that the block information to be added is stored in memory.

- The block is added at the the beginning.
- The block is added in the middle.
- The block is added at the end.
- The block is removed from the beginning.
- The block is removed from the middle.
- The block is removed from the end.

	Contiguous	Linked	Indexed
A) The block is added at the the beginning.	401	1	1
B) The block is added in the middle.	201	102	1
C) The block is added at the end.	1	2	1
D) The block is removed from the beginning.	0 or 398	1	0
E) The block is removed from the middle.	198	102	0
F) The block is removed from the end.	0	200	0

Flavio dos Santos – Ross
CSE 460 – Homework 5

Contiguous Allocation

Contiguous allocation is the simplest allocation scheme. The idea is to store each file as a contiguous run of disk space. For example, if a file has a size of 20-KB, we would allocate 20 consecutive blocks for that file.

Linked Allocation

Linked allocation is a method for storing files in which a file is kept as a linked list of disk blocks. The first word of each block is used as a pointer to the next one and the rest of the block is used for data.

Indexed Allocation

Indexed allocation is a method which associates with each file a data structure called an index-node which lists the attributes and disk addresses of the file's blocks. Given the index-node, it is possible to find all the blocks of the file.

a. The block is added at the the beginning.

Contiguous: All blocks must be shifted down as well as an extra operation to write and add the block. As a result, there are 401 I/O operations.

Linked: 1 write operation to add the block at the beginning.

Indexed: 1 write operation to add the block at the beginning.

b. The block is added in the middle.

Contiguous: Half of the blocks must be shifted down. This means one read and write operation for 100 blocks and one final write to add the block. So there are a total of 201 I/O operations.

Linked: There are 100 reads to reach the middle block. One write to add the block and one final write to fix the pointer. So there are 102 I/O operations.

Indexed: 1 write operation to add the block in the middle.

Flavio dos Santos – Ross
CSE 460 – Homework 5

c. The block is added at the end.

Contiguous: 1 write operation to add the block at the end.

Linked: 1 write operation to add the block at the end and 1 write operation to fix the last pointer. So there are 2 I/O operations.

Indexed: 1 write operation to add the block at the end.

d. The block is removed from the the beginning.

Contiguous: Only the FCB has to be updated which is in memory so there are 0 I/O operations. As an alternative, the last 199 blocks can be shifted down which requires 199 reads and writes for each block. So as a result, there can be 0 or 398 I/O operations.

Linked: The starting pointer has to be reset with the new first block so there is only 1 I/O operation.

Indexed: Only the index block has to be updated which is in memory so there are 0 I/O operations.

e. The block is removed from the middle.

Contiguous: Only the second half has to be shifted down to fill in the empty space left from the removed block. So there are 198 I/O operations.

Linked: The first 100 blocks must be read and the 100th block must be removed. Then the 99th block must write to the 101st block and the 101st pointer must be read. So there are 102 I/O operations.

Indexed: Only the index block has to be updated which is in memory so there are 0 I/O operations.

f. The block is removed from the end.

Contiguous: Only the FCB has to be updated which is in memory so there are 0 I/O operations.

Linked: The first 199 blocks must be read and the last pointer must be set to -1. So as a result, there are 200 I/O operations.

Indexed: Only the index block has to be updated which is in memory so there are 0 I/O operations.

Flavio dos Santos – Ross
CSE 460 – Homework 5

4. (10 points)
   Consider a system that has a 30-bit page-number and 128-byte pages. How large is the complete page table for such a virtual memory system?
   What are the tradeoffs involved in having smaller pages versus larger pages?

   - 30-bit page numbers,
   - The virtual memory system will have 2^30 entries.
   - Size of each page is 128-bytes

   We have:

   page table size = (entries) * (page size)

   = (2^30) * (128-bytes)

   = (1,000,000,000) * (128-bytes)

   = 128GB

   The size of the page table will be 128GB.

   For transferring data from disk to memory, it requires less time with larger pages than it does with multiple smaller pages to transfer the same amount of data.

   It seems to me like having larger page sizes is much more beneficial than having smaller page sizes.

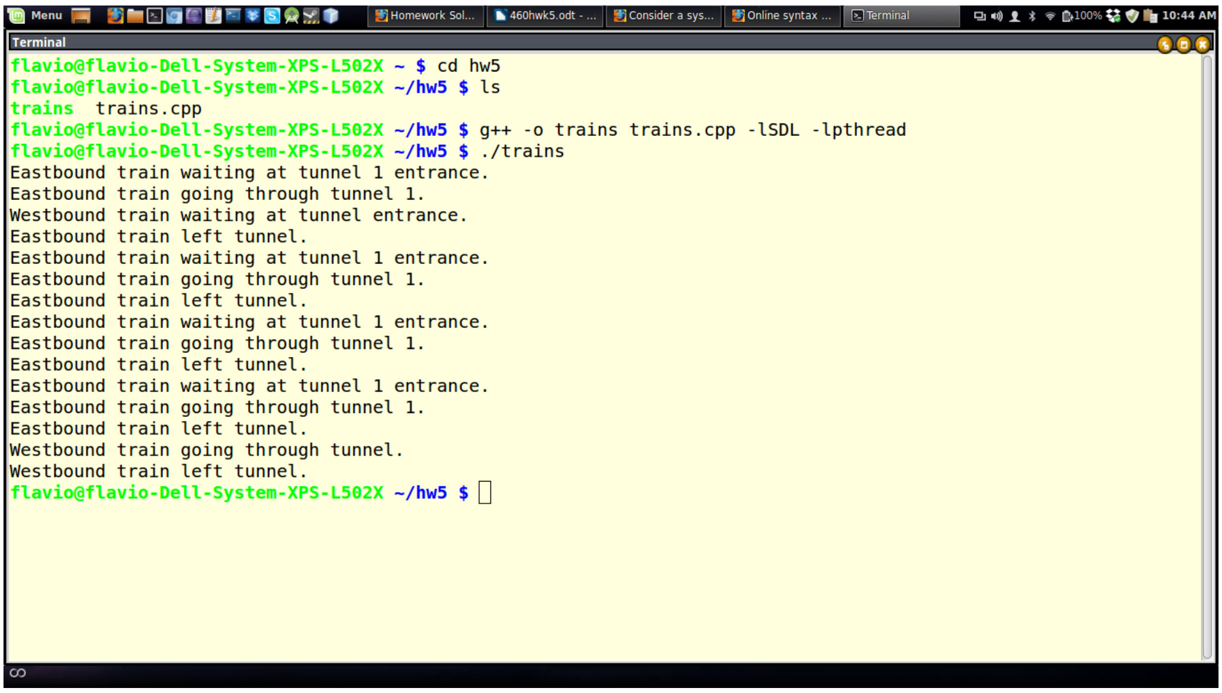
5. (20 points)
   A single-track railroad tunnel connects two villages. The railroad can become deadlocked if both a eastbound and westbound train enter the tunnel at the same time (the trains are unable to backup). Writing a program using POSIX or SDL threads that prevents deadlock using semaphores. Initially, do not concern about starvation (e.g. only eastbound trains are using the tunnel). Once your solution prevents deadlock, modify it so that it is starvation free.

```
/*
trains.cpp
Compile: g++ -o trains.cpp -ISDL -lpthread
Excute: ./trains
*/
#include <SDL/SDL.h>
#include <SDL/SDL_thread.h>
#include <cstdlib>
#include <stdio.h>


using namespace std;
```

```
SDL_sem* tunnel;

bool quit = false;

int train1(void *data)
{
        while (!quit)
        {
                puts("Eastbound train waiting at tunnel 1 entrance.");
                SDL_SemWait(tunnel);
                puts("Eastbound train going through tunnel 1.");
                SDL_Delay(5000);
                SDL_SemPost(tunnel);
                puts("Eastbound train left tunnel.");
        }

        return 0;
}

int train2(void *data)
{
        while (!quit)
        {
                puts("Westbound train waiting at tunnel entrance.");
                SDL_SemWait(tunnel);
                puts("Westbound train going through tunnel.");
                SDL_Delay(5000);
                SDL_SemPost(tunnel);
                puts("Westbound train left tunnel.");
        }

        return 0;
}

int main ()
{
        tunnel = SDL_CreateSemaphore(1);

        //Create the threads
        SDL_Thread* thread1 = SDL_CreateThread(train1, NULL);
        SDL_Thread* thread2 = SDL_CreateThread(train2, NULL);

        //Give the threads some time
        SDL_Delay(20 * 1000);

        // Signal the threads to exit
        quit = true;

        // Wait for the threads to exits
        SDL_WaitThread(thread1, NULL);
        SDL_WaitThread(thread2, NULL);

        return 0;
}
```

Program compiles and executes as shown:

Flavio dos Santos – Ross
CSE 460 – Homework 5



```
flavio@flavio-Dell-System-XPS-L502X ~ $ cd hw5
flavio@flavio-Dell-System-XPS-L502X ~/hw5 $ ls
trains  trains.cpp
flavio@flavio-Dell-System-XPS-L502X ~/hw5 $ g++ -o trains trains.cpp -lSDL -lpthread
flavio@flavio-Dell-System-XPS-L502X ~/hw5 $ ./trains
Eastbound train waiting at tunnel 1 entrance.
Eastbound train going through tunnel 1.
Westbound train waiting at tunnel entrance.
Eastbound train left tunnel.
Eastbound train waiting at tunnel 1 entrance.
Eastbound train going through tunnel 1.
Eastbound train left tunnel.
Eastbound train waiting at tunnel 1 entrance.
Eastbound train going through tunnel 1.
Eastbound train left tunnel.
Eastbound train waiting at tunnel 1 entrance.
Eastbound train going through tunnel 1.
Eastbound train left tunnel.
Westbound train going through tunnel.
Westbound train left tunnel.
flavio@flavio-Dell-System-XPS-L502X ~/hw5 $ 
```