CPSC 457 - Assignment 5

Due date: **Thursday, April 11, 2019 at 11:30pm**.
Individual assignment. No group work allowed.
Weight: 8% of the final grade.

Q1 – Written question (5 marks)

Assume the OS has already allocted memory partitions to 4 processes (P10, P11, P12 and P13) using a **dynamic partitioning** scheme. There are five free memory partitions (holes) of 100KB, 500KB, 200KB, 300KB and 600KB, as illustrated below:

free	P10	free	P11	free	P12	free	P13	free
100KB	10KB	500KB	20KB	200KB	30KB	300KB	40KB	600KB

The OS needs to allocate memory for 4 new processes in the following order: P1 of 212KB, P2 of 417KB, P3 of 112KB and P4 of 426KB. Draw the diagrams of the partitions after the OS has placed the processes using 4 different algorithms: first-fit, best-fit, worst-fit and next fit. The resulting diagrams must show the size of each partition, as well as the status of each partition, similar to the figure above. If a process cannot be placed, indicate that below the diagram. Start next-fit from the first partition.

Q2 – Written question (5 marks)

Consider a virtual memory system with a page size of 512 bytes, and a page table shown below. Convert the following logical addresses to physical addresses. Show the page numbers and page offsets for each logical address.

Logical address	Page number	Page offset	Physical address
1027			
2058			
522			
5			
2047			

	Page table
0:	3
1:	1
2:	0
3:	4
4:	1

Q3 – Written question (5 marks)

Consider a system with a 32-bit logical address space and 2KiB page size. The system supports up to 128MiB of physical memory. How many entries are there in each of the following?

a) A conventional single-level page table.
b) An inverted page table.

Show your calculations.

Q4 – Written question (5 marks)

Consider a system where a direct memory reference takes 150ns.

a) If we add a single-level page table stored in memory to this system, how much time would it take to locate and reference a page in memory?
b) If we also add a TLB, and 80% of all page-table references are found in the TLB, what is the effective access time ? Assume that searching TLB takes 20ns.

Show your calculations.

Q5 – Written question (5 marks)

Consider the following page reference string: 1,2,1,4,2,1,5,2,4,7,5,4,1,4,7,1,4,2,1,7. Assume there are 3 available frame, all initially empty. Illustrate how pages are placed into the frames using LRU and Optimal page replacement algorithms. How many page faults would occur for each algorithm? Format your answer in tables like the one below:

1	2	1	4	2	1	5	2	4	7	5	4	1	4	7	1	4	2	1	7
Number of page faults:																			

Q6 - Programming question (20 marks)

Write a program (`pagesim.c` or `pagesim.cpp`) that simulates three page replacement algorithms: Optimal, LRU and Clock. Your program will read in a reference string from standard input, and then run a simulation using all three algorithms. The number of available frames will be specified on the command line, and your simulation will start with all frames empty. For the clock algorithm you can use the single refererence bit implementation.

At the end of the simulation your program will output the content of the frames and the number of page faults for each placement algorithm. Your must format your output to match the sample output below:

Example input file test1.txt:	Sample output:

1 2 3 4 1 2 5 1 2 3 4 5	**$ ./pagesim 4 < test1.txt**
	Optimal:
	  - frames: 4 2 3 5
	  - page faults: 6
	LRU:
	  - frames: 5 2 4 3
	  - page faults: 8
	Clock:
	  - frames: 4 5 2 3
	  - page faults: 10

You can make the following assumptions:

- Number of available frames will be between 1 and 20 (inclusive).
- Number of entries in the reference string will be at most 5000.
- Frame numbers will be non-negative integers smaller than 100.

Q7 - Programming question (30 marks)

For this question you will implement a program (`fat.c` or `fat.cpp`) that will check the consistency of a file allocattion table with respect to the entries of a directory. Your program will read input from standard input, and will output results to standard output.

Input

The input will contain a simplistic representation of the filesystem. It will contain the following, all separated by white space:

- block size – an integer in range [1, 1024]
- number of entries in the directory – an integer in range [0, 50]
- number of entries in FAT – an integer in range [1, 200000]
- the entries in the directory – one entry per line, each containg:
    - filename – a string of up to 128 characters, any chars allowed except white space
    - first block pointer – an index into the FAT, an integer in range [-1, 200000), where '-1' denotes a NULL pointer
    - actual files size in bytes – an integer in range [0, 2^{30}]
- the entries in the FAT – a list of integers separated by white space
    - each entry represents a pointer to the next entry in the FAT
    - each entry is an integer in a range [-1, 200000)
    - -1 denotes a NULL pointer (end of chain)
    - the entries in FAT are numbered starting from 0

Sample input file `test1.txt`:

```
10 3 11
A.jpg 0 31
B.txt 6 23
C.zip -1 0
5 9 5 3 -1 1 8 0 6 -1 0
```

The above input describes a filesystem that has a block size of 10, contains FAT with 11 entries, and holds 3 files: `A.jpg`, `B.txt` and `C.zip`.

File `A.jpg` contains 31 bytes, and it is stored in blocks {0, 5, 1, 9}. It has the correct number of blocks, contains no cycles, and does not share blocks with any other file.

File `B.txt` has 23 bytes, and it is stored on blocks {6, 8}. The blocks belonging to file `B.txt` form a cycle, which is a problem that your program will need to detect. File `B.txt` also has an incorrect number of blocks for its size, which is another problem your program needs to report.

File `B.txt` does not share blocks with any other file. If it did, you would need to detect and report that as well.

File `C.zip` is empty. No blocks are allocated to this file. There are no problems with this file.

Finally, the total number of unused blocks on the filesystem is 5. This is a number you will need to calculate and report.

Output

After reading in the input, your program will check the consistency of the filesystem and report its findings to standard output. For every file you need to determine whether there are any issues with that file. You need to check for the following issues:

- Does the file contain the right number of blocks, or are there too many or too few?
- Do the blocks allocated to the file contain a cycle?
- Does the file share its blocks with any other file?

You then need to report all issues you found for every file.

You will also determine how many of the blocks are unused in the filesystem. Unused blocks are the ones not allocated to any file.

Sample output:

Here is an output that your program should produce for the above sample input:

```
$ ./fat < test1.txt
Issues with files:
  A.jpg:
  B.txt: not enough blocks, contains cycle
  C.zip:
Number of free blocks: 5
```

A skeleton code you can use as a starting point for this question is included in Appendix A.

Submission

You should submit 3 files for this assignment:

- Answers to the written questions combined into a single file, called either `report.txt` or `report.pdf`. Do not use any other file formats!
- Your solution to Q6 called `pagesim.c` or `pagesim.cpp`.
- Your solution to Q7 called `fat.c` or `fat.cpp`.

Since D2L will be configured to accept only a single file, you will need to submit an archive, eg. `assignment5.tgz`. To create such an archive, you could use a command similar to this:

```
$ tar zcvf assignment5.tgz report.pdf pagesim.cpp fat.cpp
```

You can use ZIP, TAR or TGZ archives. Do not submit any other types of archives!

General information about all assignments:

1. Unless stated otherwise, all programming questions will be marked with a 10s time limit per input. If your solution takes more than 10s per input, it will be marked incorrect.
2. All assignments must be submitted before the due date listed on the assignment. Late assignments or components of assignments will not be accepted for marking without approval for an extension beforehand. What you have submitted in D2L as of the due date is what will be marked.
3. Extensions may be granted for reasonable cases, but only by the course instructor, and only with the receipt of the appropriate documentation (e.g. a doctor's note). Typical examples of reasonable cases for an extension include: illness or a death in the family. Cases where extensions will not be granted include situations that are typical of student life, such as having multiple due dates, work commitments, etc. Forgetting to hand in your assignment on time is not a valid reason for getting an extension.
4. After you submit your work to D2L, make sure that you check the content of your submission. It's your responsibility to do this, so make sure that you submit your assignment with enough time before it is due so that you can double-check your upload, and possibly re-upload the assignment.
5. All assignments should include contact information, including full name, student ID and tutorial section, at the very top of each file submitted.
6. Assignments must reflect individual work. Group work is not allowed in this class nor can you copy the work of others. For further information on plagiarism, cheating and other academic misconduct, check the information at this link: http://www.ucalgary.ca/pubs/calendar/current/k-5.html.
7. You can and should submit many times before the due date. D2L will simply overwrite previous submissions with newer ones. It's better to submit incomplete work for a chance of getting partial marks, than not to submit anything.
8. Only one file can be submitted per assignment. If you need to submit multiple files, you can put them into a single container. The container types supported will be ZIP and TAR. No other formats will be accepted.
9. Assignments will be marked by your TAs. If you have questions about assignment marking, contact your TA first. If you still have questions after you have talked to your TA then you can contact your instructor.

Appendix 1 – Skeleton for Q7

The following C++ skeleton program `fat.cpp` is provided for you to make parsing and output easier. All you need to do is to reimplement the function `checkConsistency()`.

```
// CPSC457 University of Calgary
// Skeleton C++11 program for Q7 of Assignment 5.
//
// The program reads in the input, then calls the (wrongly implemented) checkConsistency()
// function, and finally formats the output.
//
// You only need to reimplement the checkConsistency() function.
//
// Author: Pavol Federl (pfederl@ucalgary.ca or federl@gmail.com)
// Date: April 1, 2019
// Version: 5

#include <stdio.h>
#include <string>
#include <vector>

typedef std::string SS;
typedef std::vector<SS> VS;

struct DEntry {
    SS fname = SS( 4096, 0);
    int size = 0;
    int ind = 0;
    bool tooManyBlocks = true;
    bool tooFewBlocks = false;
    bool hasCycle = true;
    bool sharesBlocks = true;
};

static SS join( const VS & toks, const SS & sep) {
    SS res;
    bool first = true;
    for( auto & t : toks) { res += (first ? "" : sep) + t; first = false;}
    return res;
}

// re-implement this function
//
// Parameters:
//   blockSize - contains block size as read in from input
//   files - array containing the entries as read in from input
//   fat - array representing the FAT as read in from input
// Return value:
//   the function should return the number of free blocks
//   also, for ever entry in the files[] array, you need to set the appropriate flags:
//      i.e. tooManyBlocks, tooFewBlocks, hasCycle and sharesBlocks
int checkConsistency( int blockSize, std::vector<DEntry> & files, std::vector<int> & fat)
{
    // make the first entry contain no errors
    if( files.size() > 0) {
        files[0].hasCycle = false;
        files[0].tooFewBlocks = false;
        files[0].tooManyBlocks = false;
        files[0].sharesBlocks = false;
    }
```

```
    // make the 2nd entry contain one error
    if( files.size() > 1) {
        files[1].hasCycle = true;
        files[1].tooFewBlocks = false;
        files[1].tooManyBlocks = false;
        files[1].sharesBlocks = false;
    }

    // make the 3rd entry contain two errors
    if( files.size() > 2) {
        files[2].hasCycle = false;
        files[2].tooFewBlocks = false;
        files[2].tooManyBlocks = true;
        files[2].sharesBlocks = true;
    }

    // finally, return the number of free blocks
    return 0;
}

int main()
{
    try {
        // read in blockSize, nFiles, fatSize
        int blockSize, nFiles, fatSize;
        if( 3 != scanf( "%d %d %d", & blockSize, & nFiles, & fatSize))
            throw "cannot read blockSize, nFiles and fatSize";
        if( blockSize < 1 || blockSize > 1024) throw "bad block size";
        if( nFiles < 0 || nFiles > 50) throw "bad number of files";
        if( fatSize < 1 || fatSize > 200000) throw "bad FAT size";
        // read in the entries
        std::vector<DEntry> entries;
        for( int i = 0 ; i < nFiles ; i ++ ) {
            DEntry e;
            if( 3 != scanf( "%s %d %d", (char *) e.fname.c_str(), & e.ind, & e.size))
                throw "bad file entry";
            e.fname = e.fname.c_str();
            if( e.fname.size() < 1 || e.fname.size() > 16)
                throw "bad filename in file entry";
            if( e.ind < -1 || e.ind >= fatSize) throw "bad first block in fille entry";
            if( e.size < 0 || e.size > 1073741824) throw "bad file size in file entry";
            entries.push_back( e);
        }
        // read in the FAT
        std::vector<int> fat( fatSize);
        for( int i = 0 ; i < fatSize ; i ++ ) {
            if( 1 != scanf( "%d", & fat[i])) throw "could not read FAT entry";
            if( fat[i] < -1 || fat[i] >= fatSize) throw "bad FAT entry";
        }

        // run the consistency check
        int nFreeBlocks = checkConsistency( blockSize, entries, fat);

        // format the output
        size_t maxflen = 0;
        for( auto & e : entries ) maxflen = std::max( maxflen, e.fname.size());
        SS fmt = "  %" + std::to_string( maxflen) + "s: %s\n";

        printf( "Issues with files:\n");
        for( auto & e : entries ) {
            VS issues;
```

```
            if( e.tooFewBlocks) issues.push_back( "not enough blocks");
            if( e.tooManyBlocks) issues.push_back( "too many blocks");
            if( e.hasCycle) issues.push_back( "contains cycle");
            if( e.sharesBlocks) issues.push_back( "shares blocks");
            printf( fmt.c_str(), e.fname.c_str(), join( issues, ", ").c_str());
        }
        printf( "Number of free blocks: %d\n", nFreeBlocks);
    }
    catch( const char * err) {
        fprintf( stderr, "Error: %s\n", err);
    }
    catch( ... ) {
        fprintf( stderr, "Errro: unknown.\n");
    }
    return 0;
}
```