Assignment 01 (80 points)

Due: Beginning of class, **09/10/2024**
Late Due: Beginning of class, **9/12/2024**
You must work **on your own**.

The Unix™ operating system is mostly written in C, as is true for other modern operating systems including Windows, etc. As a major part of your learning activities for this course, you are required to use C or C++ for doing the programming assignments. It is assumed that you had some form of training in C or C++ programming prior to this class. Should that not be the case, you are recommended to seriously reconsider your enrollment.

This first assignment is designed for you to reacquaint yourself with C/C++ programming, particularly around the pointer use, number representation and arithmetic, as well as the tree data structure, which is often used in representing OS entities, e.g., the organization of a file system, Unix process tree, multi-level paging table for memory management, multi-level indexed file allocation table, etc.

Task

Operating systems manage memory space in pages or blocks, this is done for many reasons that we will discuss in the memory management topic later in the class. For this assignment, you are asked to implement a **tree data structure to store the memory page (block) information** and use it to track the memory page access stats in simulating memory accesses from a memory trace file. Assume a **32-bit** system, each memory address has 32 bits.

Your **code** must **satisfy the requirements** specified for this assignment, see **coding specifications** below.

Functionality

Upon start, your program creates an empty page table (only the level 0 / root node should be allocated). The program should **read addresses one at a time** from a memory trace file for simulating a series of access attempts to memory addresses.

For **each address read in**:

- extract the memory page number based on the numbers of bits for the page levels from command line arguments,
- store (insert) the page number along a page tree path,
- update the number of accesses to the page at the leaf node level, and
- print to the standard output one address per line: memory address, its page indices along page levels, number of accesses to the page so far.

Compilation

1) You must create and submit a **Makefile** for compiling your source code. Makefiles are program compilation specifications. See **appendix** below (also refer to the Programming

reference page in Canvas) for details on how to create a makefile. A makefile example is also provided to you.

2) The make file must create the executable with a name as **pagetrace**; otherwise, auto-grading will automatically fail.

Program Inputs

Command line arguments

The executable **uses two mandatory command line arguments:**

- The first mandatory argument is the **name of the trace file** consisting of memory reference traces for simulating a series of attempts of accessing addresses.
    - **trace.tr** from the prompt is given for your testing.
    - Auto-grading on Gradescope will use **all** or **part** of **trace.tr**.
    - Appropriate error handling should be present if the file is not existent or cannot be opened. It must print to standard error output (stderr) or standard output the following error messages:
      Unable to open *<<trace.tr>>*
- The second mandatory argument is a **string** that specifies the **numbers of bits of page table levels**. For example, "4 8 8" means to construct a 3-level page table with 4 bits for level 0, 8 bits for level 1, and 8 bits for level 2.
    - With a **32-bit** system, the total number of bits from all levels should be less than **32**. In this assignment, you do NOT need to worry about checking the argument on this constraint.
- See **appendix** below for how to process command line arguments in C/C++.

Memory trace file

The traces were collected from a Pentium II running Windows 2000 and are courtesy of the Brigham Young University Trace Distribution Center. The files ***tracereader.h*** and ***tracereader.c*** (for C++, change the file name to ***tracereader.cpp***) implement a small program to read and print address trace files. You can include these files in your compilation and use the functionality to read the trace file. The file trace.tr is a sample of the trace of an executing process. See **appendix** below for an **example of reading trace** file.

Program Outputs (required)

Use functions from the given **log.h** and **log.c** (for C++, change log.c to log.cpp) for printing program output. **Important**: Do **NOT** implement your own output functions, autograding is strictly dependent on the output formats from the provided output functions.

Before reading addresses, print to the standard output:

- the bitmasks for each level starting with the lowest tree level (root node is at level 0), one level per line.
- use the given **log_bitmasks** function from log.c.

For each address read in, print to the standard output one address per line:

- **memory address, its page indices along page levels, number of accesses to the page so far**,
- use the given **log_pgindices_numofaccesses** function from log.c.

Sample Invocation

Note these samples not necessarily will be used for the autograding test cases

```
./pagetrace trace.tr “4 8 8”
```

Constructs a 3-level page table with 4 bits for level 0, 8 bits for level 1, and 8 bits for level 2. The remaining 12 bits would be for the offset in each page.

Processes addresses from the trace file and prints the output as specified above.

See the **expected** output in **trace_4_8_8_output.txt**.

```
./pagetrace trace.tr “7 15”
```

Constructs a 2-level page table with 7 bits for level 0, and 15 bits for level 1. The remaining 10 bits would be for the offset in each page.

Processes addresses from the trace file and prints the output as specified above.

See the **expected** output in **trace_7_15_output.txt**.

Before getting into code specifications for the assignment, let’s first give a **background** on how memory address information is represented and managed by the operating system.

Representation of memory addresses

OS views the memory space as an array of bytes. Assuming byte addressable, the array index of each byte represents a location of the memory, i.e., the memory address. A memory address is basically a number starting from 0.

For example, 0x8A2E6745 is a 32-bit address in hexadecimal form (8 bytes). In binary form, it would be 0b10001010001011100110011101000101. This address has an offset of 0x8A2E6745 from the 0th memory address location.

Manage memory space in pages (or blocks)

For various reasons, OS manages the memory allocation in blocks. A memory block is conventionally called a **memory page**. With a particular page size, a memory address can be decomposed into the combination of a memory page number and an offset into the memory page the address belongs to:

- **memory address** = **page number** * page size + **offset into the page**.
- **page number** = memory address **/** page size
- **offset into the page** = memory address **mod** page size

For example, with a page size of 4K (2^12) bytes, memory address **0x8A2E6745** can be expressed by:

- **page number** = memory address **/** page size = 0x8A2E6745 / 2^12
  = 0x8A2E6745 / 0x1000 = **0x8A2E6**
  - Using bitwise operations, this operation can be accomplished by bit masking and right shifting (0x8A2E6745 & 0xFFFFF000) >> 12, or just 0x8A2E6745 >> 12
- **offset into the page** = memory address **mod** page size
  = 0x8A2E6745 **mod** 0x1000 = **0x745**
  - Using bitwise operations, this operation can be accomplished by bit masking 0x8A2E6745 & 0x00000FFF
- This translates to the **0x8A2E6745** address is in the **0x8A2E6th** (zero-based) page and at the **0x745th offset** into that page.
- With the page size as 2^12, offset would use the bottom 12 bits, and page number would use the top 20 bits.

Store the memory page number hierarchically into a tree

In the memory management discussion later for the class, we will investigate how to efficiently (or save space) store the memory page number information.

One strategy is to store the memory page number in a tree along multiple levels, you will write a **basic implementation** of it in this assignment.

Let's use an example to demonstrate the idea.

Continuing from the example above, with a page size of 2^12 (4K) bytes, the page number would be encoded in the top 20 bits of the address. Suppose we would like to store this 20-bit number to a multi-level page tree, the number of tree levels and the number of bits for the levels could have many possibilities:

- split the 20 bits along a 3-levels page tree with 4 bits for level 0, 8 bits for level 1, and 8 bits for level 2, or
- split the 20 bits to 2 levels with 8 bits for level 0 and 12 bits for level 1, or
- split the 20 bits to 4 levels with 4 bits for level 0, 6 bits for level 1, 5 bits for level 2, and 5 bits for level 3,
- and so on and so forth.

We can use **bit masking and shifting** to extract page numbers (indices) from the address for insertion along the tree levels.

Again, using address 0x8A2E6745 as an example, with page size 2^12, page number would be 0x8A2E6. Suppose we want to store this page number to a 3-levels page table tree with number of bits for levels as **4 8 8**, to extract the page numbers (indices) for the 3 levels:

- First construct Level **bit masks**
  Level 0 mask: 0xF0000000
  Level 1 mask: 0x0FF00000
  Level 2 mask: 0x000FF000

- Next, calculate number of right shifting to get the page number for each level:
  Level 0, total number of bits (32) – number of bits of level 0 = 28
  Level 1, total number of bits (32) – number of bits of level 0 and 1 = 20

Level 2, total number of bits (32) – number of bits of level 0 and 1 and 2 = 12

- Apply bitwise AND operation using mask, then right shifting:
  Level 0 page number → 0x8A2E6745 & 0xF0000000 >> 28 → 0x8
  Level 1 page number → 0x8A2E6745 & 0x0FF00000 >> 20 → 0xA2
  Level 2 page number → 0x8A2E6745 & 0x000FF000 >> 12 → 0xE6

See **pagetable.pdf** diagrams for demonstrations of how the address page number is stored to a multi-level tree.

Coding - Tree specifications for storing page information

Please pay attention to the **implementation requirements** below for the page table structures and operation.

Page table structures

- See **pagetable.pdf** for **a sample data structure for N-level page tables**
- **PageTable** – A descriptor containing the attributes of a N level page table and a pointer to the root level (Level 0) object.
  - PageTable stores the multi-level paging information which is used for each Level or tree node object: number of levels, the bit mask and shift information for extracting the part of page number pertaining to each level, number of entries to the next level objects, etc.
  - Since the tree operations start from root node, it would be convenient to have the PageTable have a reference / pointer to the root node (Level) of the page tree.

- **Level** (or PageNode) – An entry for an arbitrary level, this is the structure (or class in c++) which represents a node of one of the levels in the page tree/table.
  - Level is essentially the structure for the **multi-level page tree NODE**. Multi-level paging is about splitting and storing the page number information into a tree data structure along the tree paths. Starting from the root node, each tree path (the root node to a leaf node) stores a page number.
  - Level (or Node) contains **an array of pointers (entries)** to the next level or page.
    - Implementation **requirements**:
      - You **must** implement this **Level structure as a tree** and use the **double Level **** pointer for storing the array of next Level* pointers.
        - Do **NOT** use any **collection** data structure from the STL (standard template library), such as vector, list, etc.
      - **Violating the above requirement would result in a zero point** for **a1** assignment.
  - You may add other data members to your node structure as you see needed, such as node depth / level, number of how many times a node has been visited, etc.
  - Programming tips:
    - Sample code for instantiating a double pointer array,

        - nextLevelPtr = new Level*[numOfEntries]; // C++
        - nextLevelPtr = (struct Level**) malloc(numOfEntries*sizeof(struct Level*)); // C
    - As a best practice, after instantiation, always **explicitly initialize** all entries in the array to NULL or nullptr. And in C and C++, it is always a best practice to explicitly initialize all variables before using them.
        - The "**run in one environment but not in another**" problem is often caused by non or improper initializations.

Page table mandatory interfaces

Implementation **requirements** (violation would incur **50% penalty** of **autograding**): You **must** implement **similar** functions for multi-level paging as proposed below. Your exact function signatures may vary, but the functionality should be the same.

All other interfaces may be developed as you see fit.

unsigned int **recordPageAccess**(unsigned int address)

- Record the access to the page of the given address by traversing the corresponding page path in the tree
    - insert entries to the page table tree if needed,
        - note when inserting a page, **do not re-create and add the level / node if the page entry already exists** at the level, just continue to the next level, and
        - only **create the next level nodes** as needed based on the page being inserted, **do not create** all nodes at the beginning for all levels
    - increment the number of accesses to the page at the leaf node level and return it after reaching the leaf node.
    - You could use a field numOfAccesses in Level class for tracking the number of accesses to the page / node.

unsigned int **extractPageNumberFromAddress**(unsigned int address, unsigned int mask, unsigned int shift)

- Given an address, apply the given bit mask and shift right by the given number of bits. Returns the page number or index. This function can be used to extract the **page number (index) of any page level** or the **full-page number** by supplying the appropriate parameters.
- Example: With a 32-bit system, suppose the level 1 pages occupied bits 22 through 27, and we wish to extract the level 1 page number of address 0x3c654321.
    - Mask is 0b00001111110000000000, shift is 22. The invocation would be **extractPageNumberFromAddress**(0x3c654321, 0x0FC00000, 22) which should return 0x31 (decimal 49).
    - First take the bitwise '**and**' operation between 0x3c654321 and 0x0FC00000, which is 0x0C400000, then shift right by 22 bits. The last

        five hexadecimal zeros take up 20 bits, and the bits higher than this are 1100 0110 (C6). We shift by two more bits to have the 22 bits, leaving us with 11 0001, or 0x31.
    - Check out the given **bitmasking-demo.c** for an example of bit masking and shifting for extracting bits in a hexadecimal number.
  - **Note**: to get the full-Page Number from all page levels, you would construct the bit mask for all bits preceding the offset bits, take the bitwise **and** of the address and the mask, then shift right for the number of offset bits.

Programming reference and testing:

- Please refer to C/C++ programming in Linux / Unix page.
- You may use **C++ 11** standard for this assignment, see appendix below and the sample Makefile.
- We strongly recommend you set up your local development environment under a Linux environment (e.g., Ubuntu 20.04 or 22.04), develop and test your code there first, then port your code to Edoras (e.g., filezilla or winscp) to compile and test to verify. The gradescope autograder will use an Ubuntu system to compile and autograde your code.

Turning Into Gradescope

Make sure that all files mentioned below (Source code files and Makefile) contain **your name and Red ID**! **Do NOT compress / zip** files into a ZIP file and submit, submit all files separately.

- Source code files (.h, .hpp, .cpp, .c, .C or .cc files, etc.)
- Makefile
- A sample output (in a text file) from a test of your program. (use > to export standard output to a file, e.g., ./pagetrace trace.tr “6 8 8” > testoutput.txt)
- **Single** Programmer Affidavit with your name and RED ID as signature, use **Single Programmer Affidavit.docx** from Canvas Module Assignments.
- **Number of submissions:**
    a. Please note the autograder counts the number of your submissions. For this assignment, you will be allowed **99** submissions, but <u>**future assignments will be limited to around 10 submissions.**</u> As stressed in the class, you are supposed to do the testing in your own dev environment instead of using the autograder for testing your code. It is also the responsibility of you as the programmer to sort out the test cases based on the requirement specifications instead of relying on the autograder to give the test cases.

Grading

Passing 100% autograding may NOT give you a perfect score on the assignment. The structure, coding style, and commenting will also be part of the rubrics for the final grade (**see Syllabus Course Design - assignments**). Your code shall follow industry best practices:

- Be sure to comment on your code appropriately. Code with no or minimal comments are automatically lowered one grade category.
- Design and implement clean interfaces between modules.
- Meaningful variable names.

- NO global variables.
- NO hard code – Magic numbers, etc.
- Have proper code structure between .h and .c / .cpp files, do not #include .cpp files.

Test data including the correct program output are given to you, **any hardcoding** to generate the correct output without implementing the tree will automatically **result in a zero grade** of the assignment.

Academic honesty

Posting this assignment to any online learning platform and asking for help is considered academic dishonesty and will be reported.

An automated program structure comparison algorithm will be used to detect code plagiarism.

- Plagiarism detection generates similarity reports of your code with your peers as well as from online sources. It would be purely based on similarity check, two submissions being like each other could be due to both copied from the same source, whichever that source is, even the two students did NOT copy each other.
- We will also include solutions from the popular learning platforms (such as Chegg, github, etc.) as well as code from **ChatGPT** (see below) as part of the online sources used for plagiarism similarity detection. Note not only the plagiarism detection checks for matching content, but also it checks the structure of your code.
- **If plagiarism is found in your code**, you will be automatically disenrolled from the class. You will also be reported for plagiarism.
- Note the provided source code snippets for illustrating proposed implementation would be ignored in the plagiarism check.

The Center for Teaching & Learning and Instructional Technology Services also shared information concerning the rapidly evolving impact of artificial intelligence (AI) within academia — e.g., ChatGPT, etc.

SDSU's Center for Student Rights and Responsibilities have officially added plagiarism policies of using AI generated content for academic work, as put below:

- "Use of ChatGPT or similar entities [to represent human-authored work] is considered academic dishonesty and is a violation of the Student Code of Conduct. Students who utilize this technology will be referred to the Center for Student Rights and Responsibilities and will face student conduct consequences up to, and including, suspension."

Appendix

Parse command line arguments

Suppose you put the main program code in a file main.cpp (C++) or main.c (C). It should contain the **main(int argc, char **argv)** function. Implementation tips are below.

In the signature of the **main**(**int argc, char **argv**) function:

- **argc** gives the number of command line arguments of the program including the starting executable name, using
  ./pagetrace trace.tr “4 8 8” as an example, **argc would be 3**
- **argv** contains all command line arguments starting from the executable name, using the above command line execution example:
  - **argv[0]** would be ./pagetrace
  - **argv[1]** would be the file path to the trace.tr
  - **argv[2]** would be “4 8 8”

Use unsigned int for representing addresses and masks

In a 32-bit address space, each address has 32 bits. You can use an **unsigned int** type in C/C++ to represent an address, the size of it is 32 bits in both the 32-bit and 64-bit systems. Using **unsigned** is because an address cannot be negative.

Reading memory trace file

Use ***tracereader.h*** and ***tracereader.c*** for reading addresses from the trace file:

- You will need to call ***NextAddress*** function to get the next address.
- Please note that it takes a FILE pointer (output of fopen), and a *pointer* to an object of type p2AddrTr. This means that you need to allocate an object of that type. Please note that we are using the term object loosely here, this is not a C++ object, but rather a structure that is typedefed in the *tracereader.h* file. It has several fields, but the most important one is addr which will contain the address. The return code lets you know if an address was found or not.

Read the *tracerreader.c* code for further details. You only **need to call the NextAddress** function, see below for a quick example:

```
p2AddrTr mtrace;

unsigned int vAddr;

//tracef_h - file handle from fopen

if(NextAddress(tracef_h, & mtrace))
{
  vAddr = mtrace.addr;
}
```

Bit masking and shifting

Refer to ***bitmasking-demo.c*** for a demo of code for bit masking and shifting.

Debugging tip

If you are using gdb, you can print out a number in hexadecimal with p/x; e.g. p/x addr if addr contains a value you want to inspect in hex form. It is much easier to think about bit patterns in hexadecimal (or binary) than in decimal when you are trying to think about bit positions. Remember printf and cout also have methods to print hex as well.

Makefile

a. A Makefile is for compiling and linking C/C++ code to generate an executable file. The sample Makefile provided is for compiling and linking C++. Suppose you have orgchart.h and orgchart.cpp for tree implementation, and countOrgEmployees.cpp having the main function that orchestrates the program flow.
   - CXX is the variable specifying the C++ compiler as g++ (note autograder uses g++ compiler version 7 or 9, Edoras has g++ 4.8.x installed)
   - CXXFLAGS (-std=c++11 -Wall -g3 -c) specifies compilation flags to the compiler specified by CXX, instructs the compiler to use the ISO 9899 standard C++ implementation published in 2011 (commonly called c++11, c11 for C), c++11 and c11 have functionality that is desirable (e.g., the bool type in C and the type safe nullptr in C++). -g3 adds debugging information to the executables. Debugging information lets the use of the GNU symbolic debugger (gdb) to debug your programs.

   If you are writing in C, you would change the CXX= and CXXFLAGS= to:
   - `CC=gcc`
   - `CCFLAGS=-std=c11 -g -c`

b. To compile your code, simply type **make** at the prompt. For the C++ version, make will execute compilation similar to the following if you do not have any errors:

```
g++ -std=c++11 -g3 -Wall -c pageTableLevel.cpp
g++ -g3 -c tracereader.c
g++ -std=c++11 -g3 -Wall -c main.cpp
g++ -g3 -c log.c
g++ -o pagetrace pageTableLevel.o tracereader.o main.o log.o
```

   With either C or C++, -o specifies the output file. The -c flag implies that we are compiling part of a program to an object file (machine code, but not a stand-alone program). Makefiles usually do this as it permits us to not recompile the whole program when only one module has changed. The last line links the object files together and adds some glue to create an executable program.

c. Use “make clean” to delete compiled object code and the executable.