Pavlos Spanoudakis (sdi1800184)

Project 2 on Operating Systems class

Documentation & Additional comments

Project structure

- `main.cpp` : Main client program & main functions for LRU & Second Chance algorithms
- `utils.cpp` & `utils.hpp`: Useful routines that are used throughout the program, as well some `#define`'s that do not need to be changed unless the trace format changes.
- `page_table.cpp` & `page_table.hpp` : Page Table related classes and functions
- `page_handling.cpp` & `page_handling.hpp` : Page Replacememt Algorithm related classes and functions
- `Makefile`
- `traces` directory with the 2 given trace files

Compilation & Execution

- To build, run `make`.
  Note that c++11 is required, because `std::forward_list` is used. The -std=c++11 flag is already enabled in the `makefile`. Another requirement is `openssl/sha`, which is used for hashing.
- To execute, run `./main <lru>/<2ch> <Number of Frames> <References per Process turn> <max total traces>`.
  For example: `./main lru 100 25 10000` or `./main 2ch 100 40`.
  The 3 last arguments can only be positive numbers. The `<max total traces>` argument is optional. If it is not specified, the execution will end when EOF is found in one of the input files.
- To clean up when done, run `make clean`.

Operation

This is a process Page Management simulation. The program reads page traces from the 2 "processes" (their respective input files) interchangeably, and handles it properly.

Each trace is an 8-digit hexadecimal number, followed by `W` (write) or `R` (read) character, for example `001fe308 R` or `002af402 W`.
The pages have the same size as the frames (4096 bytes), so each frame can contain one page at a time. Since the page size is 4096 bytes, the 12 less-significant bits of the trace represent the page offset, and the rest represent the page number (there is no outer-inner page table).

After a trace is read, it is checked whether the page exists in memory, by looking up in the Page Table.
If there is a page fault, the requested page is retrieved from the disk and placed in a memory frame. If all frames are occupied, a page is evicted, by following the specified memory replacement algorithm. A global allocation scheme is followed: a page fault from one process may cause a page of the other process to be evicted.

During the execution, the program keeps track of:

- Read from Disk operations
- Page Faults
- Write-back at Disk operations (a to-be-evicted page is marked as "modified")

Hashed Page Table

A Hashed Page Table is used for each process, with `SHA1` used as a hash function.
The Page Table is a typical array of `PageTableBucket`'s.
Each `PageTableBucket` is an `std::forward_list` of `PageTableEntry` elements (a simple, single-linked list).
The Bucket also keeps track of the last item in the list using an `iterator`, to speed up insertions. Each `PageTableEntry` contains:

- The page number
- The number of the frame where the page is placed
- Bits for recent reference and modification.

When a page is evicted, it is removed from the page table of its process.

All the above, along with their functions, classes and methods are defined and implemented in `page_table.hpp` & `page_table.cpp` respectively. Since this is a memory management simulation, all class fields are publicly visible, since using setter & getter function calls could slow down the execution.

Second Chance Page Replacement

To implement the Second Chance algorithm, we use an `std::deque` (a double-ended queue) of `QueueEntry` elements.
Each `QueueEntry` contains:

- A pointer to the Page Table Entry for the page it is associated with
- The ID of the process that this page belongs to

When there is a Page Fault and all memory frames are occupied, a page must be evicted.
We check the first element in the Queue: If the reference bit (in the Page Table Entry) is `true`, we remove the entry from the head of the Queue and re-insert it at the back of the Queue, after setting the bit to `false`.
We repeat this procedure, until the reference bit of the element at the head of the Queue is `false`. When this is the case, we evict that page (deleting the Page Table Entry & the Queue Entry etc.) and insert the new one at the now available frame.

LRU Page Replacement

To implement the LRU algorithm, we use a double-linked list (`std::list`) of `QueueEntry` elements.
When we receive a trace and detect that the page is in memory (by looking at the page table), we would normally have to iterate over the list to find the entry for this page and place it the top of the list. That would cost O(n) time at worst.
To prevent that, we use a lookup hash table. This table consists of `LRU_LookupBucket`'s. Each bucket contains a list of iterators that point to `QueueEntry` elements (`QueueIteratorList`).
When we want to find the `QueueEntry` for a specific page of a specific process, we get the hashcode for this page (same as for the Page Table), head to the corresponding bucket, and search for it in the bucket list. In this way, the time for finding an entry for a page in the LRU "Queue" is reduced.
Of course the quickest way would be to store the iterator for each `QueueEntry` in the `PageTableEntry` and

have access to it as soon as we find out whether the page is in the memory or not. It was decided not to choose this implementation, to keep the Page Table structure more minimal (since a `PageTableEntry` contains bit information as well).

Files & Output

The 2 input trace file paths are defined in `main.cpp`.
Before the program starts reading traces, the user's selections (as specified in the arguments) are printed. If any option was invalidly specified, the program informs the user and exits.
After the end of the execution the program displays all the tracked statistics:

- Read from Disk operations
- Page Faults
- Write-back at Disk operations
- Number of read traces

It should be noted that when EOF is reached, the execution always stops, even if the user specified maximum number of traces is not reached yet:
For example, if the 2 files have 100 traces each, and the user asks for 26 traces per process turn and 190 total traces, the program will read 6x26 = 156 traces + 22 from the first file, 178 in total (because after having read 3x26 = 78 traces from the first file, only 22 traces will be left in it).

Resource handling

All resources are properly allocated and released throughout the execution. No memory leaks are reported by Valgrind, after extensive testing.

Note that that there is a reported "bug" in Valgrind used in DIT Labs, regarding C++ `iostream` memory pooling, which causes some bytes to be reported as "still reachable" after the end of the execution of **any** C++ program. You can read more in 4.1 here and 6.7 here. When testing with Valgrind in DIT Labs, apart from the above "leak", no other leaks were reported.

Development & Testing

Developed in WSL Ubuntu 20.04, using Visual Studio Code. Successfully tested in DIT Lab Ubuntu 16.04 as well.