Simulating Virtual Memory

CIS 310: Operating Systems

Brian C. Ladd

Fall 2021

Introduction

Program reads a *virtual address trace* (mixed with some commands) and translates it to a *physical* address trace. As written, the memory spaces, both virtual and physical, are hardwired. They can easily be changed and the data structures, `RAM` and `PageTable` must be set up to permit that.

User Interaction

The command loop is provided in `main` (`vmSimulator.cpp`). It reads standard input by line. The '#' character starts an EOLN style comment, commands cannot span a line, and blank lines are ignored by the processor.

A simulated clock, the event clock, is maintained and increments every time an address is accessed.

Commands

READ XXXXXXXX

WRITE XXXXXXXX These two commands contain the virtual address trace. The XXXXXXXX is a 32-bit virtual address expressed in hexadecimal. Each of these is processed by looking up the frame number using the page number and page table; if that fails (page is not loaded in a frame), then that page is loaded. When the page is accessed, the reference timestamp in the frame is updated to the current event clock value and the referenced bit in the page table is set.

PAGES The page table is dumped to standard output. The format for this is documented in the printing functions for a page table entry.

FRAMES The content of the RAM (frames) is dumped to standard output. The format for this is documented in the printing function for a frame.

TIME Use the frame timestamps to find the LRU "victim" frame on a page fault.

REF Use the page referenced bit to find the LRU "victim" frame on a page fault.

CLEAR Clear referenced bits for all pages.

Getting Started

You must finish the data structures used by `vmSimulator`. As shipped, this code will **not** compile. This is because all `.cpp` files in the `physical` and `virtual` modules were excluded (as well as `util/virtualMemoryTypes.cpp`).

- `util/virutalMemoryTypes.cpp` implements the two functions defined in the `.h` file. These require bit manipulation to extract the page number and offset from a 32-bit value.
- The `Frame` and `PTE` types have no data members. You will need to add the ones you feel are needed.

As is mentioned in the header files for the types, there is no default constructor provided. This means that the compiler will deduce one. In particular, this means that the fields that have non-default starting values will not be set. You can solve this by either writing a default constructor -or- using {} initialization when declaring the fields in the header files.

- While you saw the use of C++ lambda functions in class along with the use of the `<algorithm>` header, you are not expected to use either of these. All of that usage in the sample solution was pushed into the (now missing) `.cpp` files. You can do whatever you want to make the functions work.

Deliverables

A git repo with your solution.

Building the project executables

This section assumes you are in the root directory of the project (the same directory with the `Makefile`).

Build the executables

```
$ make
```

All directories listed in the make include file in `src/` are "made" (checked if generated files are out of date or non-existent and generated if necessary). The make system picks up `.cpp` and `.s` files in the listed folders. It links executables in the `build/` directory, named for the source files in the various executable folders.

To Clean — delete all generated files

The next build will have to compile *all* of the files rather than just the ones that have changed. This is helpful if different, dependent packages get out of sync.

```
$ make clean
```

Primarily deletes (recursively) the `build/` folder.

To Test

The `tests/` folder has command files

It also includes the executable `vmSimulator.benchmark` that can generate the expected output from input. It is likely that malformed command files will crash the benchmark.

To Run

Each project, when built, shows the name of the executable as the parameter of the -o commandline argument. So, in the commandline `make` used to build an executable named `vmSimulator`,

```
g++ -std=c++17 -Wall -Werror -g  -o build/vmSimulator ...
```

the executable is build in `./build/vmSimulator`. The executable is named automatically for the `.cpp` file containing the `main` function; the source file containing the `main` function is in `src/main/` which is in the `EXECUTABLE_MODULES` variable.

To run the program, type the path of the executable at the commandline

```
build/vmSimulator
```

If it takes commandline parameters, they come after the executable