T7: Memory Trace Analysis

Due Tuesday by 11:59p.m. **Points** 0 **Available** after Nov 2 at 7p.m.

Objective

Valgrind has an option that will allow you to print out the memory reference trace of a running program (using the *lackey* tool), which we are using to generate traces for the virtual memory simulator in Assignment 4. This exercise is intended to help you familiarize yourself with the traced programs, how they are using memory, and the memory trace format.

Background

There are four different C programs that can be found in `/u/csc369h/fall/pub/a4` on the teach.cs machines. We will be using address traces from these programs in our virtual memory simulator for Assignment 4. The programs are:

- `simpleloop.c` - loops over an array allocated in the heap (You can also modify the code to run the same loop using stack memory)
- `repeatloop.c` - loops over a heap-allocated array multiple times
- `matmul.c` - naive matrix multiply with the ability to change the element size to change the memory access behaviour
- `blocked.c` - a more memory-aware matrix multiply that should exhibit a better hit rate under some page replacement algorithms

The `/u/csc369h/fall/pub/a4` directory also contains some scripts that we use to generate and transform memory reference traces from these programs.

- `trimtrace.py` - discards uninteresting parts of a raw Valgrind memory trace, keeping only the program trace between BEGIN and END markers.
- `fastslim-with-offset.py` - reduces the size of a trace by removing repeated accesses to pages that are very close together in the trace (e.g., back-to-back references to the same page, or alternating references to the same pair of pages).
- `simify-trace.py` - adjusts page offsets and adds values to be written to, or read from, memory addresses.
- `run.sh` - puts it all together to run Valgrind on program specified as the first program, then trim, slim, and simifiy the resulting trace.

These programs transform the raw Valgrind memory traces to address three issues that make the raw traces unsatisfactory for our purposes.

1. Valgrind includes the entire program memory trace, including loading the program and libraries into memory, which makes it harder to focus on the access patterns of the algorithm. To address this problem, we have added special variables to the programs whose sole purpose is to serve as a marker, so that we can keep only the reference trace between the two markers. The python program `trimtrace.py` carries out this task. These address-level traces for the four programs are stored in `/u/csc369h/fall/pub/a4/traces/addr-*.ref`. **These are the trace files that you will analyze in this tutorial exercise.**
2. Even the trimmed traces are massive because they include every memory reference, even back-to-back references to the same page. The `fastslim-with-offset.py` program reduces the size of the traces to focus on distinct page accesses, but it outputs the full virtual address of each reference that it keeps. (The `fastslim.py` program is similar, but it zeros the offset portion of the virtual address. We are not using it this year but are providing it for reference.)
3. Valgrind's memory traces output the virtual address and size of the memory access, but not the value written to, or read from, memory. For our memory simulator, we want to provide some values so that we can check that virtual addresses are being translated correctly, and that paging in/out is working correctly. In addition, our simulator's physical page frames are much smaller than real pages, so we need to adjust the page offset part of the virtual address so that they are all in the range of our simulated page frames. The `simify-trace.py` program takes a slimmed trace and turns it into the needed format for the memory simulator. These traces can be found in `/u/csc369h/fall/pub/a4/traces/simvaddr-*.ref`. **You will use these traces with the virtual memory simulator in Assignment 3.**

You can read and analyze the traces from the course pub directory on a teach.cs server, however if you want to work on your own machine you can find all traces in compressed archive format at `/u/csc369h/fall/pub/a4-traces.tar.gz`. Use scp or rsync to download them to your machine. The file can normally be unpacked by a program on your system by clicking on it, or you can use `tar -xzf a4-traces.tar.gz`. **You will need approximately 1 GB of space to store the decompressed traces.**

The directory includes all of the code to build and generate the traces, so that you can try it out on different programs if you like. If you want to do this, you will need to copy the scripts to your account where you have permission to write the output files. Remember not to commit trace files to your repo because some of them are quite large.

The Exercise

Your task is to read the four C programs to get a picture of how they are using memory and write a short program in the language of your choice to analyze the trace output. Then compare your counts to the program code to match the access pattern to variables and operations in the program.

The trimmed address trace files (addr-*.ref) have the following format, where the first character indicates whether the type of access (Instruction fetch, Load, Store or Modify), the second value is the address in hexadecimal format, and the third value is the size (in bytes) of the access. For example:

```
S 04228b70,8
I  04001f4f,4
I  04001f53,3
L 04227f88,8
I  04001f99,7
L 04228a50,8
I  04001fa0,3
I  04001fa3,2
I  04001fa5,4
M 04227e80,8
I  04001fa9,7
L 04228a48,8
```

Your analysis program will translate each memory reference into a page number assuming pages are 4096 bytes. It will output three tables:

- Counts: For each reference type, a count of the number of entries in the trace of that type. The order of the output rows should be Instructions, Loads, Stores, Modifies as shown in the example below.
- Instructions: For each unique instruction page, a count of the number of accesses for that page. The output rows should be sorted in descending order by the number of accesses (i.e., the most frequently accessed page will be first, and the least frequently accessed page will be last).
- Data: For each unique data page, a count of the number of accesses for that page. The count should sum up all types of accesses (Loads, Stores and Modifies) to the page. The output rows should be sorted in descending order by the number of accesses (i.e., the most frequently accessed page will be first, and the least frequently accessed page will be last).

For example, on the above trace snippet the analysis program will produce the following:

```
Counts:
  Instructions 7
  Loads        3
  Stores       1
  Modifies     1

Instructions:
0x4001000,7

Data:
0x4228000,3
0x4227000,2
```

The tables show that there is only a single Instruction page, with 7 total references, and two Data pages that have 3 and 2 references, respectively. Page 0x4228000's 3 references include 1 Store and 2 Load references. Note that in the expected output, you should display addresses in hexadecimal format with the leading "0x" and you should output the virtual page number as a full virtual address with 0's for the offset part (i.e., 0x4228000 rather than just 0x4228).

Think about which pages are accessed the most frequently and try to explain to yourself which variables (data) or code from the program might be stored in these pages. The intent is that you should be able to do this exercise in a couple of hours.

Note that the formatting for the numbers under the Count section needs to be right-justified with 8 digits, e.g.:

```
Counts:
  Instructions   662153
  Loads          120435
  Stores          30290
  Modifies           24
```

In C, you need a format specifier like this: `%-8d`

What to submit:

You will find a "T7" assignment on MarkUs. Add the following files to your repo (for this exercise, the web upload is enabled, as well as submission via git):

- The analysis program you wrote to produce the required tables given an input trace.
- A shell script named `analyze.sh` that will run your analysis program on a trace file supplied as an argument. For example, running `./analyze.sh addr-matmul.ref` should run your analysis program on the addr-matmul.ref trace and output the required tables. Your shell script and analysis program must run on teach.cs. If you choose to write your analysis program in C, your shell script should include steps to compile the C program into an executable. You may wish to refer to our `run.sh` program for an example.
- Any additional files that are needed for your program. For example, if you wrote a C program, and have a .h file and/or a Makefile, make sure you add those files as well.

DO NOT add tracefiles or other generated files (.o files, program binaries) to your MarkUs repo.