CS 416 - Operating Systems Design

Homework 0
Detecting Cache Size / User Level Threads

January 24, 2015
Due February 12th, 2015

1 Introduction

This homework will consist of two discrete parts. In the first part, you will determine the parameters of the second-level memory cache through time measurements. In the second part, you will implement a simple user-level thread (ULT) library for Linux.

2 Part 1 - Determining L2 Cache Parameters

In class, it was discussed how the cache works to mitigate the time costly of main memory data access. The cache can be characterized by three parameters:

- Size of the Cache Block/Line
- Cache Size
- Associativity

As the cost to access data from the Main Memory is several orders of magnitude higher than from the cache, a good program design is one that most of the accessed data is available in the cache, avoiding several expensive accesses to the Main Memory. Cache's design exploits the *principle of locality*, which states that data accessed now has a higher chance to be accessed again in the near future (*temporal locality*), and that data blocks near the one accessed now have, also, higher chances to be accessed in the near future (*spacial locality*).

Different processors, even from the same manufacturer, have different Cache designs and Memory Hierarchies, in terms of Cache levels (L1, L2, L3, shared L3, etc). Refer to the Processor's Manual to know the Cache Parameters and Memory Hierarchy of your CPU.

For this part of the homework, you are asked to design a program that will allow you to extract the L2 Cache parameters. You should be able to determine:

1. Cache Block/Line Size
2. Cache Size
3. Cache Miss Penalty (the time spent when data is not cached and must be fetched from main memory)

The following list outlines some ideas that will help you to achieve this part:

- Since the cache is transparent to the application programmer, there is no direct method to measure the cache size. Therefore, the cache should be measured indirectly.
- The way we achieve this is by repeatedly running a program that accesses a large array with different patterns to lead to predictable capacity misses on the cache. By measuring the running time of the program we can infer the cache size.
- Assume that the cache line and size are a power of two.
- Make sure that the memory used in the program is already mapped into the physical memory before measuring the running time. This can be achieved by traversing the array once before starting the measurement.
- The array you traverse in your program is allocated at virtual addresses, while the L2 cache whose parameters you are measuring maps physical addresses. Your approach can ignore this since it will not significantly affect the result.

2.1 Requirements

We will provide a Makefile and simple template. You should do all of your implemtation inside the include files **analyzecache.c** and **analyzecache.h**. The Makefile will produce a binary called **analyzecache**. The output of this program should be:

```
Cache Block/Line Size: XXX B
Cache Size: XXX KB
Cache Miss Penalty: XXX us
```

Deviations from this output format will not be accepted.

You should also include a short report about your methodology and results (namely, the size of the cache line and the size of the cache, and how you determined them). This report should also contain a short description about each C file and the main function's input argument(s). Sometimes it is easier to analyze the data collected by your memory accesses in a chart, submit it if you have created one.

3 Part 2 - User Level Thread Library

For this part you will implement a cooperative User Level Thread (ULT) library for Linux that can replace the default PThreads libary. Cooperative user level threading is conceptually similar to a concept known as coroutines, in which a programming language provides a facility for switching execution between different contexts, each of which has its own independent stack. A very simple program using coroutines might look like this:

```
void coroutine1()
{
     // Some work
     yield(coroutine2);
}
```

```
void coroutine2()
{
     // Some different work
     if (!done)
          yield(coroutine1);
}

int main()
{
     coroutine1();
}
```

Note that *cooperative* threading is fundamentally different than the *preemptive* threading done by many modern operating systems in that every thread ***must*** yield in order for another thread to be scheduled.

3.1 Basic User Level Thread Library

Write a **non-preemptive cooperative user-level** thread library that operates similarly to the Linux pthreads library, implementing the following functions:

- mypthread_create
- mypthread_exit
- mypthread_yield
- mypthread_join

Please note that we are prefixing the typical function names with "my" to avoid conflicts with the standard library. In this ULT model, one thread yields control of the processor when one of the following conditions is true:

- thread exits by calling **mypthread_exit**
- thread explicitly yields by calling **mypthread_yield**
- thread waits for another thread to terminate by calling **mypthread_join**

You should use the functionality provided by the Linux functions **setcontext()**, **getcontext()**, and **swapcontext()** in order to implement your thread library. See the Linux manual pages for details about using these functions.

3.2 Requirements

We will provide a Makefile, test program, and simple template. You may not modify the test program, so your library must provide exactly the API that we specify. You should implement all of your code in the provided files **mypthread.h**, and **mypthread.c**.

4 Coding and Submission Instructions

4.1 Coding

Along with this PDF, you will find on Sakai a file named **homework0-template.tar.gz**. To get started, download this file to a Linux/Unix system and type **tar zxvf homework0-template.tar.gz** at your console. This will produce a new directory called **homework0**. Inside you will find two directories, named **analyzecache** and **ult**, respectively. In each of these directories, you will find some template source and header files, as well as a Makefile. For this assignment, you should NOT have to create any other files, simply do your implementation in these existing files.

4.1.1 analyzecache

For Part 1, you will do your implementation inside **analyzecache.c** and, possibly **analyzecache.h**. The Makefile is simple, and will produce a binary called **analyzecache**.

4.1.2 ult

For Part 2, we provide a **mypthread.h** that defines the required API. You will need to make changes to the types defined here, but do not change the function call API. You will implement your library in **mypthread.c**. Here, the Makefile is more complex. Once you do your implementation, you may compile it by typing **make ult**. You may also type **make system** to produce a version of the test program compiled against the system pthreads library (to see how it is supposed to work). The test program is implemented in **mtsort.c**, and the compiled versions will be **mtsort-ult** and **mtsort-system**, respectively.

4.2 Submission

When you have completed the project, **cd** to the top level directory (**homework0**) and type **make repack netid=YOUR_NETID**. This will produce a file called **homework0-YOUR_NETID.tar.gz**. Submit this file on Sakai along with a report in PDF or TXT format (no MS, OpenOffice, or LibreOffice).

Only one group member should submit the project, but the *names and netids* of all group members should be entered into the text field on the Sakai submission, and should also appear on the report.

5 Notes

- The code has to be written in **C** language. You should discuss on Piazza if you think inline asm is necessary to accomplish something.
- Do not copy the solution from other students. Use Piazza to discuss ideas of how to implement it. Do not post your solution there. Use Sakai to submit it.