How large is the TLB?

Johan Montelius

VT2016

Introduction

The Translation Lookaside Buffer, TLB, is a cache of page table entries that are used in virtual to physical adress translation. Since the cache is of finite size we should be able to detect a difference in the time it takes to perform a memory operation dependng on if we find the page table entry in the cache or not.

We will not have access to any operations that will explicitly control the TLB but we can access pages in a way that forces the CPU to have more or less page hits. We will set up a number of virtual pages in memeory and if we access them linearly we should be able to see some changes when we have more pages tha the TLB can handle.

1 Measuring time

To explore the difference between hitting the TLB and having to find the page table entry in memory, we will write a small benchmark program that accesses more and more pages to see when the number is to large to fit in the TLB. In order to do this we firts of all need a way to measure time.

1.1 process or wall time

First of all we need to decide if we want to measure the *process time* or the *wall time*. The process time is the time our process is scheduled for execution whereas the wall time also includes the time the process is suspended. The process is suspended if the operating system has to schedue another orocess or if our porcess is waiting for something, for example I/O.

Look up the manual entry for the two procedures:

- `clock()`
- `clock_gettime()`

The first procedure will give us a time stamp that we can use to calculate the elapsed process time. There is then a macro, `CLOCKS_PER_SEC`, that we use to translate the timestamp into seconds. This is the time we would normally use if we are not doing any I/O or other things that we also want to include in our measurements.

We will also measure the time using `clock_gettime()` to see if there is any difference. We will use the clock `CLOCK_MONOTONIC_COARSE` that suits our purposes. There are other clocks but we dont't want to use a clock that will jump backwards if the CPU clock is reset by a NTP procedure. The procedure `clock()` is implented using the `CLOCK_PROCESS_CPUTIME_ID`; there are six other clocks to choose from so something that we tought was simple is quite complicated.

The first thing we do is to check that our timing procedure works so we set up the experiment using a dummy operation. This will give us information on the accuracy of our clock and how muck time it takes just to run the benchmarks program.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define PAGES 32
#define REFS (1024*1024*1024)

int main(int argc, char *argv[]) {

  clock_t c_start, c_stop;

  struct timespec t_start, t_stop;

  printf("#pages\t proc\t wall\n");

  for(int pages = 1; pages <= PAGES; pages++) {

    int loops = REFS / pages;

    clock_gettime(CLOCK_MONOTONIC_COARSE, &t_start);

    c_start = clock();

    int m = 0;

    for(int l = 0; l < loops; l++) {
      for(int p = 0; p < pages; p++) {
        /* dummy operation */
        m++;
      }
    }
```

```
      c_stop = clock();

      clock_gettime(CLOCK_MONOTONIC_COARSE, &t_stop);
      {
        double proc;
        long wall;

        proc = ((double)(c_stop - c_start))/CLOCKS_PER_SEC;
        wall = t_stop.tv_sec - t_start.tv_sec;

        printf("%d\t %.2f\t %ld\n", pages, proc, wall);
      }
    }
    return 0;
}
```

Compile and run the program, what can we say about the accuracy of our clock. Is two decimals ok or should we report our findings using more or less? The wall time is reported only in seconds and is as you probably see more or less equal to the process time. You might wonder what the point is to include this but this will all be clear once you're finished with this assignment.

2 The benchmark

Ok, so now for the actual benchmark. We're going to allocate a huge array that we will access in a linear fashion. We will make a total of `REFS` references and the questio is if it takes more time to access 4 pages compared to 16 or 32 pages. As you see in the source code for the timer we keep the total numner of references constant so the difference in the time it takes to reference a page will be immediately observerable.

We first define the page size, and we start with a value that we know is too low; we run our benchmark with a page size of 64 bytes.

```
#define PAGESIZE 64
```

Then we allocate a huge array and why not call it `memory`. Note that we cast the `PAGESIZE` to a long integer i order for the multiplication to work even if we start to use a total memory that is larger than 4Gib (and we will in the end). We also run through all pages and write to them just to force the operating system to actually allocate the pages and set up the TLB entries.

```
  char *memory = malloc((long)PAGESIZE * PAGES);

  for(int p = 0; p < PAGES; p++) {
```

```
    long ref = (long)p * PAGESIZE;
    memory[ref] += 1;
  }
```

We also add some nice print-out so that we can look at our figures later and see what parameters we used.

```
  printf("#TLB experiment\n");
  printf("#  page size = %dKib\n", (PAGESIZE/1024));
  printf("#  max pages = %dKi\n", (PAGES/1024));
  printf("#  total number or references = %dMi\n", (REFS/(1024*1024)));
  printf("#pages\t proc\t wall\n");
```

Then it is time to do benchmark and we simply replace the dummy operation with a page reference and we are ready to go.

```
  long ref = (long)p * PAGESIZE;
  memory[ref] += 1;
```

If you have everything in place you should see something like the printout below; nothing much happens and that is what we would expect. We're referencing the same page so we will have a TLB hit no mather what we do.

```
#TLB experiment
#  page size = 0Kib
#  max pages = 0Ki
#  total number or references = 1024Mi
#pages  proc  wall
1  3.34  3
2  3.23  4
3  3.08  3
4  2.99  3
  :
  :
```

Now make things a bit more interestng by setting the page size to 4Kibyte, `(4*1024)`. What does the performance look like now?

3 Turn it into a graph

So you now have a nice printout of the time it takes to reference a page depending on the number of pages that we use. As you see there is definately a penalty as we use more pages. We can create a nice plot of the values using a tool called `gnuplot`.

First we save the output in a file called `tlb.dat`. This is easily done using the redirection operator of the shell.

```
$ gcc tlb.c -o tlb32
$ ./tlb32 > tlb32.dat
```

Now, you can run `gnuplot` in interactove mode (and you should learn how to do so to quickly generat a graph) but it's better to write a small script that generats the plot. Write the following in a file `tlb32.p`.

```
set terminal png
set output "tlb32.png"

set title "TLB benchmark, 32 pages, 1 Gi operations"

set key right center

set xlabel "number of pages"

set ylabel "time in s"

plot "tlb.dat" u 1:2 w linespoints title "page refs"
```

Now run gnuplot from the command line and let it execute the script. You should then find a file called `tlb32.png` with a nice graph.

4 Even more pages

What happens if we use 32Ki pages? We don't have time to wait for the benchmark to try every possible limit of pages and we are probably not even interested in all 32 thousand data points. To make it easier to follow, We change the benchmark routine so that it doubles the number of pages in each test.

```
for(int pages = 1; pages <= PAGES; pages = pages*2) {
    :
    :

}
```

Compile and save this program as `tlb32k` and then generate a file `tlb32k.dat` with the output. Creata a new file `tlb32k.p` and modify the script to generate the plot for the new set of data. You want to include the following directive in teh script sin ce we want the x-scale to be logarithmic.

```
set logscale x 2
```

What is happening? Check the specifications of your CPU and see if you can find some explanation of what is goin on.

4.1 before you go to bed

One more experiment before you're done but make sure that your laptop is connected to power and don't sit up waiting. We're going to challenge the system and allocate more memory than you have in your computer. We assume that you're running a 64-bit operating system and have something like 8 Gibyte of memory. If you allocate 2 Mi page, `(2*1024*1024)`, you're asking for more memory than you have. The operating system will have to swap pages to the hard-drive in order to meet your needs. This will take time .... and I'm not joking.

Check how much memory you have on your laptop and then set the number of pages so the whole memory would be needed. Start the benchmark and go to bed. In the morning you will be surpised how long time it took.