CSE 422S: Studio 6

Timing and Benchmarking

Xingjian Xuanyuan

March 12, 2022

A man with a watch knows what time it is.
A man with two watches is never sure.

Segal's law

In this studio, we will:

1. *Benchmark userspace programs using command line tools.*
2. *Benchmark userspace programs using Linux's clock functions.*
3. *Use the same clocks to measure elapsed time in a kernel module.*

System clocks are built by incrementing a counter at a regular interval. In hardware, an oscillator produces a regular wave-form known as a clock which increments the counter at a regular frequency. In most computer systems, this oscillator is a quartz crystal cut to resonate at a specific frequency. All PCs include a clock called Real Time Clock (RTC), which is independent of the CPU and all other chips[1]. The RTC continues to tick even when the PC is switched off, because it is energized by a small battery. We can view it on the Linux machine by:

```
$ grep "time" /var/log/dmesg

# Third line of output:
[    0.672742] RTC time: 20:38:13, date: 02/15/22
```

Linux uses the RTC only to derive the time and date at boot time. Before stepping into the studio exercises, let's first take a look at the available clock sources on the Linux machine:

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
```

including the one automatically set by the kernel at boot time:

```
$ cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

As we can see, there are three registered clock sources: `tsc` ("Time Stamp Counter"), `hpet` ("High Precision Event Timer"), and `acpi_pm` ("ACPI Power Management Timer"). Though not recommended, it is possible to select a different clock source by:

[1] All i386 PCs, and ACPI-based systems, have an RTC that is compatible with the Motorola MC146818 chip on the original PC/AT. Today such an RTC is usually integrated into the mainboard's chipset (south bridge), and uses a replaceable coin-sized backup battery. See `rtc(4) - Linux manual page`.

```
echo hpet >
↪  /sys/devices/system/clocksource/clocksource0/current_clocksource
```

Looking into /proc/cpuinfo, we may find several flags relating to “TSC”:

```
/* /arch/x86/include/asm/cpufeatures.h */

/* Intel-defined CPU features, CPUID level 0x00000001 (EDX), word 0 */
#define X86_FEATURE_TSC                         ( 0*32+ 4) /* Time Stamp
↪  Counter */

/* Other features, Linux-defined mapping, word 3 */
#define X86_FEATURE_CONSTANT_TSC        ( 3*32+ 8) /* TSC ticks at a
↪  constant rate */
#define X86_FEATURE_NONSTOP_TSC                 ( 3*32+24) /* TSC does not
↪  stop in C states */

/* Intel-defined CPU features, CPUID level 0x00000001 (ECX), word 4 */
#define X86_FEATURE_TSC_DEADLINE_TIMER         ( 4*32+24) /* TSC deadline
↪  timer */

/* Intel-defined CPU features, CPUID level 0x00000007:0 (EBX), word 9 */
#define X86_FEATURE_TSC_ADJUST                 ( 9*32+ 1) /* TSC adjustment
↪  MSR 0x3B */
```

According to Bovet & Cesati (2006):

> *All 80x86 microprocessors include a CLK input pin, which receives the clock signal of an external oscillator. Starting with the Pentium, 80x86 microprocessors sport a counter that is increased at each clock signal. The counter is accessible through the 64-bit Time Stamp Counter register, which can be read by means of the* RDTSC *(“Read TSC”) assembly language instruction*[2]*. When using this register, the kernel has to take into consideration the frequency of the clock signal: if, for instance, the clock ticks at 1 GHz, the Time Stamp Counter is increased once every nanosecond.*

For more details, see “Time Stamp Counter” in Chapter 17 of the *Intel®64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*. HPET is an external hardware timer available on some newer systems. An HPET could be 32-or-64bit and can provider higher resolution than the PIT or CMOS RTC. Due to lax specifications, HPET implementations are not guaranteed to have a high resolution or low drift. It is designed to replace the PIT and CMOS in newer systems. The ACPI Power Management Timer (or ACPI PMT) is another clock device included in almost all ACPI-based motherboards. Its clock signal has a fixed frequency of roughly 3.58 MHz. The device is actually a simple counter increased at each clock tick; to read the current value of the counter, the kernel accesses an I/O port whose address is determined by the BIOS during the initialization phase. This Red Hat reference guide documents that reading from the TSC is faster than reading from the HPET because the former means reading a register from the processor while the latter means reading a memory area. Therefore, there is a significant performance advantage when timestamping hundreds of thousands of messages per second using TSC.

[2] Some vendors have implemented an additional instruction, RDTSCP, which returns atomically not just the TSC, but an indicator which corresponds to the processor number. This can be used to index into an array of TSC variables to determine offset information in SMP systems where TSCs are not synchronized. The presence of this instruction must be determined by consulting CPUID feature bits. See Zachary Amsden, “Timekeeping Virtualization for X86-Based Architectures,” 2010.

Required Exercises

To start out on the studio, download a code package that includes five programs:

```
$ cd test_programs
$ ls
arr_search.c  dense_mm.c  Makefile  parallel_dense_mm.c  sing.c  sort.c
```

By looking inside the `dense_mm.c` file, we can tell that it takes an integer $N \leq 65536$ as input and performs matrix multiplication by first representing three matrices as one-dimensional arrays, then assigning random values to two matrices' $2 \times N \times N$ entries, and finally using three `for` loops to complete the multiplication. The `parallel_dense_mm.c` program differs from `dense_mm.c` in that it calculates the multiplication twice and verifies two results. The `sing.c` program takes an input integer as the number of iterations to display 9 lines of verses. The `sort.c` program takes an input integer as array size and performs quick sort in which the values of array elements and the pivot are determined randomly.

Coarse grained benchmarking can be done directly from the command line using the `time` utility built into the bash shell. It outputs three different pieces if timing information. The **real** time refers to elapsed wall-clock time. The total CPU time is the sum of **user** CPU time and **system** CPU time. User CPU time is the amount of time spent performing certain tasks for a userspace program while system CPU time is the amount of time spent executing system calls in the kernel. The real time may not be equal to the sum of user and system CPU times.

Now compare the results of the following two commands:

```
time ./dense_mm 1000
time ./parallel_dense_mm 1000
```

The output from the first command:

```
real    0m5.031s
user    0m5.017s
sys     0m0.011s
```

We can see that the `real` time is slightly bigger than the `user` time (and the sum of the latter two times). The output from the second command:

```
real    0m0.492s
user    0m13.416s
sys     0m0.014s
```

This time, the `real` time is only a tenth of running `dense_mm` while the `user` time accumulates up to more than twice of running `dense_mm`. If we run the commands for multiple times, the above observations still hold. The `sys` time does not change much among these outputs. One possible reason for the `user` time greatly exceeding the `real` time is that the program is executing in a parallelized manner.

Look at the code in `sing.c` and execute the following command:

```
time ./sing 1000
```

From output:

```
real    0m0.175s
user    0m0.010s
sys     0m0.032s
```

we can see that both the `real` and `user` times are rather small. However, the `sys` time increases. This could largely be attributed to 9000 `printf` function calls.

Now we are going to switch to using the C API for Linux's clocks. The POSIX 1003.1b standard introduced a type of software timers for User Mode programs—in particular, for multithreaded and real-time applications. Every implementation of the POSIX timers must offer to the User Mode programs a few **POSIX clocks**, that is, virtual time resources having predefined resolutions and properties. Whenever an application wants to make use of a POSIX timer, it creates a new timer resource specifying one of the existing POSIX clocks as the timing base. According to The IEEE Std 1003.1-2017, the `<time.h>` header shall define the following symbolic constants:

- `CLOCK_MONOTONIC`: the identifier for the system-wide monotonic clock, which is defined as a clock measuring real time, whose value cannot be set via `clock_settime()` and which cannot have negative clock jumps.
- `CLOCK_PROCESS_CPUTIME_ID`: the identifier of the CPU-time clock associated with the process making a `clock()` or `timer*()` function call.
- `CLOCK_REALTIME`: the identifier of the system-wide clock measuring real time.
- `CLOCK_THREAD_CPUTIME_ID`: the identifier of the CPU-time clock associated with the thread making a `clock()` or `timer*()` function call.

The Linux kernel implements the POSIX timers by means of dynamic timers[3]. Open a Terminal window on the Linux machine, run the following command:

```
man 2 clock_getres
```

to get up-to-date information about each available clock. `clock_getres()` takes two argument: the `clockid` argument is the `clockid_t` type identifier of the particular clock on which to act and the `struct timespec` pointed to by `res` stores the resolution of the specified clock `clockid`. The `timespec` structure is specified in `<time.h>`:

```
struct timespec {
    time_t  tv_sec;     /* seconds */
    long    tv_nsec;    /* nanoseconds */
};
```

Use `clock_getres` to write a short program (`getres.c`) that gives the resolutions for several different clock types.

```
/* This program is written to compare different POSIX clock resolutions */
#include <stdio.h>
#include <time.h>

int main()
{
  int rc;
  struct timespec res;

  rc = clock_getres(CLOCK_REALTIME, &res);
  if (!rc) printf("CLOCK_REALTIME: %ldns\n", res.tv_nsec);

  rc = clock_getres(CLOCK_REALTIME_COARSE, &res);
```

[3] The Linux 2.6 kernel offers two types of POSIX clocks: `CLOCK_REALTIME` and `CLOCK_MONOTONIC`. For implementation details, see this patch by George Anzinger.

```c
  if (!rc) printf("CLOCK_REALTIME_COARSE: %ldns\n", res.tv_nsec);

  rc = clock_getres(CLOCK_MONOTONIC, &res);
  if (!rc) printf("CLOCK_MONOTONIC: %ldns\n", res.tv_nsec);

  rc = clock_getres(CLOCK_MONOTONIC_COARSE, &res);
  if (!rc) printf("CLOCK_MONOTONIC_COARSE: %ldns\n", res.tv_nsec);

  return 0;
}
```

The resolutions of CLOCK_REALTIME and CLOCK_MONOTONIC are both 1ns while the other two are 1ms. Currently, timers in Linux are only supported at a resolution of 1 jiffy. The length of a jiffy is dependent on the value of HZ (the frequency of the system timer, or the "tick rate") in the Linux kernel, and is 1ms on i386 and some other platforms, and 10ms on most embedded platforms. On Linux systems, we obtain 100 by issuing the command getconf CLK_TCK. CLOCK_MONOTONIC and CLOCK_MONOTONIC_COARSE both use wall_to_monotonic to convert a value derived from xtime. The man page is vague about the features of CLOCK_MONOTONIC_COARSE. According to the original article by John Stultz, CLOCK_MONOTONIC_COARSE (as well as CLOCK_REALTIME_COARSE) returns the time at the last tick, which consumes less time than CLOCK_MONOTONIC. Thus, typically, the _COARSE variants of the POSIX clocks are preferred on systems which use hardware clocks with high costs for the reading operations.

Write a second short program called getdelay.c that uses the function clock_gettime() to figure out how long a call to clock_gettime takes. Note that To compile a program that uses clock_gettime, you need to link with librt.a (the real-time library) by specifying -lrt on the compile line. My getdelay.c program takes an input as the number of iterations. For each iteration, I write ten clock_gettime() functions. In the end, I print out the average.

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

int main(int argc, char **args)
{
  struct timespec t_start, t_end, t_overhead;
  int i, iterations;
  long long res;
  iterations = atoi(args[1]);
  // The first call could be an outlier
  clock_gettime(CLOCK_MONOTONIC, &t_start);
  for (i = 0; i < iterations; i++) {
    clock_gettime(CLOCK_MONOTONIC, &t_overhead);
    clock_gettime(CLOCK_MONOTONIC, &t_overhead);
    clock_gettime(CLOCK_MONOTONIC, &t_overhead);
    clock_gettime(CLOCK_MONOTONIC, &t_overhead);
    clock_gettime(CLOCK_MONOTONIC, &t_overhead);
    clock_gettime(CLOCK_MONOTONIC, &t_overhead);
    clock_gettime(CLOCK_MONOTONIC, &t_overhead);
    clock_gettime(CLOCK_MONOTONIC, &t_overhead);
    clock_gettime(CLOCK_MONOTONIC, &t_overhead);
    clock_gettime(CLOCK_MONOTONIC, &t_overhead);
  }
  clock_gettime(CLOCK_MONOTONIC, &t_end);

  // Calculate empirical mean of overhead
  res = (((int64_t) t_end.tv_sec * 1000000000 + (int64_t) t_end.tv_nsec) -
        ((int64_t) t_start.tv_sec * 1000000000 + (int64_t)
        ↪  t_start.tv_nsec)) / iterations / 10;
  printf("Estimated overhead of clock_gettime(): %lldns\n", res);

  return 0;
}
```

What confused me is that when I used `CLOCK_MONOTONIC` inside `clock_gettime`, the output value was 26ns; when I used `CLOCK_THREAD_CPUTIME_ID`, the output value was 672ns. I did not expect the difference could be this large. I then wrote a test program `gettime.c` to compare the two clocks:

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

int main(int argc, char **argv)
{
  int64_t elapse;
  struct timespec t_start, t_end;
  int i;

  clock_gettime(CLOCK_MONOTONIC, &t_start);
  sleep(1);
  clock_gettime(CLOCK_MONOTONIC, &t_end);
  elapse = 1000000000 * (t_end.tv_sec - t_start.tv_sec) +
                        (t_end.tv_nsec - t_start.tv_nsec);
  printf("Time gap for CLOCK_MONOTONIC: %lldns\n", elapse);

  clock_gettime(CLOCK_THREAD_CPUTIME_ID, &t_start);
  sleep(1);
  clock_gettime(CLOCK_THREAD_CPUTIME_ID, &t_end);
  elapse = 1000000000 * (t_end.tv_sec - t_start.tv_sec) +
                        (t_end.tv_nsec - t_start.tv_nsec);
  printf("Time gap for CLOCK_THREAD_CPUTIME_ID: %lldns\n", elapse);

  return 0;
}
```

And I obtained the following output:

```
$ ./gettime
Time gap for CLOCK_MONOTONIC: 1000139392ns
Time gap for CLOCK_THREAD_CPUTIME_ID: 32311ns
```

Additionally, I ran the `clock.c` program from Bill Torpey's blog post, which calls `clock_getres` and `clock_gettime` successively, on my Linux machine and obtained the following output:

```
               clock            res (ns)            secs            nsecs
        gettimeofday               1,000   1,647,046,506      311,761,000
        CLOCK_REALTIME                      1         1,647,046,506         311,787,025
   CLOCK_REALTIME_COARSE            1,000,000         1,647,046,506         310,430,602
       CLOCK_MONOTONIC                      1            18,435,533         601,596,630
     CLOCK_MONOTONIC_RAW                    1            18,434,594         511,148,203
  CLOCK_MONOTONIC_COARSE            1,000,000            18,435,533         600,217,457
CLOCK_PROCESS_CPUTIME_ID                    1                     0           2,460,708
 CLOCK_THREAD_CPUTIME_ID                    1                     0           2,475,055
```

Copy `parallel_dense_mm.c` into a new file called `timed_parallel_dense_mm.c`. First modify the code in the new file so that the critical computational loop is timed with the `CLOCK_MONOTONIC_RAW` clock. Then modify the code again so that the program takes a second parameter (which defaults to 1) that specifies the number of iterations the timed segment is executed, and output the minimum, mean, and maximum times over all timed iterations.

```c
/* My timed_parallel_dense_mm.c program */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>

const int num_expected_args = 2;
const unsigned sqrt_of_UINT32_MAX = 65536;
const long BILLION = 1000000000L;

// http://c-faq.com/stdio/commaprint.html
#include <locale.h>
char *commaprint(unsigned long n)
{
        static int comma = '\0';
        static char retbuf[30];
        char *p = &retbuf[sizeof(retbuf)-1];
        int i = 0;

        if(comma == '\0') {
                struct lconv *lcp = localeconv();
                if(lcp != NULL) {
                        if(lcp->thousands_sep != NULL &&
                                *lcp->thousands_sep != '\0')
                                comma = *lcp->thousands_sep;
                        else    comma = ',';
                }
        }

        *p = '\0';

        do {
                if(i%3 == 0 && i != 0)
                        *--p = comma;
                *--p = '0' + n % 10;
                n /= 10;
                i++;
        } while(n != 0);

        return p;
```

```
}

int main( int argc, char* argv[] )
{
    unsigned index, row, col; // Loop indices
    unsigned matrix_size, squared_size;
    double *A, *B, *C;
    #ifdef VERIFY_PARALLEL
    double *D;
    #endif

    unsigned iterations = 1; // Default number of iterations
    unsigned i = 0;
    unsigned long min = 2147483647, max = 0, sum = 0, average = 0;
    struct timespec start, end;
    long interval;

    if ( argc < num_expected_args || argc > num_expected_args + 1 ) {
        printf("Usage: ./dense_mm <size of matrices> <number of
         ↪  iterations>\n");
                exit(-1);
    }

    matrix_size = atoi(argv[1]);
    if ( argc == 3 ) iterations = atoi(argv[2]);

    if ( matrix_size > sqrt_of_UINT32_MAX ) {
        printf("ERROR: Matrix size must be between zero and 65536!\n");
                exit(-1);
    }

    squared_size = matrix_size * matrix_size;

        printf("Generating matrices...\n");

        A = (double*) malloc( sizeof(double) * squared_size );
        B = (double*) malloc( sizeof(double) * squared_size );
        C = (double*) malloc( sizeof(double) * squared_size );
        #ifdef VERIFY_PARALLEL
        D = (double*) malloc( sizeof(double) * squared_size );
        #endif

    for( index = 0; index < squared_size; index++ ) {
                A[index] = (double) rand();
                B[index] = (double) rand();
                C[index] = 0.0;
                #ifdef VERIFY_PARALLEL
                D[index] = 0.0;
                #endif
        }

        printf("Multiplying matrices...\n");
    for ( i = 0; i < iterations; i++ ) {
```

```c
        clock_gettime( CLOCK_MONOTONIC_RAW, &start );
        #pragma omp parallel for private(col, row, index) // Critical
         ↪  section
            for( col = 0; col < matrix_size; col++ ) {
                    for( row = 0; row < matrix_size; row++ ) {
                            for( index = 0; index < matrix_size; index++) {
                                C[row*matrix_size + col] +=
                                 ↪  A[row*matrix_size + index]
                                 ↪  *B[index*matrix_size + col];
                            }
                    }
            }
        clock_gettime( CLOCK_MONOTONIC_RAW, &end );
        interval = (end.tv_sec * BILLION - start.tv_sec * BILLION) +
         ↪  (end.tv_nsec - start.tv_nsec);
        sum += interval;
        if ( interval < min ) min = interval;
        if ( interval > max ) max = interval;
    }
    average = sum / iterations;
    printf("%25s\t%15s\t%15s\n", "Statistics", "secs", "nsecs");
    printf("%25s\t%15s\t%15s\n", "Minimum", commaprint(min/BILLION),
     ↪  commaprint(min%BILLION));
    printf("%25s\t%15s\t%15s\n", "Maximum", commaprint(max/BILLION),
     ↪  commaprint(max%BILLION));
    printf("%25s\t%15s\t%15s\n", "Average", commaprint(average/BILLION),
     ↪  commaprint(average%BILLION));

    #ifdef VERIFY_PARALLEL
        printf("Verifying parallel matrix multiplication...\n");
        for( col = 0; col < matrix_size; col++ ) {
                for( row = 0; row < matrix_size; row++ ) {
                        for( index = 0; index < matrix_size; index++) {
                        D[row*matrix_size + col] += A[row*matrix_size +
                         ↪  index] *B[index*matrix_size + col];
                        }
                }
        }

        for( index = 0; index < squared_size; index++ )
                assert( C[index] == D[index] );
        #endif //ifdef VERIFY_PARALLEL

        printf("Multiplication done!\n");

        return 0;
}
```

For `matrix_size = 100`, I first ran the program for 10 iterations, and obtained the following results:

```
               Statistics                  secs              nsecs
                  Minimum                     0          6,816,510
                  Maximum                     0         10,551,090
                  Average                     0          8,129,290
```

Then for 100 iterations:

```
Statistics          secs          nsecs
   Minimum             0      5,690,520
   Maximum             0     10,641,670
   Average             0      6,341,260
```

And finally for 1000 iterations:

```
Statistics          secs          nsecs
   Minimum             0      4,276,190
   Maximum             0     10,291,620
   Average             0      4,491,930
```

We can see that: (1) the maximum timing value does not differ much across different numbers of iterations used; (2) the minimum, average, and the difference between the two decrease as the number of iterations increases.

For the final task, save a copy of `simple_module.c` in the previous studio and rename it to `ktime_module.c`. Modify the module as follows:

- Create two static global variables of type `ktime_t`: `init_time` and `exit_time`.
- Call one of the `ktime_get*` functions described in the ktime accessors documentation in both of the module `init` and `exit` functions to set the corresponding `ktime_t` global variables.
- In the module `exit` function, use the `ktime_sub` function to get the elapsed time between module initialization and exit.
- Use the `ktime_to_timespec` function macro, which returns a `timespec` structure, to convert the elapsed time to printable type.
- Modify the `printk()` statement in the module `exit` function to print the seconds and nanoseconds elapsed from module initialization and exit.

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static ktime_t init_time, exit_time;
struct timespec elapse;

/* init function - logs that initialization happened, returns success */
static int
simple_init(void)
{
    init_time = ktime_get();
    printk(KERN_ALERT "simple module initialized\n");
    return 0;
}

/* exit function - logs that the module is being removed */
static void
simple_exit(void)
{
    exit_time = ktime_get();
    printk(KERN_ALERT "simple module is being unloaded\n");
    elapse = ktime_to_timespec(ktime_sub(exit_time, init_time));
    printk(KERN_ALERT "Time gap between module initialization and exit:
     ↪  %llds %lldns\n",
            (int64_t)elapse.tv_sec, (int64_t)elapse.tv_nsec);
}

module_init(simple_init);
module_exit(simple_exit);

MODULE_LICENSE ("GPL");
MODULE_AUTHOR ("LKD Chapter 17");
MODULE_DESCRIPTION ("Simple CSE 422 Module Template");
```

I obtained the following system log messages:

```
[  2588.532408] simple module initialization
[  2606.412881] simple module is being unloaded
[  2606.412914] Time gap between module initialization and exit: 17s
 ↪  880733185ns
```

References

Bovet, D. P. & Cesati, M. (2006), *Understanding the Linux Kernel*, 3 edn, O'Reilly Media.

Love, R. (2010), *Linux Kernel Development*, 3 edn, Addison-Wesley.