CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 15
October 26, 2016

Clocks and Time Measurement

Demo: Stopwatch

Clocks and Time Measurement

How to measure run time of a program

- There is no standard procedure in C++ for accurately measuring time.
- Time measurement depends on the software clocks provided by your computer and operating system.
- Clocks advance in discrete clicks called **jiffies**. A jiffy on the Zoo linux machines is one millisecond (0.001 seconds) long.
- Even if the clock is 100% accurate, the measured time can be off by as much as one jiffy.
- Hence, times shorter than tens of milliseconds cannot be directly measured with much accuracy using the standard software clock.

High resolution clocks

- Linux also provides high resolution clocks based on CPU timers.
- High resolution clocks are useful to the operating system for task scheduling and timeouts.
- They are also available to the user for higher-precision time measurements.
- Be aware that reading the clock involves a kernel call that takes a certain amount of time. This itself may limit the accuracy of timing measurements, even when the clock resolution is sufficiently high for the desired accuracy.
- See `man 7 time` for more information about linux clocks.

Measuring time in real systems

- Measuring code efficiency in real systems is challenging. Many factors can influence the results that are hard to control.
  - Other process running on the same machine.
  - Time spent in the OS moving data on and off disks.
  - Memory caching behavior.
- Lacking a controlled laboratory environment, one can still take steps to improve accuracy of tests:
  - Do some tests to determine what factors seem to have a sizable effect on the run time, e.g., the first run of a program is likely to be slower than subsequent runs because of caching.
  - Run the same test several times to get a feeling for the variance of results.
  - Make sure the optimizer isn't optimizing away code that you think is being executed.

Demo: Stopwatch

Realtime measurements

`StopWatch` is a class I wrote for measuring realtime performance of code.

It emulates a stopwatch with 3 buttons: `reset`, `start`, and `stop`.

At any time, the watch displays the cumulative time that the stopwatch has been running.

HirezTime class

`HirezTime` is a wrapper class for the system-specific functions to read the clock.

It hides the details of the underlying time representation and provides a simple interface for reading, computing, and printing times and time intervals.

`HirezTime` objects are intended to be copied rather than pointed at, and they try to behave like other numeric types.

Versions of `HirezTime`

There are two versions:

15-StopWatch (Linux/Unix/MacOSX) Function `gettimeofday()` returns the clock in a `struct timeval`, which consists of two `long int`s representing seconds and microseconds. The resolution of the clock is system-dependent, typically 1 millisecond. (See demo `15-StopWatch`.)

15-StopWatch-hirez (Linux only) Function `clock_gettime()` returns the clock in a `struct timespec`, which consists of two `long int`s representing seconds and nanoseconds. The resolution of the clock is system-dependent and can be obtained with the `clock_getres()` function. (See demo `15-StopWatch-hirez`.)

`HirezTime` structure

- In C++, `struct T` and `class T` are very similar. In both cases, `T` becomes a new type name.
- `struct` members are public by default.
  `class` members are private by default.
- `HirezTime` is derived from `struct timeval` or `struct timespec`, depending on the version.
- It uses `protected` derivation to hide the underlying representation.
- It presents two interfaces to the world:
  1. The normal public interface treats `HirezTime` as an opaque object.
  2. A class derived from it can access the fields of the underlying `timespec`/`timeval`.

Printing a `HirezTime` number

Something seemingly simple like printing `HirezTime` values is not so simple. Naively, one might write:

```
cout << t.tv_sec << "." << t.tv_usec;
```

where `tv_sec` and `tv_usec` are the seconds and microseconds fields of a `timeval` structure.

If `t` represents 2 seconds and 27 microseconds, then what would print is `2.27`, not the correct `2.000027`.

The class contains a `print` function that fixes this problem.

`StopWatch` class

`StopWatch` contains five member variables to remember

- Whether the watch is running or not.
- The cumulative run time to point when last stopped.
- The most recent start and stop times.

All functions are `inline` to minimize inaccuracies of measurement due to the overhead within the stopwatch code itself.

Casting a StopWatch to a HirezTime

An operator extension defines a cast for reading the cumulative time from a stop watch:

```
operator HirezTime() const { return cumSpan; }
```

Thus, if `sw` is a `StopWatch`,

```
cout << sw;
```

will print `sw.cumSpan` using `sw.print()`.

Why it works

This works because `operator<<()` is not defined for righthand operands of type `StopWatch` but it is defined for `HirezTime`.

The compiler then **coerces** `sw` to something that is acceptable to the `<<` operator.

Because `operator HirezTime()` is defined for class `StopWatch`, the compiler will invoke it to obtain a `HirezTime` object, for which `<<` is defined.

Note that a similar coercion happens when one writes

```
if(!in) {...}
```

to test if an `istream` object `in` is open for reading. Here, the `istream` object is coerced to a `bool` because `operator bool()` is defined inside the streams package.