How to profile functions with calclock

Practical Class 7-b

Systems and Storage Laboratory

Department of Computer Science and Engineering

Chung-Ang University

What is calclock?

- **Implemented to measure function execution time**
  - Accuracy is up to nanoseconds
  - When there are multiple threads, it guarantees the concurrently calculated result
    - By using atomic operations
  - Because the total time will be accumulated from every thread, you need to divide by the number of threads to get the pure execution time

Basic usage

- Let's profile some_time_consuming_function()

```
int foo(void)
{
    int output;

    output = some_time_consuming_function();

    return output;
}
```

Basic usage

- **we can utilize *calclock* to profile function overhead**
  - First, include `calclock.h` into source code
    - Make sure that you have `calclock.h` and `calclock.c` at the same directory with your source code

```
#include "calclock.h"

int foo(void)
{
    ...
}
```

Basic usage

- Define array of ***struct timepsec***, which will be used as a stopwatch

```
#include "calclock.h"

int foo(void)
{
    struct timespec stopwatch[2];
    ...
}
```

Basic usage

- **Now, capture times right before and right after the function that you want to profile with**
  - `getrawmonotonic` saves current time into stopwatch
    - Notice that parameter is reference to the `struct timespec` (pointer of `struct timespec`)

```
#include "calclock.h"

int foo(void)
{
    struct timespec stopwatch[2];
    ...
    getrawmonotonic(&stopwatch[0]);  // start point
    output = some_time_consuming_function();
    getrawmonotonic(&stopwatch[1]); // end point
    ...
}
```

Basic usage

- **Define global variables: count, time**
  - each will be the result of *calclock*
  - Notice that types are <u>`unsigned long long`</u>

```
#include "calclock.h“

unsigned long long count = 0;
unsigned long long time = 0;

int foo(void)
{
    ...
}
```

Basic usage

- **Finally, utilize *calclock* at the end of the foo()**
  - Calculates the time gap between `stopwatch[0]` and `stopwatch[1]`, and accumulates it into variable `time` by nanoseconds
  - Accumulates how many times calclock() has been called into variable `count`

```c
#include "calclock.h"

unsigned long long count, time;

int foo(void)
{
    struct timespec stopwatch[2];
    ...
    calclock(stopwatch, &time, &count);

    return output;
}
```

SySLAB Systems and Storage Laboratory

Basic usage

❖ Overall Usage of calclock

```
#include "calclock.h"

unsigned long long count, time;

int foo(void)
{
    int output;
    struct timespec stopwatch[2];

    getrawmonotonic(&stopwatch[0]);
    output = some_time_consuming_function();
    getrawmonotonic(&stopwatch[1]);
    calclock(stopwatch, &time, &count);

    return output;
}
```

Basic usage

❖ **Don't forget to include calclock.o to your Makefile!**

```
#--------- Makefile ---------#

obj-m = hello_module.o

hello_module-y := hello_module-base.o calclock.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

default:
	$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
	$(MAKE) -C $(KDIR) M=$(PWD) clean
```

Source Code

❖ **calclock.h**

```
#ifndef __CALCLOCK_H
#define __CALCLOCK_H

#include <linux/time.h>

#define BILLION 1000000000UL

unsigned long long calclock(struct timespec *myclock,
    unsigned long long *total_time, unsigned long long *total_count);

#endif
```

Source Code

❖ calclock.c

```c
#include "calclock.h"

unsigned long long calclock(struct timespec *myclock,
    unsigned long long *total_time, unsigned long long *total_count)
{
    unsigned long long timedelay = 0, temp = 0, temp_n = 0;

    if (myclock[1].tv_nsec >= myclock[0].tv_nsec) {
        temp = myclock[1].tv_sec - myclock[0].tv_sec;
        temp_n = myclock[1].tv_nsec - myclock[0].tv_nsec;
        timedelay = BILLION * temp + temp_n;
    } else {
        temp = myclock[1].tv_sec - myclock[0].tv_sec - 1;
        temp_n = BILLION + myclock[1].tv_nsec - myclock[0].tv_nsec;
        timedelay = BILLION * temp + temp_n;
    }

    __sync_fetch_and_add(total_time, timedelay);
    __sync_fetch_and_add(total_count, 1);

    return timedelay;
}
```

Analysis in detail

calclock.c

- `timedelay` : total time gap between `myclock[0]` and `myclock[1]` by nanoseconds
- `temp` : time gap between `myclock[0]` and `myclock[1]` by second
- `temp_n` : time gap between `myclock[0]` and `myclock[1]` by nanosecond less than a second

```
#include "calclock.h"

unsigned long long calclock(struct timespec *myclock,
    unsigned long long *total_time, unsigned long long *total_count)
{
    unsigned long long timedelay = 0, temp = 0, temp_n = 0;
    ...
```

Analysis in detail (cont'd)

- **calclock.c**
  - If nanoseconds in `myclock[1]` is greater than those of in `myclock[0]`, just subtract them
  - If not, borrow `BILLION` nanoseconds from `temp`, and then subtract them.
  - `timedelay` will be total difference of nanoseconds

```
    if (myclock[1].tv_nsec >= myclock[0].tv_nsec) {
        temp = myclock[1].tv_sec - myclock[0].tv_sec;
        temp_n = myclock[1].tv_nsec - myclock[0].tv_nsec;
        timedelay = BILLION * temp + temp_n;
    } else {
        temp = myclock[1].tv_sec - myclock[0].tv_sec - 1;
        temp_n = BILLION + myclock[1].tv_nsec - myclock[0].tv_nsec;
        timedelay = BILLION * temp + temp_n;
    }
```

Analysis in detail (cont'd)

Example case 1

- Let's say `myclock[1]` captured 3.7 second, and `myclock[0]` captured 1.2 second.
  - `myclock[1]`
    - ✓ `.tv_sec == 3; .tv_nsec == 700,000,000`
  - `myclock[0]`
    - ✓ `.tv_sec == 1; .tv_nsec == 200,000,000`
  - `temp = 3 – 1; temp_n = 700,000,000 - 200,000,000`
  - `timedelay = 2 * BILLION + 500,000,000`

    `= 2,500,000,000 nanoseconds`

```
if (myclock[1].tv_nsec >= myclock[0].tv_nsec) {
    temp = myclock[1].tv_sec - myclock[0].tv_sec;
    temp_n = myclock[1].tv_nsec - myclock[0].tv_nsec;
    timedelay = BILLION * temp + temp_n;
}
```

Analysis in detail (cont'd)

❖ Example case 2

- Let's say `myclock[1]` captured 3.4 second, and `myclock[0]` captured 1.8 second.
  - `myclock[1]`
    - ✓ `.tv_sec == 3; .tv_nsec == 400,000,000`
  - `myclock[0]`
    - ✓ `.tv_sec == 1; .tv_nsec == 800,000,000`
  - `temp = (3 – 1) – 1`
  - `temp_n = BILLION + (400,000,000 - 800,000,000)`
  - `timedelay = 2 * BILLION - 400,000,000`

```
            = 1 * BILLION + (BILLION – 400,000,000)
            = 1,600,000,000 nanoseconds
```

```
else {
    temp = myclock[1].tv_sec - myclock[0].tv_sec - 1;
    temp_n = BILLION + myclock[1].tv_nsec - myclock[0].tv_nsec;
    timedelay = BILLION * temp + temp_n;
}
```

Analysis in detail (cont'd)

calclock.c

- Finally, do an atomic add operation for each of `total_time` and `total_count`
  - `total_time` will be increased by `timedelay`
  - `total_count` will be increased by 1

```
    __sync_fetch_and_add(total_time, timedelay);
    __sync_fetch_and_add(total_count, 1);

    return timedelay;
}
```

Printing out the result

- **After all, let's print out the result**
  - Make sure that you don't put any `printk()` into the profiling code, since the cost of printing out is very expensive in terms of time.

```
unsigned long long count, time;

void print_result(void)
{
    printk("some_time_consuming_function is called %llu times, \
        and the time interval is %lluns\n", count, time);
}
```

Printing out the result

- **If you are inside the kernel module, consider printing out the results right before exiting the module**
  - To prevent from any `printk()` bothering us

```
...
extern unsigned long long count, time;

static void __exit happy_exit(void)
{
    ...
    /* Various exit stuff goes here … */
    ...
    printk("some_time_consuming_function is called %llu times, \
        and the time interval is %lluns\n", count, time);
}

...

module_exit(happy_exit)
...
```

Printing out the result (optional)

- **If you are too tired of dealing with large numbers, consider separating with commas**
  - Example codes are below

```
/**
 * seperate_num() - Make number as separated with commas.
 * @number: Input number.
 * @buffer: Number string buffer.
 *
 * Return: Separated number string buffer itself.
 */
static const char *
seperate_num(unsigned long long number, char buffer[])
{
    char tmp_buff[100]; // temp buffer for characterized numbers
    char tmp_reverse_buff[100]; // temp buffer for saving reversed numbers
    int cur, counter = 0, rvs_cur = 0;
```

Printing out the result (optional) cont'd

```
    sprintf(tmp_buff, "%llu", number);
    cur = strlen(tmp_buff);

    for (--cur; cur > -1; cur--) {
        if (counter == 3) {
            tmp_reverse_buff[rvs_cur++] = ',';
            counter = 0;
            cur++;
        }
        else {
            tmp_reverse_buff[rvs_cur++] = tmp_buff[cur];
            counter++;
        }
    }

    cur = 0;
    for (--rvs_cur; rvs_cur > -1; rvs_cur--) {
        buffer[cur++] = tmp_reverse_buff[rvs_cur];
    }

    buffer[cur] = '\0'; // inserting null char, EOL

    return buffer;
}
```

Printing out the result (optional)

- **Defining custom dump function makes output neat**
  - Example codes are below

```
extern unsigned long long total_time;

static void
printout(int depth, char *func_name, unsigned long long count, unsigned long long time)
{
    char char_buff[100]; // buffer for characterized numbers
    int percentage;

    if (!total_time)
        total_time = 1;
    percentage = time * 100 / total_time;

    printk("%s", "");

    while(depth--)
        printk(KERN_CONT "\t");
    printk(KERN_CONT "%s is called ", func_name);
    printk(KERN_CONT "%s times, ", seperate_num(count, char_buff));
    printk(KERN_CONT "and the time interval is %sns", seperate_num(time, char_buff));
    printk(KERN_CONT " (%d%%)\n", percentage);
}
```

Printing out the result (optional)

- **Defining custom dump function makes output neat**
  - Example Usage

```
extern unsigned long long file_write_iter_time, file_write_iter_count; // this is total_time
extern unsigned long long __generic_file_write_iter_time, __generic_file_write_iter_count;
extern unsigned long long generic_perform_write_time, generic_perform_write_count;

static void __exit pxt4_exit_fs(void)
{
    ...
    printout(1, "file_write_iter", file_write_iter_count, file_write_iter_time);
    printout(2, "__generic_file_write_iter",
        __generic_file_write_iter_count, __generic_file_write_iter_time);
    printout(3, "generic_perform_write", generic_perform_write_count, generic_perform_write_time);
   ...
}
...
module_exit(pxt4_exit_fs)
```

Printing out the result (optional)

- **Defining custom dump function makes output neat**
  - “`sudo dmesg`” output results

```
file_write_iter is called 24,576,000 times, and the time interval is 1,217,567,807,263ns (per thread is 19,024,496,988ns) (100.0%)
    __generic_file_write_iter is called 24,576,000 times, and the time interval is 1,195,832,166,346ns (per thread is 18,684,877,599ns) (98.21%)
        generic_perform_write is called 24,576,000 times, and the time interval is 1,173,845,919,119ns (per thread is 18,341,342,486ns) (96.40%)
```

getrawmonotonic

- Reads the current time when you are in the **kernel space**

```
#include <linux/ktime.h>

void getrawmonotonic(struct timespec *ts);
```

  - @ts: timespec where to save the current time

- **Reference Link:**
  - https://docs.kernel.org/core-api/timekeeping.html

clock_gettime

- Reads the current time when you are in the user space

```
#include <time.h>
int clock_gettime(clockid_t clockid, struct timespec *tp);
```

  - @clockid: the identifier of the particular clock on which to act. Set this to `CLOCK_MONOTONIC_RAW` to get a raw hardware-based time
  - @tp: timespec where to save the current time

- Example

```
struct timespec myclock[2];
clock_gettime(CLOCK_MONOTONIC_RAW, &myclock[0]);
function_to_profile(...);
clock_gettime(CLOCK_MONOTONIC_RAW, &myclock[1]);
calclock(myclock, &time, &count)
```

- Reference Link:
  - https://man7.org/linux/man-pages/man3/clock_gettime.3.html