COSC 301: Operating Systems

Lab 3: Measuring the cost of a system call

In this lab, you'll empirically measure the costs of a system call, and speculate on how you might measure the cost of a context switch.

Measuring the cost of a system call is *relatively* easy. For example, you could repeatedly call a really simple system call (e.g., performing a 0-byte read), and time how long it takes; dividing the time by the number of iterations gives you a rough estimate of the cost of a system call.

One thing you'll have to take into account is the precision and accuracy of your timer. A typical timer that you can use is `gettimeofday()`; a short example of using this call is given below, but you should also consult the `man` page. What you'll see there is that `gettimeofday()` returns the time in microseconds since 1970; however, this does not mean that the timer is precise to the microsecond. Measure back-to-back calls to `gettimeofday()` to learn something about how precise the timer really is; this will tell you how many iterations of your null system-call test you'll have to run in order to get a good measurement result. If `gettimeofday()` is not precise enough for you, you might look into using the rdtsc instruction available on x86 machines (a short example is given below). This instruction reads the current value of a cycle timer; you'll have to convert the results to seconds yourself of course.

You should measure three separate system calls; which three you measure is entirely up to you. Here are just some of the possibilities: `stat()`, `read()`, `write()`, `getpid()`, `gethostname()`, `gettimeofday()`, `pipe()`, `fork()`, `signal()`, and `select()` (or `poll()`). Any of these should be relatively straightforward to use (note that with `read()` and `write()`, you'll need to open a file first - you can try to read zero bytes from a file, or write zero bytes to `/dev/null`, a special "black hole" file).

When you have computed the average time for each of your three system calls, write the average cost on the board (we'll create a big table).

Before you leave the lab

1. Demo/show your code to me
2. Turn your code in via Moodle (it should be short!)
3. Wait until we've discussed the results on the board

Example of using `getttimeofday` and `rdtsc` assembly call

An example using the `gettimeofday` system call for timestamping:

```
#include <stdio.h>
#include <sys/time.h>
#include <time.h>

int main(int argc, char **argv)
{
    struct timeval begin, end, diff;
    gettimeofday(&begin, NULL);
    gettimeofday(&end, NULL);
    timersub(&end, &begin, &diff); // timersub is a macro defined in sys/time.h
                                   // there's also timeradd and some others
    printf("begin timestamp: %d.%06d\n", begin.tv_sec, begin.tv_usec);
```

```
    printf("end timestamp  : %d.%06d\n", end.tv_sec, end.tv_usec);
    printf("difference     : %d.%06d\n", diff.tv_sec, diff.tv_usec);
    return 0;
}
```

An example using the rdtsc assembly instruction for very high-precision timestamping:

```
/* using the x86 cycle timer (rdtsc) to obtain a quasi-timestamp */
#include <stdio.h>

/* macro to use rdtsc assembly instruction */
#define rdtscll(val) \
     __asm__ __volatile__ ("rdtsc" : "=A" (val))

int main(int argc, char **argv)
{
    /* long long is a 64-bit integer */
    long long begin = 0LL, end = 0LL, diff = 0LL;
    rdtscll(begin);
    rdtscll(end);
    diff = end - begin;
    printf("cycle timer begin: %lld\n", begin);
    printf("cycle timer end  : %lld\n", end);
    printf("difference       : %lld\n", diff);
    return 0;
}
```