Fundamentos de Sistemas de Operação

MIEI 2019/2020

Laboratory session 5

Objectives

Evaluate the time it takes to make different system calls.

1. Introduction

Start by measuring the average time it takes to call a function in your program. For this we measure the time to accomplish several function calls and then calculate the average, as shown in the following program:

```
#include <sys/time.h>
#include <stdio.h>
#define NTRIES  1000

int do_something(void) { return 1; }

int main (int argc, char *argv[])
{
    int i, p;
    long elapsed;
    struct timeval t1,t2;

    gettimeofday(&t1, NULL);
    for (i = 0;  i < NTRIES; i++)
        p = do_something();  // code to evaluate
    gettimeofday(&t2, NULL);
    elapsed = ((long)t2.tv_sec - t1.tv_sec) * 1000000L + (t2.tv_usec - t1.tv_usec);
    printf ("Elapsed time = %6li us (%g us/call)\n", elapsed, (double)elapsed/NTRIES);
    return 0;
}
```

A system call usually takes more time than a regular function call. Each system call can take more or less time depending on the actions the kernel must complete internally before returning a reply to the processes making the call. Change *do_something* to measure one of the simplest system calls, the *getuid* (this call returns the identifier of the user that launched the process). Why the time here is much greater than the one spent when calling a function?

2. Process creation, termination and switching

Perform the following experiments where the time taken by different system and library calls related with process management is measured.

1. fork + wait system call – evaluate the time to create and terminate a new process. Remember the example in figure 5.2 of the recommended book. Please note that the child process should exit immediately.

2. fork+exec+wait system calls – now evaluate the time to execute and terminate a new program. Remember the example in figure 5.3. For this test compile and execute from your program, with *execvp*, the following program:

```
int main( int argc, char *argv[] ) { return 0; }
```

Compare the times between test 3 and test 4 and explain the differences.

3. **pthread_create + pthread_join library calls** – now evaluate the time to execute and terminate a new thread. The organization of the code should be as in test 4. The created threads should execute the code:

```
void *func( void *arg ) { return NULL; }
```

Compare the times between test 4 and test 5 and explain the differences.

4. Measuring the switching time between lightweight processes using sched_yield Consider the program below that uses the system call *sched_yield( ).* When a process invokes this system call, it voluntarily releases the CPU, allowing the operating system to schedule another process. See the manual page of *sched_yield( ).*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sched.h>
#include <pthread.h>
void * xxx1(void *t){
       int i;
       long elapsed;
       struct timeval t1,t2;
       int ntimes = (int)t;
       gettimeofday(&t1, NULL);
       for(i=0; i< ntimes; i++){
              sched_yield(); }
       gettimeofday(&t2, NULL);
       elapsed = ((long)t2.tv_sec-t1.tv_sec)*1000000L + (t2.tv_usec-t1.tv_usec);
       printf ("Elapsed time = %6li us (%g us/process switch)\n", elapsed, (double)elapsed/(2*ntimes));
       return NULL;
}
void * xxx2(void * t){
       int i;
       int ntimes = (int)t;
       for(i=0; i< ntimes; i++){
              sched_yield(); }
         return NULL;
}
int main(){
       pthread_t p1, p2;
       pthread_create(&p1,NULL, xxx1, (void *)100000);
       pthread_create(&p2,NULL, xxx2, (void *)100000);
       pthread_join( p1, NULL);
       pthread_join( p2, NULL);
       return 0;
```

Study the code and run it. Try to understand why the obtained result is an approximation of the time overhead that corresponds to switching between (lightweight) processes. What are the aspects that contribute to the difference between the real process switching time and the one measured in this experiments.

5. Measuring the switching time between processes created with fork() using sched_yield Modify the program above for measuring the context switching time when processes are created using *fork( ).*

6. Another way to estimate process switching time In the program below, the initial process creates two pipes *fd1* and *fd2*; after that, it creates two child processes P1 and P2. Pipe *fd1* is used for transferring bytes from P1 to P2 and pipe *fd2* is used in the opposite direction. Why do we need two pipes?

```
char c = 'X';
void pipe1(int inp,int out, int ntimes){
       // inp is the channel used to read from the pipe
       // out is the channel used to write into the pipe int i; char l;
       long elapsed;
       struct timeval t1,t2;
       gettimeofday(&t1, NULL);
       for(i=0; i< ntimes; i++){
              // code to fill in, that sends 'X' to the other process and receives the same byte from it
       }
       gettimeofday(&t2, NULL);
       elapsed = ((long)t2.tv_sec - t1.tv_sec) * 1000000L + (t2.tv_usec - t1.tv_usec);
       printf ("Elapsed time = %6li us (%g us/call)\n", elapsed, (double)elapsed/(2*ntimes));
}

void pipe2(int inp, int out, int ntimes){
       // inp is the channel used to read from the pipe
```

```
        // out is the channel used to write into the pipe int i; char l;
        for(i=0; i< ntimes; i++){
                // code to fill in that receives a byte from the other process and echoes it
        }
}

int main(){
        int fd1[2]; int fd2[2];
        pipe(fd1); pipe(fd2);
        int p = fork();
        if(p==0){
                // replace ? by the appropriate channels
                pipe2(? , ? , 100000 );
                exit(0);
        }else{
                p = fork();
                if(p==0){
                        // replace ? by the appropriate channels
                        pipe1(?, ?, 100000);
                        exit(0);
                }
                wait(NULL); wait(NULL);
        }
}
```

Complete the code as asked and run it. Why the obtained result is an approximation of the time overhead that corresponds to switching between (heavyweight) processes? Is this approach better or worse than the one used before? Explain why.

3. Input/output using system calls and stdio library

I/O programming using the C standard library – change *do_something* to evaluate the time it takes to write some text to screen in the following two cases:

1. `printf("writing… ");`
2. `printf("writing… "); fflush(stdout);`

Explain why case 2 is slower than case 2.

File copy using the C standard library

Look at the code in `fcopia.c`, a program that uses the standard C functions for the I/O operations, namely `fopen`, `fread` and `fwrite`. The command can be used like:

```
fcopia 100 FILE1 FILE2
```

to create a replica of file `FILE1` with name `FILE2` (copying 100 bytes at a time).

Use the `time` command to obtain the time that each program takes to copy a big file. For that purpose, place the `time` command before your own. Example:

```
time fcopia 100 file1 newfile

real 0m0.175s

user 0m0.001s sys 0m0.009s
```

Use an existing file with at least 10Mbytes (you can use any one or create a new one for testing using a command like: `dd if=/dev/zero bs=1M count=10 of=file1`). Execute your program with different block sizes (1, 128, 1024, 10240). Also count the number of system calls using *strace* command for the `fcopia` and `copia` programs:

```
strace –c fcopia 100 file1 newfile
```

Compare the time spent by executions of the program with each block size

File copy using the read and write system calls

Consider the program **`copia.c`**. that has the same functionality and usage of `fcopia.c` . This version uses only Unix's system calls C interface, `open`, `read`, `write` and `close` for the I/O operations.

Repeat the file copies of previous sections, Compare the performance of `fcopia` and `copia`, with distinct block sizes.. Justify the results and performance discrepancies.