HW#2
Advanced Operating Systems, Fall 2024

Lee Ming Fa (CBB110213)
Department of Computer Science and Information Engineering
National Pingtung University

1. You have to write C programs for measuring the costs of any 5 system calls.

Solution: Please refer to List 1 (q1.c):

Listing 1: q1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>
#include <sys/wait.h>

#define iterationsForSystemCall 100000
#define toNarrowSeconds 1000000000

long getTimeInterval(struct timespec start, struct timespec end) {
    return (end.tv_sec - start.tv_sec) * toNarrowSeconds + (end.tv_nsec - start.tv_nsec);
}

void measure_time_getpid() {
    struct timespec start, end;

    clock_gettime(CLOCK_MONOTONIC, &start);
    for (int i = 0; i < iterationsForSystemCall; i++) {
        getpid();
    }
    clock_gettime(CLOCK_MONOTONIC, &end);
    printf("getpid() average time: %ld ns\n", getTimeInterval(start, end) / iterationsForSystemCall);
}

void measure_time_read() {
    int fd = open("./tempText.txt", O_RDONLY);
    if (fd < 0) {
        perror("open failed");
        return;
    }

    struct timespec start, end;
    char buffer[1];

    clock_gettime(CLOCK_MONOTONIC, &start);
    for (int i = 0; i < iterationsForSystemCall; i++) {
        read(fd, buffer, 0);
    }
    clock_gettime(CLOCK_MONOTONIC, &end);
    close(fd);
    printf("read() average time: %ld ns\n", getTimeInterval(start, end) / iterationsForSystemCall);
}

void measure_time_write() {
    int fd = open("./tempText.txt", O_WRONLY);
    if (fd < 0) {
        perror("open failed");
        return;
    }
```

```c

    struct timespec start, end;
    char buffer[1] = {0};

    clock_gettime(CLOCK_MONOTONIC, &start);
    for (int i = 0; i < iterationsForSystemCall; i++) {
        write(fd, buffer, 0);
    }
    clock_gettime(CLOCK_MONOTONIC, &end);
    close(fd);
    printf("write() average time: %ld ns\n", getTimeInterval(start, end) /
        iterationsForSystemCall);
}

void measure_time_open_close() {
    int file_directory[iterationsForSystemCall];
    long totalTimeForOpenAndClose, totalTimeForClose;
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);
    for (int i = 0; i < iterationsForSystemCall; i++) {
        file_directory[i] = open("./tempText.txt", O_RDONLY);
        if (file_directory[i] < 0) {
            perror("open failed");
        }
        close(file_directory[i]);
    }
    clock_gettime(CLOCK_MONOTONIC, &end);
    totalTimeForOpenAndClose = getTimeInterval(start, end);
    clock_gettime(CLOCK_MONOTONIC, &start);
    for (int i = 0; i < iterationsForSystemCall; i++) {
        close(file_directory[i]);
    }
    clock_gettime(CLOCK_MONOTONIC, &end);
    totalTimeForClose = getTimeInterval(start, end);
    printf("open() average time: %ld ns\n", (totalTimeForOpenAndClose -
        totalTimeForClose) / iterationsForSystemCall);
    printf("close() average time: %ld ns\n", (totalTimeForClose /
        iterationsForSystemCall));
}


int main() {
    printf("Measuring the average costs for five different system calls. :)~YA\n");

    measure_time_getpid();
    measure_time_read();
    measure_time_write();
    measure_time_open_close();

    return 0;
}
```

Its execution results are as follows:

```
$ cc q1.c
$ ./a.out
Measuring the average costs for five different system calls. :)~YA
getpid() average time: 3 ns
read() average time: 837 ns
```

```
write() average time: 980 ns
open() average time: 14293 ns
close() average time: 626 ns
```

As we can see in lines 11 to 13 of List 1, I declare a function called `getTimeInterval` that calculates the time interval for each system call. Specifically, in line 11, I define two variables: `struct timespec start` and `end`. These variables utilize the `timespec` structure, which is intended for high-precision time measurements and is included in the `<time.h>` header file. In line 12, I represent the time interval in nanoseconds. This choice is crucial because calculating the difference using only (`end.tv_sec` − `start.tv_sec`) yields a result of 0 seconds, despite the process completing. The higher precision offered by nanoseconds allows for a more accurate representation of the brief durations involved in system calls. This function calculates the time interval from the start of the system calls to their completion and returns the duration in nanoseconds.

Next, in lines 15 to 24, I introduce another function named `measure_time_getpid()`. This function performs 100,000 iterations to measure the average time taken by the `getpid()` system call. In line 18, we observe a function invocation of `clock_gettime`. This function, provided by `<time.h>`, uses the constant `CLOCK_MONOTONIC`, which ensures that the measured time is unaffected by any changes in the system clock. This characteristic makes it ideal for accurately tracking the time intervals of system calls.

Moving on to lines 26 to 43, we have a function named `measure_time_read`. This function is designed to measure the time taken for the read system call. In line 27, it opens a directory called `worldPeace` on the computer. The flag `O_RDONLY` indicates that this directory will be opened in read-only mode, meaning that it can only be read and no other actions can be performed on it. In line 34, I declare a character buffer `char buffer[1]`, which will be used to read a single character from the directory, but in reality, there is nothing in the directory. Next, in lines 45 to 62, similar to the read function, there is a function named `measure_time_write`, which is intended for writing a character to this directory.

Finally, in lines 64 to 86, there is a function named `measure_time_open_close`, which calculates the time interval for opening and closing a directory. In line 65, an integer array named `file_directory` is declared to store file descriptors, with each element corresponding to 1,000,000 iterations. In lines 68 to 76, multiple identical directories will be opened and closed to calculate the time interval. In lines 78 to 82, similar to the lines 68 to 76, but this time, the focus is solely on closing the directories. The main objective is to close the directories that were opened and calculate the time interval for them. In lines 84 and 85, I calculate the total sum of open and close operations, subtract the total sum of close operations from the total sum of open operations, and then divide the resulting total time interval by the number of iterations to output the time interval per iteration.

In the end, the `main` function calls these functions to display the results. :)

2. You have to write C programs for measuring context switch.

Solution:

Listing 2: q2.c

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

```c
#include <time.h>
#include <sys/wait.h>

#define iterationsForSystemCall 100000
#define stringSize 16
#define toNarrowSeconds 1000000000

long getTimeInterval(struct timespec start, struct timespec end) {
    return (end.tv_sec - start.tv_sec) * toNarrowSeconds + (end.tv_nsec - start.
        tv_nsec);
}

void measure_time_context_switch() {
    int pipe_fd[2];
    pipe(pipe_fd);

    struct timespec start, end, readStart, readEnd, writeStart, writeEnd;
    int pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }

    char dataForParent[stringSize] = "mingfaisgoodman";
    char dataForChild[stringSize];
    long readAndWriteTime = 0;

    if (pid == 0) {
        for (int i = 0; i < iterationsForSystemCall; i++) {
            clock_gettime(CLOCK_MONOTONIC, &readStart);
            read(pipe_fd[0], dataForChild, stringSize);
            clock_gettime(CLOCK_MONOTONIC, &readEnd);

            clock_gettime(CLOCK_MONOTONIC, &writeStart);
            write(pipe_fd[1], dataForChild, stringSize);
            clock_gettime(CLOCK_MONOTONIC, &writeEnd);

            readAndWriteTime += getTimeInterval(readStart, readEnd) + getTimeInterval
                (writeStart, writeEnd);
        }
        close(pipe_fd[0]);
        close(pipe_fd[1]);
        exit(0);
    } else {
        clock_gettime(CLOCK_MONOTONIC, &start);
        for (int i = 0; i < iterationsForSystemCall; i++) {
            write(pipe_fd[1], dataForParent, stringSize);
            read(pipe_fd[0], dataForParent, stringSize);
        }
        clock_gettime(CLOCK_MONOTONIC, &end);

        wait(NULL);
        close(pipe_fd[0]);
        close(pipe_fd[1]);

        long totalContextSwitchTime = getTimeInterval(start, end) - readAndWriteTime;
        printf("Context switch average time: %ld ns\n", totalContextSwitchTime / (
            iterationsForSystemCall * 2));
    }
```

```c
61 }
62
63 int main() {
64     measure_time_context_switch();
65     return 0;
66 }
```

Its execution results are as follows:

```
$ cc q2.c
$ ./a.out
Context switch average time: 2425 ns
```

In lines 16 to 61, I introduce a function named `measure_time_context_switch`. This function is designed to measure the average time interval for 100,000 iterations of context switching between a parent and child process using pipe-based communication.

In lines 17 to 18, a pipeline is set up using `pipe(pipe_fd)`. In line 20, I declare six `struct_timespec` variables: `start`, `end`, `readStart`, `readEnd`, `writeStart`, and `writeEnd`, to measure the time intervals. In line 21, the `fork` function is called to create a child process, storing the PID in `pid`.

In lines 27 to 28, two arrays, `dataForParent` and `dataForChild`, are declared for use by the parent and child processes, respectively.

In lines 31 to 46, the child process repeatedly reads and writes from the pipe for 100,000 iterations. The process reads data from the pipe via `read(pipe_fd[0], dataForChild, stringSize)` and writes it back using `write(pipe_fd[1], dataForChild, stringSize)`. In lines 33 and 35, I use `clock_gettime` to get the time cost of the `read` command. Similarly, in lines 37 and 39, `clock_gettime` is used to measure the time cost for the `write` command.

In lines 46 to 60, the parent process calculates the total context switch time interval by reading and writing through the pipe. In line 54, the `wait(NULL)` function ensures that the child process finishes execution before proceeding. Lines 55 and 56 are used to close the pipeline. In line 58, a variable called `totalContextSwitchTime` is declared, which stores the context switch time by subtracting the `readAndWriteTime` from the total time.

Finally, in line 59, the average time for each context switch is printed. Since there are two processes involved (parent and child), the total time interval is divided by twice the number of iterations to get the average context switch time per iteration.

In the `main` function, `measure_time_context_switch` is called to display the output value.