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;
    }
```

```
51
52      struct timespec start, end;
53      char buffer[1] = {0};
54
55      clock_gettime(CLOCK_MONOTONIC, &start);
56      for (int i = 0; i < iterationsForSystemCall; i++) {
57          write(fd, buffer, 0);
58      }
59      clock_gettime(CLOCK_MONOTONIC, &end);
60      close(fd);
61      printf("write() average time: %ld ns\n", getTimeInterval(start, end) /
            iterationsForSystemCall);
62  }
63
64  void measure_time_open_close() {
65      int file_directory[iterationsForSystemCall];
66      long totalTimeForOpenAndClose, totalTimeForClose;
67      struct timespec start, end;
68      clock_gettime(CLOCK_MONOTONIC, &start);
69      for (int i = 0; i < iterationsForSystemCall; i++) {
70          file_directory[i] = open("./tempText.txt", O_RDONLY);
71          if (file_directory[i] < 0) {
72              perror("open failed");
73          }
74          close(file_directory[i]);
75      }
76      clock_gettime(CLOCK_MONOTONIC, &end);
77      totalTimeForOpenAndClose = getTimeInterval(start, end);
78      clock_gettime(CLOCK_MONOTONIC, &start);
79      for (int i = 0; i < iterationsForSystemCall; i++) {
80          close(file_directory[i]);
81      }
82      clock_gettime(CLOCK_MONOTONIC, &end);
83      totalTimeForClose = getTimeInterval(start, end);
84      printf("open() average time: %ld ns\n", (totalTimeForOpenAndClose -
            totalTimeForClose) / iterationsForSystemCall);
85      printf("close() average time: %ld ns\n", (totalTimeForClose /
            iterationsForSystemCall));
86  }
87
88
89  int main() {
90      printf("Measuring the average costs for five different system calls. :)~YA\n");
91
92      measure_time_getpid();
93      measure_time_read();
94      measure_time_write();
95      measure_time_open_close();
96
97      return 0;
98  }
```

Its execution results are as follows:

```
1 $ cc q1.c
2 $ ./a.out
3 Measuring the average costs for five different system calls. :)~YA
4 getpid() average time: 3 ns
5 read() average time: 837 ns
```

```
6 write() average time: 980 ns
7 open() average time: 14293 ns
8 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 averageReadWriteTime, averageProcessTime;

void measure_time_readAndWrite() {
    int pipe_fd[2];
    pipe(pipe_fd);
    struct timespec start, end;
    char pipeWrite[stringSize] = "mingfaisgoodman";
    char pipeRead[stringSize] = "mingfaisgoodman";
    long readAndWriteTime = 0;
    clock_gettime(CLOCK_MONOTONIC, &start);
    for (int i = 0; i < iterationsForSystemCall; i++) {
        write(pipe_fd[1], pipeWrite, stringSize);
        read(pipe_fd[0], pipeRead, stringSize);
    }
    clock_gettime(CLOCK_MONOTONIC, &end);

    readAndWriteTime = (end.tv_sec - start.tv_sec) * toNarrowSeconds + (end.tv_nsec -
         start.tv_nsec);
    close(pipe_fd[0]);
    close(pipe_fd[1]);
    averageReadWriteTime = readAndWriteTime / (iterationsForSystemCall);
    printf("Read&Write average time: %ld ns\n", averageReadWriteTime);
}

void measure_time_context_switch() {
    int pipe_fd[2];
    pipe(pipe_fd);
    struct timespec start, end;
    int pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    char dataForParent[stringSize] = "mingfaisgoodman";
    char dataForChild[stringSize] = "mingfaisgoodman";
    if (pid == 0) {
        for (int i = 0; i < iterationsForSystemCall; i++) {
            read(pipe_fd[0], dataForChild, stringSize);
            write(pipe_fd[1], dataForChild, stringSize);
        }
        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]);
```

```
63          long ProcessTime = ( (end.tv_sec - start.tv_sec) * toNarrowSeconds + (end.
                tv_nsec - start.tv_nsec) );
64          averageProcessTime = ProcessTime / (iterationsForSystemCall * 2);
65          printf("Process average time: %ld ns\n", averageProcessTime);
66      }
67  }
68
69  int main() {
70      measure_time_readAndWrite();
71      measure_time_context_switch();
72      printf("Context switch average time: %ld ns\n", averageProcessTime -
            averageReadWriteTime);
73      return 0;
74  }
```

Its execution results are as follows:

```
1 $ cc q2.c
2 $ ./a.out
3 (base) limingfa@MacBook-Pro Mechanism-Limited-Direct-Execution % ./a.out
4 Read&Write average time: 1893 ns
5 Process average time: 2172 ns
6 Context switch average time: 279 ns
```

In lines 8 to 10, I defined the constants used in the program. The basic iterations for calculating the average time interval will be 100,000 times, and `stringSize` for the read and write command will be 16. `toNarrowSeconds` will be used to convert seconds to nanoseconds for more accurate time interval calculations.

In line 12, I declared two global variables named `averageReadWriteTime` and `averageProcessTime` to store the average times for read/write operations and the process for parent-child communication, respectively.

In lines 14 to 33, I declared a function called `measure_time_readAndWrite`. This function is designed to measure the time interval of the read and write commands using `clock_gettime()`. In lines 15 and 16, I declared the `pipe_fd` array for calculating the time interval. In lines 18 and 19, I declared two variables, `pipeRead` and `pipeWrite`, to pass data through the pipe and measure the time for read and write commands. In line 20, I declared a variable called `readAndWriteTime` to store the time interval of the read and write operations. Lines 21 to 26 calculate the time interval using `clock_gettime`. Lines 29 and 30 close the pipe, and in line 31, `averageReadWriteTime` is calculated by dividing the total time by the number of iterations.

In lines 36 to 69, I declared a function called `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 36 and 37, a pipe is set up using `pipe(pipe_fd)`. In line 38, I declared two `struct timespec` variables: `start` and `end` to measure the time intervals. In line 39, the `fork()` function is called to create a child process. In the child process (when `pid == 0`), the pipe is used to read and write data for 100,000 iterations, using the `read()` and `write()` system calls. The time for each operation is recorded using `clock_gettime()`. In the parent process, the total time for both the read and write operations, including context switching, is measured using `clock_gettime()`.

In lines 52 to 66, the parent process calculates the total time interval for reading, writing, and context switching through the pipe. In line 60, the `wait(NULL)` function ensures that the child process finishes

before proceeding. Lines 61 and 62 close the pipe. In line 63, the variable `ProcessTime` is declared to store the total time, including both the parent and child process times. In lines 64 and 65, the process time is divided by `iterationsForSystemCall * 2` for more accurate results, and the process average time is printed in line 65.

In the `main` function, `measure_time_readAndWrite` and `measure_time_context_switch` are called to display the output values. Finally, in line 72, the result of the average context switch time is printed. It is calculated by subtracting `averageReadWriteTime` from `averageProcessTime`.