Homework 04

CS307-Operating System (D), Chentao Wu, Spring 2020.

Name: 方泓杰(Hongjie Fang) Student ID: 518030910150 Email: galaxies@sjtu.edu.cn

- (4.1) Provide three programming examples in which multi-threading provides better performance than a single-threaded solution.

  Solution. Here are three examples.

  - **(Program Execution)** If a program can be executed in parallel, then we can create some threads to execute different parts of the program (or the same program based on different data) in parallel, which can optimize its time complexity and provide better performance than a single-thread solution. (*A more specific example is like divide-and-conquer algorithm, if we execute several divided parts in separate threads in parallel, then its time complexity will be optimized and its performance will be better*).
  - **(Server)** If many users send requests to a server, the server may handle each request in a separate thread. (*A more specific example is the Canvas platform, if the platform has multiple threads to serve some students' requests in the same time, then its performance will be better.*)
  - **(Devices Interaction)** If an application program needs to interact with different devices, then it can open a separate thread to handle each devices. (*A more specific example is the video player program, If it has different threads such as a thread for audio, a thread for monitor, a thread for decoding, etc., then its performance will be better.*)

  □

- (4.4) What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?

  Solution. The two differences are as follows.

  - **(Awareness of kernel)** Kernel does not know about user-level threads, but it is aware of kernel-level threads.
  - **(Management)** Kernel-level threads are managed directly by kernel, but user-level threads are managed by kernel library.

  Here are some circumstances that one type is better than the other.

  - **(Kernel-level threads are better than user-level threads)** If we use a multi-processor or multi-core computer, then kernel-level threads may be better than user-level threads, because different kernel-level threads can be executed in different processors in parallel while different user-level threads in the same process can only be executed in one processor.
  - **(User-level threads are better than kernel-level threads)** If we use a time-sharing kernel, the user-level threads may be better than the kernel-level threads, because the time cost of the context switch between threads in kernel mode is much larger than its in user mode.

  □

- (4.10) Which of the following components of program state are shared across threads in a multi-threading process?

  a. Register values

  b. Heap memory

  c. Global variable

  d. Stack memory

  Solution. The heap memory and global variables are shared across threads in a multi-threading process, since the heap and the global variables are allocated in the process, but stack memory and register values are allocated in thread separately. Therefore, **b** and **c** can be shared across threads in a multi-threading process. □

- (4.17) Consider the following code segment:

```
pid_t pid;
pid = fork();
if (pid == 0) { /* child process */
    fork();
    thread_create( ... );
}
fork();
```

  a. How many unique processes are created?

  b. How many unique threads are created?

  Solution. The answers to each question are as follows.

  a. **Total 6 processes are created (including the main process).** Suppose the parent process is called P. In the first "fork()" in line 2 we create its child process, called P_1. When handling the child process P_1, we create the its child process P_2 using the "fork()" in line 4, which is the grand-child process of process P. In the third "fork()" in line 7, we will create a child process for each of the process P, P_1 and P_2, and let's call them P_3, P_4, P_5 respectively. So the whole program will create totally 6 threads $(P, P_1, P_2, P_3, P_4, P_5)$.

  b. Here are two situations.

     – Here we assume that a "fork()" instruction will duplicate all the threads in a process. Then **total 10 threads are created (including the main thread of each process).** The "thread_create()" instruction only appears once in line 5. Process P_1 and P_2 execute the "thread_create()" instruction and create an **extra** thread for both P_1 and P_2. So up to now, there is one thread in process P and there are two threads in each of processes P_1, P_2. The third "fork()" will duplicate all the threads in the three processes P, P_1 and P_2. So in P_3 there is only one thread, and there are two threads in each of the processes P_4, P_5. In conclusion, there are two threads in each of the processes P_1, P_2, P_4, P_5 and one thread in each of the processes P, P_3, so totally there are 10 threads.

- Here we assume that a "fork()" instruction only copy the current thread in a process. Then **total 8 threads are created (including the main thread of each process).** The "thread_create()" instruction only appears once in line 5. Process P_1 and P_2 execute the "thread_create()" instruction and create an **extra** thread for both P_1 and P_2. So up to now, there is one thread in process P and there are two threads in each of processes P_1, P_2. The third "fork()" will only copy the current thread in the three processes P, P_1 and P_2. So there is only one thread in each of the processes P_3, P_4, P_5. In conclusion, there are two threads in each of the processes P_1, P_2 and one thread in each of the processes P, P_3, P_4, P_5, so totally there are 8 threads.

□

- (4.19) The following program uses the Pthreads API. What would be the output from the program at LINE C and LINE P?

```
# include <pthread.h>
# include <stdio.h>

int value = 0;
void *runner (void *param); /* the thread */

int main(int argc, char *argv[]) {
    pid_t pid;
    pthread_t tid;
    pthread_attr_t attr;

    pid = fork();
    if(pid == 0) { /* child process */
        pthread_attr_init(&attr);
        pthread_create(&tid, &attr, runner, NULL);
        pthread_join(tid, NULL);
        printf("CHILD: value = \%d", value); /* LINE C */
    } else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("PARENT: value = \%d", value); /* LINE P */
    }
}

void *runner(void *param) {
    value = 5;
    pthread_exit(0);
}
```

Solution. The output from the program at LINE C will be as follows.

```
CHILD: value = 5
```

The reason is that **the global variables are shared between different threads in the same process**, so in "runner()" thread we change the global variable "value" to 5, in the main process, the global variable "value" will still be 5.

And the output from the program at LINE P will be as follows.

```
   PARENT: value = 0
```

The child process will copy the data from the parent process, so the modification of “value” in the child process have no influence on the “value” in the parent process. Therefore, the “value” in the parent process will remain to be 0. □