Threads and Concurrency

Overview

The practise questions are designed to complement and enhance your knowledge of topics covered in the lectures. Not all answers will be readily found on the lecture notes and slides, and you may be required to engage in some self-learning to complete the questions.

Suggested answers to the practise questions shall be provided at a later date. It is strongly advised that you attempt these questions on your own, and discuss them with your peers before consulting the suggested answers.

Optional questions are designed for further challenge. For these questions, answers may or may not be provided.

Practise Questions

1. Provide two programming examples in which multithreading does not provide better performance than a single-threaded solution.

2. Which of the following components are shared across threads in a multithreaded process?
    a. Register values
    b. Heap memory
    c. Global variables
    d. Stack memory

3. Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system?

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

5. Consider the code segment in Figure 1.

```
pid t pid;

pid = fork();

if (pid == 0) { /* child process */
    fork();
    thread create( . . .);
}

fork();
```

Figure 1

a. How many unique processes are created?
b. How many unique threads are created?

6. The program shown in Figure 2 uses the POSIX 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) {
        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) {
        wait(NULL);
        printf("PARENT: value = %d", value); /* LINE P */
    }
}

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

Figure 2

Optional Questions

7. A CPU-scheduling algorithm determines an order for the execution of its scheduled processes. Given *n* processes to be scheduled on one processor, how many different schedules are possible? Give a formula in terms of *n*.

8. The traditional UNIX scheduler enforces an inverse relationship between priority numbers and priorities, i.e., the higher the number, the lower the priority. The scheduler recalculates process priorities once per second using the following function

$$\text{priority} = \left(\frac{\text{recent\_cpu\_usage}}{2}\right) + base$$

Where *base* = 60 and *recent_cpu_usage* refers to a value indicating how often a process has used the CPU since priorities were last recalculated. Assume that the recent CPU usage for process P1 is 40, process P2 is 18, and process P3 is 10, what will be the new priorities for these three processes when priorities are recalculated? Based on this information, does the traditional UNIX scheduler raise or lower the relative priority of a CPU-bound process?

END OF DOCUMENT