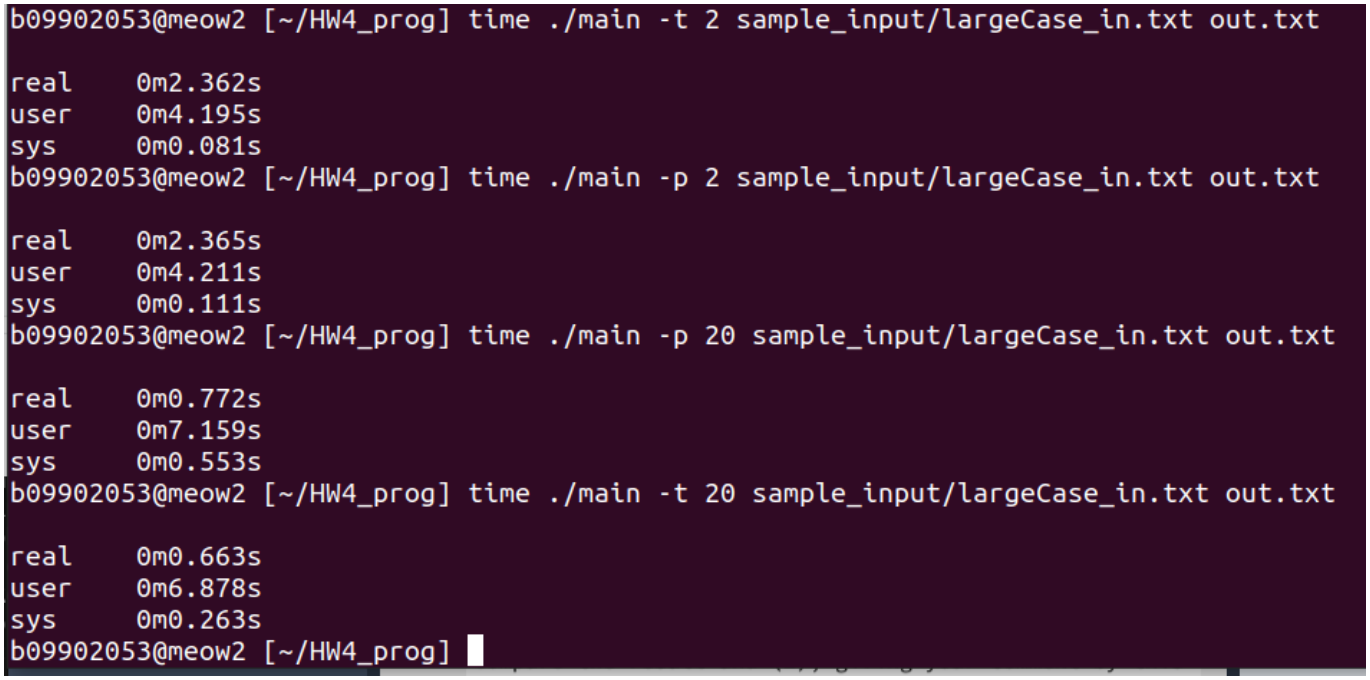
# Programming Homework 4

## 4. 6. Giving your comment by this result on the report.

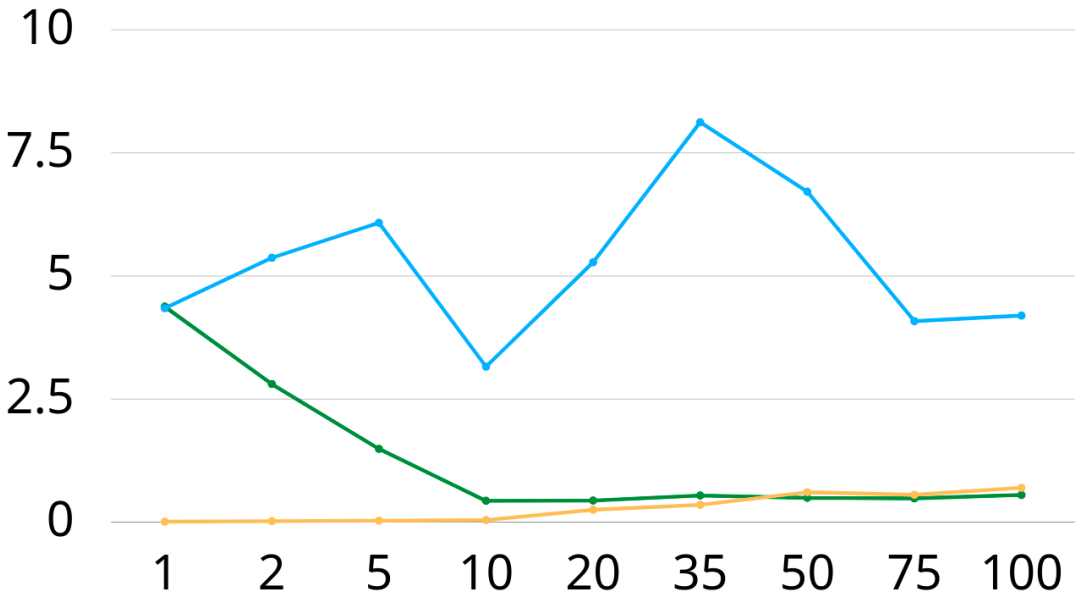



- 2 threads v.s. 20 threads
  From the screenshot, we can easily observe that running more threads significantly reduces the real time taken. The reason to this outcome is that the workload can be shared by different CPUs and more CPUs can solve the problem simultaneously with 20 threads.
  Additionally, user time and system time increase. This my be the result of more **pthread_create** and **pthread_join**, which are system calls, are called. And we spend more time on assigning jobs because the workload is divided into more subproblems.

- 2 threads v.s. 2 processes
  From the screenshot, we can see that all 3 types of time taken increases, which illustrate the fact that threads are more lightweight compared with processes.

# Programming Homework 4

**5. Use large test case to calculate execution time with different of thread ( = 1, 5, 10, . . . , 100), drawing the line graph and giving your comment.**




In the above graph, x-coordinate denotes the number of threads, y-coordinate denotes the time taken (in seconds), blue line denotes user time, green line denotes real time, orange denotes system line.

From the above chart, we can see that the user time seems to have nothing to do with the number of threads. User time is more related to system loading instead of the number of threads we assign. As for the real time, it will decline sharply when the number of threads is small. And it will reach a steady state when the number attains a certain value. Lastly, the system time increases steadily because creating threads and joining threads are system calls.

# Programming Homework 4

## Critical parts of the program

- Assign jobs

```
for (int idx = 0; idx < worknum; idx++) {
    tmp2 = tmp1 + worklen / worknum + (idx < worklen % worknum);
    jobs[idx].left = tmp1, jobs[idx].right = tmp2;
    // fprintf(stderr, "job: %d %d\n", tmp1, tmp2);
    tmp1 = tmp2;
}
```

Assign the workload into several continual segments.

- Threads  Processes

```
for (epoch_cnt = 0; epoch_cnt < epoch; epoch_cnt++) {
    for (int idx = 0; idx < n_thread; idx++) {
        pthread_create(&tid[idx], NULL, job, (void *)idx);
    }
    // fprintf(stderr, "epoch_cnt: %d\n", epoch_cnt);
    for (int idx = 0; idx < n_thread; idx++) {
        pthread_join(tid[idx], NULL);
    }
}
```

```
for (epoch_cnt = 0; epoch_cnt < epoch; epoch_cnt++) {
    for (int idx = 0; idx < n_process; idx++) {
        if (fork() == 0) {
            job((void *)idx);
            _exit(0);
        }
    }
    while (wait(NULL) > 0);
}
```

We want to wait for all the assigned jobs in the current epoch before proceeding the next generation. By applying **pthread_join** and **wait**, we can make this happen.

- The jobs

```
void *job(void *pidx) {
    int idx = (int)pidx;
    for (int fidx = jobs[idx].left; fidx < jobs[idx].right; fidx++) {
        int i = fidx / (col + 1), j = fidx % (col + 1);
        if (j == col) continue;
        char num = 0, live = '.';

        int i2l = max(i - 1, 0), i2r = min(i + 2, row), j2l = max(j - 1, 0), j2r = min(j + 2, col);

        for (int i2 = i2l; i2 < i2r; i2++) {
            for (int j2 = j2l; j2 < j2r; j2++) {
                if (get_board(i2, j2, (epoch_cnt % 2)) == 'O') num++;
            }
        }

        if (num == 3 || (get_board(i, j, (epoch_cnt) % 2) == 'O' && num == 4))
            live = 'O';
        set_board(live, i, j, (epoch_cnt + 1) % 2);
    }
}
```

One thread can obtain its work id and then get its segment and then do its job.