Simulation of Process Scheduling For Single Processor and Multi-Processor Systems

By
Richard Liu
Chris Yu
Xin Huang

Instructor:
Prof. Ming Hwa Wang
Santa Clara University

Table of Contents

2. Introduction

This project we try to find a new algorithm for process scheduling and try to simulate the process by comparing the benchmark of different algorithms for process scheduling. Process scheduling algorithm is a major component in operating systems, it will be beneficial to computer performance efficiency to find a new algorithm. We will investigate and evaluate the new algorithm and find a new algorithm that may or may not have niche uses would be very beneficial to operating system.

Each algorithm is not best for every situation. For example, longest job first is not suitable for single processor and shortest job first is not the best algorithm for multiple processes. Our goal is trying to find the optimal situation for each algorithm to be the most efficient.

The scope is we are trying to find and implement a proposed algorithm from a paper within a simulation. Because nowadays computer systems using multiple processors are the norm in industry, we will focus our scope on multiple processor scheduling.

3. Theoretical Bases and Literature Review

We did research on each algorithm and compare the pros and cons of each one. We find a proposed algorithm that would potentially be more effective than existing algorithms. This algorithm has an outstanding performance under the multiple processors condition. This algorithm behaves similar to back tracking method for solving knapsack, it's based on longest job first and its state space tree is created according to the research paper as follows: firstly we motion from root to left node until process is assigned to processor and if it is not, back tracking happens and we motion to the right node. Via path refusing and back tracking to other path we achieve to optimized solution. After construction state space tree, the path with the time approximately equals ideal time is the answer path.

As for other algorithms, they are pretty standard in textbooks or online sources, we chose to implement them because the implementation methods are readily available and they are to standardize for testing.

4. Hypothesis

With the implementation of this novel algorithm, we believe the one we are going to implement is better than some common algorithm like LPT, SPT and PSO in the multiple processors situation. Our goal is trying to implement this algorithm by using a specific

programming language. We will hope to find an unique situation for each of the algorithm so that the particular algorithm would be the most efficient in that particular situation.

5. Methodology

What we proposal to do is to write a C/C++ program that ask the user to input the data from the User interface or just command line. The input data include process id number, process arriving time, process end time etc.

In total we have 4 single-processor algorithms and 3 multi-processor algorithms. Once we input the data required, we will run all the algorithms in sequence, and compare the average running time for the algorithms and comment on our findings.

Our output will be generated in table, with listing the input parameters like process id, run time, end time, etc. The important data will be the total running time and average running time, as those results will give us an idea of how efficient the algorithms are.

We will test against our hypothesis by looking at the different input data and see if there is a way to generate scenarios that can fit each algorithm's strengths. If not, then we will have to say that only certain algorithms can be situational-superior.

6. Implementation

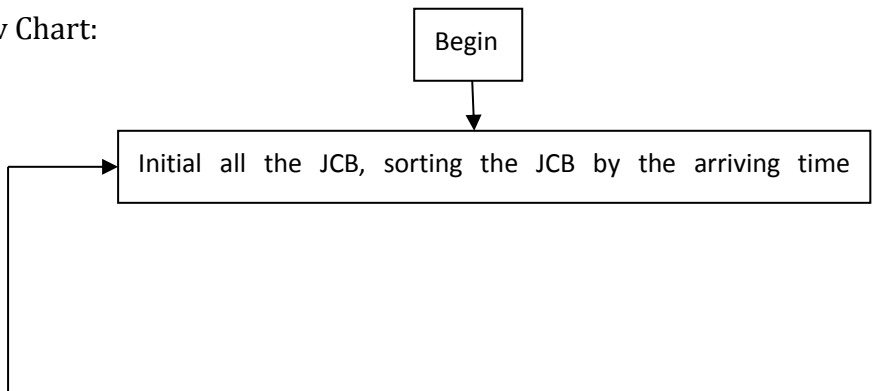
First Come First Server algorithm:
The FCFS scheduler simply executes processes to completion in the order they are submitted. We will implement FCFS using a queue data structure. Given a group of processes to run, insert them all into the queue and execute them in that order. Thus, our API will need the following methods and data structures and methods.

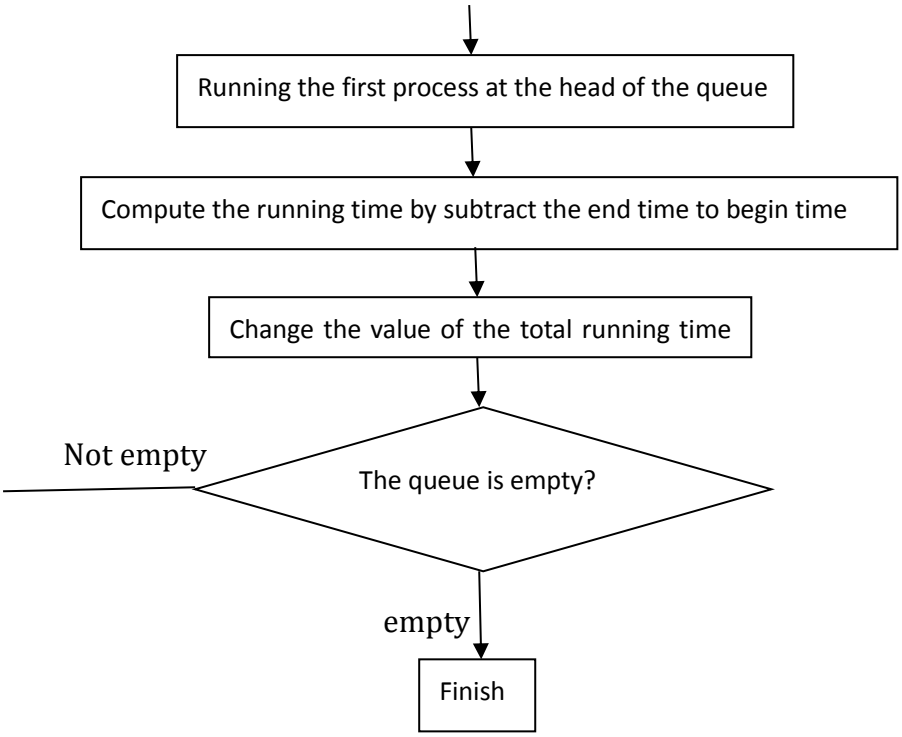
```
fifo_queue Q;                  // the queue containing the processes

add_task(proc) {               // method to add a process to the queue
   Q.add_tail(proc);           // by inserting it at the tail end
   Q.size++; // Accounting
}

reschedule()  {                // remove process from queue
    p = Q.remove_head();
    Accounting;
    return p;
}
```

Flow Chart:





Shortest Job First algorithm:
The SJF scheduler is exactly like FCFS except that instead of choosing the job at the front of the queue, it will always choose the shortest job (i.e. the job that takes the least time) available. We will use a sorted list to order the processes from longest to shortest. When adding a new process/task, we need to figure out the where in the list to insert it. Our API will need:

```
sorted_list  sl                          // sorted list data structure
add_task(proc, expected_runtime) {       // method to add a process to the list.
   sl.insert(proc, proc.runtime);        // we input the process's expected runtime
}                                        // to indicate where it should be inserted.

reschedule()  {                          // method to remove the shortest job from
   return sl.remove_head();              // the list and run it.
}
```

Priority algorithm:
A priority is associated with each process. We can think of the SJF algorithm as a special case of PRI. Processes with equal priorities may be scheduled in accordance with FCFS

PRI (L, M, H (RR)) // using Round-Robin

```
fifo_queue fQ[3]                                    // data structure

add_task(proc, pri)
        fQ[pri].add_tail(proc)

reschedule(why)
        if (why == timer)
                add_task(current, current.pri);
        set_timer(TQ);
        for pri=H to L
                if(fQ[pri].empty())
                        return fQ[pri].remove_head();
```

Round Robin algorithm:
RR is a preemptive scheduler, which is designed especially for time-sharing systems. In other words, it does not wait for a process to finish or give up control. In RR, each process is given a time slot to run. If the process does not finish, it will "get back in line" and receive another time slot until it has completed. We will implement RR using a FIFO queue where new jobs are inserted at the tail end.

```
fifo_queue fQ                   // FIFO queue

add_task (proc) {               // method to add a task
  fQ.add_tail(proc)
}

reschedule(why) {               // method to remove the next process and run it
   if (why == timer)
      add_task(current);
      set_timer(time quanta);
      return fQ.remove_head;
}
```

Knapsack algorithm:

At this approach first processor is assumed as knapsack and its state space tree is created according to research paper as follows: firstly we motion from root to left node until process is assigned to processor and if it isn't possible back tracking is happened and we motion to right node. Via path refusing and back tracking to other path we achieve to optimized solution. After construction state-space tree, the path with time approximately equals ideal time is answer path. The flow chart for these above algorithms is the same as first come first serve.

Lottery Algorithm:

This algorithm is based on handing out tickets for each process based on priority number and runtime. With higher priority and/or shorter runtime, the more tickets each process will receive. Once a ticket is drawn from the collective pool, the select process will run and eliminated from ticket pool. Draw rounds will continue until no more processes to be run. Any new processes joining will trigger a reshuffle of tickets and the odds will be adjusted accordingly.

7. Data analysis and discussion

In our project, we test the first three algorithms (FCFS, PS, SJF) under the single processor condition. From the output we can see that: SJF algorithm is the best algorithm to minimize the total running time. But under certain particular condition, because different job may have different priority, priority scheduling is best. Then we test the round robin and knapsack algorithm under the multiple processors condition. From the output we can see that round robin is best than knapsack because it can disperse the time properly to different process. Knapsack may sometimes bring the bad result if the first several processes running time is too long.

For single processor algorithms, most situations will yield the same results except for lottery. Lottery on average takes about 10 times longer when average waiting time is compared.

When running processes with a bell curve distribution (sorted by runtime), the Knapsack algorithm is slightly more efficient, it has a few seconds less of average wait time. Same results for uphill slant distribution and even distribution. KS runs into trouble when there is a sudden jump in a process's runtime. This is a situation where KS will not perform as well as the other two.

For FCFS, when processes alternate between short and long runtimes it performs the worst out of the three, having almost twice as long average waiting time.

Overall, RR is the best algorithm for multi-processor systems. When there is a sudden jump in run time it will run into trouble, however this isn't a common occurrence in systems.

Average Waiting Time

	FCFS	KS	RR
Bell Curve	17	15	16
Even Distr	30	28	30
Uphill Slant	27	25	26
Sudden Jump	14	19	14
Alternating	50	35	28
Ends and Middle	23	27	19

8. Conclusions and recommendation.

From this project we test several different kinds of process scheduling algorithm under different condition. Because different algorithm has its suitable scope, it is very hard to tell

which the best one for all conditions is. So we need to investigate more algorithms and choose the best one for that particular condition in the future research.

Knapsack algorithm has trouble with spikes in run-time, any sudden variation in process flow will slow it down significantly. As for Round Robin, most of the cases we tested did not yield much variance, in almost every case it had the least average waiting time. For FCFS, alternating run times slows down the wait time in between, and it is very vulnerable to such process flow.

Recommendations for future studies:

R. W. Schulte, Y. V. Natis, “Service Oriented Architectures”, Part 1, very good book regarding the process scheduling algorithms. For future studies, finding novel algorithms to test would be ideal. Overall, the standard algorithms cover most of the areas in terms of efficiency and optimization; most likely any new algorithms will be some sort of hybrid of preexisting ones.

9. Bibliography.

[1]M. Kranz, P. Holleis, A. Schmidt, “Embedded Interaction: Interacting with the Internet of Things”, IEEE Internet Computing, Vol. 14, pp.44-53, 2010.

[2] M. Starsinic, "System Architecture Challenges in the Home M2M Network," IEEE Applications and Technology Conference, pp. 1-7, May 2010.

[3] R. W. Schulte, Y. V. Natis, “Service Oriented Architectures”, Part 1, SPA-401- 068, Gartner Group.

[4] R. W. Schulte, Y. V. Natis, “Service Oriented Architectures”, Part 2, SPA-401- 069, Gartner Group.

[5] Newcomer, G. Lomow, Understanding SOA with Web Services(Independent Technology Guides): Addison-Wesley Professional, pp.27-28, 2008.

[6] M. N. Huhns, M. P. Singh, “Service-Oriented Computing: Key Concepts and Principles”, IEEE Internet Computing, Vol. 9, pp. 75-81,2005.

[7] M. P. Papazoglou, W. J. Heuvel, “Service Oriented Architectures: Approaches, Technologies and Research Issues”, The VLDB Journal,Vol. 16, pp. 389-415, 2007.

[8] Kreger, “Fulfilling the Web Services Promise”, Communications of the ACM, Vol. 46, pp. 29-34, 2003.

[9] Z. Shelby, “Embedded Web Services”, IEEE Wireless Communications, Vol. 17, pp. 52-57, 2010.

[10] Ning, Z. Wang, “Future Internet of Things Architecture: Like Mankind Neural System or Social Organization Framework?,” Journal IEEE Communications Letters., vol. 15, no. 4, pp. 461-463, April 2011.

10. Appendices

Program Source Code:

```
#include "process_scheduling.h"
void main(){
    int choice;
    p_input();
    //algorithm for single processor
    if(processor_number == 1){
        cout <<"Please input the single process algorithm:\n";
        cout << "1. first come first serve\n";
        cout << "2. priority scheduling\n";
        cout << "3. shortest job first\n";
        cout << "0. exit\n";
        cin >> choice;
        switch(choice){
        case 0:
            cout << "exit\n";
            break;
        case 1:
            fcfs();
            p_output();
            break;
        case 2:
            ps();
            p_output();
            break;
        case 3:
            sjf();
            p_output();
            break;
        }
    }else{// algorithm for multiple processor
        cout <<"Please input the multiple processes algorithm:\n";
        cout << "1. napsack\n";
        cout << "0. exit\n";
        cin >> choice;
        switch(choice){
        case 0:
            cout << "exit\n";
            break;
        case 1:
            double max_process_time;
```

```
                max_process_time = knapsack();
                p_output2(max_process_time);
                break;
            }
        }
        fflush(stdin);
        getchar();
}

double knapsack(){
        double total_time = 0;
        for(int i = 0; i < counter; i++){
            total_time += tasks[i].run_time;
        }
        double average_time = total_time/processor_number;
        for(int i = 0; i < counter; i++){
            for(int j = i + 1; j < counter; j++){
                if(tasks[i].run_time < tasks[j].run_time){
                    struct task_struct temp = tasks[i];
                    tasks[i] = tasks[j];
                    tasks[j] = temp;
                }
            }
        }
        double max_process_time = 0;
        double total_run_time = 0;
        for(int i = 0; i < processor_number; i++){
            total_run_time = 0;
            for(int j = 0; j < counter; j++){
                if(total_run_time + tasks[j].run_time <= average_time && tasks[j].run_flag == 0){
                    tasks[j].begin_time = total_run_time;
                    tasks[j].end_time = total_run_time + tasks[j].run_time;
                    total_run_time += tasks[j].run_time;
                    tasks[j].run_flag = 1;
                }
            }
            if(total_run_time > max_process_time){
                max_process_time = total_run_time;
            }
        }
        for(int i = 0; i < counter; i++){
            if(tasks[i].run_flag == 0){
                tasks[i].begin_time = tasks[counter-1].end_time;
                tasks[i].end_time = tasks[i].begin_time + tasks[i].run_time;
```

```
                tasks[i].run_flag = 1;
            }
        }
        return max_process_time;
}


int fcfs(){
        double time_temp = 0;
        int i, number_schedule;
        time_temp = tasks[0].arrive_time;
        for(i = 0; i < counter; i++){
            tasks[i].begin_time = time_temp;
            tasks[i].end_time = tasks[i].begin_time + tasks[i].run_time;
            tasks[i].run_flag = 1;
            time_temp = tasks[i].end_time;
            number_schedule = i;
            tasks[number_schedule].order = i+1;
        }
        return 0;
}

int ps(){
        double temp_time = 0;
        int i = 0, j;
        int number_schedule, temp_counter;
        int max_priority;
        max_priority = tasks[i].priority;
        j = 1;
        while(j < counter){
            if(tasks[j].priority>tasks[i].priority){
                max_priority = tasks[j].priority;
                i = j;
            }
            j++;
        }
        number_schedule = i;
        tasks[number_schedule].begin_time = 0;
        tasks[number_schedule].end_time = tasks[number_schedule].begin_time +
tasks[number_schedule].run_time;
        tasks[number_schedule].run_flag = 1;
        temp_time = tasks[number_schedule].end_time;
        tasks[number_schedule].order = 1;
        temp_counter = 1;
```

```
    while(temp_counter<counter){
        max_priority = 0;
        for(j = 0; j < counter; j++){
            if((tasks[j].arrive_time<=temp_time)&&(!tasks[j].run_flag)){
                if(tasks[j].priority>max_priority){
                    max_priority = tasks[j].priority;
                    number_schedule = j;
                }
            }
        }
        tasks[number_schedule].begin_time = temp_time;
        tasks[number_schedule].end_time =
tasks[number_schedule].begin_time+tasks[number_schedule].run_time;
        tasks[number_schedule].run_flag = 1;
        temp_time = tasks[number_schedule].end_time;
        temp_counter++;
        tasks[number_schedule].order = temp_counter;
    }
    return 0;
}

int sjf(){
    double temp_time = 0;
    int i = 0, j;
    int number_schedule, temp_counter;
    double run_time;
    run_time = tasks[i].run_time;
    j = 1;
    while((j<counter)&&(tasks[i].arrive_time == tasks[j].arrive_time)){
        if(tasks[j].run_time<tasks[i].run_time){
            run_time = tasks[j].run_time;
            i = j;
        }
        j++;
    }
    number_schedule = i;
    tasks[number_schedule].begin_time = tasks[number_schedule].arrive_time;
    tasks[number_schedule].end_time = tasks[number_schedule].begin_time +
tasks[number_schedule].run_time;
    tasks[number_schedule].run_flag = 1;
    temp_time = tasks[number_schedule].end_time;
    tasks[number_schedule].order = 1;
    temp_counter = 1;
    while(temp_counter<counter){
```

```
            for(j = 0; j < counter; j++){
                if((tasks[j].arrive_time<=temp_time)&&(!tasks[j].run_flag)){
                    run_time = tasks[j].run_time;
                    number_schedule = j;
                    break;
                }
            }
            for(j = 0; j < counter; j++){
                if((tasks[j].arrive_time<=temp_time)&&(!tasks[j].run_flag)){
                    if(tasks[j].run_time<run_time){
                        run_time=tasks[j].run_time;
                        number_schedule = j;
                    }
                }
            }
            tasks[number_schedule].begin_time = temp_time;
            tasks[number_schedule].end_time = tasks[number_schedule].begin_time +
tasks[number_schedule].run_time;
            tasks[number_schedule].run_flag = 1;
            temp_time = tasks[number_schedule].end_time;
            temp_counter++;
            tasks[number_schedule].order = temp_counter;
        }
        return 0;
}

int lottery()
{
        int time_temp = 0;
        float * tickets = new float[counter];
        float * ratios = new float[counter];
        float totaltickets = 0, interval = 0;
        int i, flagcounter=0;
        srand(time(NULL));

        for(i=0;i < counter;i++)
        {
            tickets[i] = (100./tasks[i].priority) + (100./tasks[i].run_time);
            totaltickets += tickets[i];
        }
        for(i=0;i<counter;i++)
        {
            interval += tickets[i]/totaltickets;
            ratios[i] = interval;
```

```
    }
    interval = 0;

    while(flagcounter < counter)
    {
        float r = (float)rand()/RAND_MAX;
        int found = 0, j=0, draw_index=0;
        while(found == 0)
        {
            if(r < ratios[j] && tasks[j].run_flag == 0)
            {
                draw_index = j;
                found = 1;
                tasks[draw_index].run_flag = 1;
                totaltickets -= tickets[draw_index];
                ratios[draw_index] = -1;
                for(i=0;i<counter;i++)
                {
                    if(tasks[i].run_flag == 0)
                    {
                        interval += tickets[i]/totaltickets;
                        ratios[i] += interval;
                    }
                }
                tasks[draw_index].begin_time = time_temp;
                tasks[draw_index].end_time = tasks[draw_index].begin_time +
                tasks[draw_index].run_time;
                time_temp += tasks[draw_index].end_time;
            }
            if(found == 1) break;

            j++;
        }
        printf("index %d found\n", draw_index);
        flagcounter++;
    }
    delete[] tickets;
    delete[] ratios;
    return 0;
}

int p_input(){ //input the parameters for the process
    cout << "Please inut the number of processor:\n";
    cin >> processor_number;
```

```
    cout << "Please input the total number of process:\n";
    cin >> counter;
    for(int i = 0; i < counter; i++){
        tasks[i].pid = (i+1);
        cout << "Please input the running time for process: " << i+1 << endl;
        cin >> tasks[i].run_time;
        cout << "Please input the priority:\n";
        cin >> tasks[i].priority;
        tasks[i].begin_time = 0;
        tasks[i].end_time = 0;
        tasks[i].order = 0;
        tasks[i].run_flag = 0;
    }
    return 0;
}

int p_output(){
    int i;
    double turn_round_time = 0, w = 0;
    cout << "pid     run_time     begin_time     end_time     priority     order
turn_round_time\n";
    for(i = 0; i < counter; i++){
        turn_round_time += tasks[i].end_time;
        cout<<tasks[i].pid<<"     "<<tasks[i].run_time<<"     "
            <<tasks[i].begin_time<<"     "<<tasks[i].end_time<<"     "<<tasks[i].priority<<"
"<<tasks[i].order<<"     "<<tasks[i].end_time<<endl;
    }
    cout << "total_running_time= " << turn_round_time << endl;
    cout << "average_running_time= " << turn_round_time/counter << endl;
    return 0;
}
int p_output2(double max_process_time){
    double turn_round_time = 0;
    cout << "pid    run_time     begin_time     finish_time\n";
    for(int i = 0; i < counter; i++){
        turn_round_time += tasks[i].end_time;
        cout<<tasks[i].pid<<"     "<<tasks[i].run_time<<"     "
            <<tasks[i].begin_time<<"     "<<tasks[i].end_time<<endl;
    }
    cout << "total_consuming_time= " << turn_round_time << endl;
    cout << "average_running_time= " << turn_round_time/counter << endl;
    cout << "total_running_time= " << max_process_time << endl;
    return 0;
}
```