Abstract

This Project focus on the scheduling problem. The simulation implements FCFS, SJF, RR and Priority methods and output corresponding output include average waiting time, average turnaround time, CPU utilization and Gantt chart. All the code is used C++ 11 standard and VS 2015 to compile.

1

The Shortest Job First scheduling algorithm is optimal in that it minimizes the average waiting time.

Proof

Denote there are n processes whose burst time is t_1, t_2, t_3 … t_n, respectively. We also define that $t_1<t_2<t_3 \ldots <t_n$. Assume all processes arrive at the same time. The SJF algorithm results set is $\{n_1,n_2,n_3 \ldots n_n\}$, and average waiting time is

$$T = (t_1+t_2+\ldots+t_n-1) / n$$

Assume SJF algorithm is not the minimum average waiting time algorithm. Thus, there must exist a set whose specific element n_j burst time larger than n_i where n_j's burst time larger than n_i. And this makes $(T + t_j-t_i)/n < T$. However, $t_j > t_i$, which means the result is against the assumption. In other words, moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, T is the minimum time which means SJF is optimal in it minimizes the average waiting time.

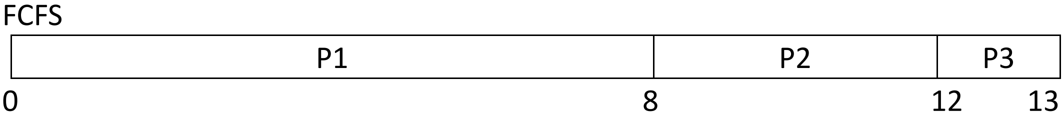
2

2.1 Exercise 5.3

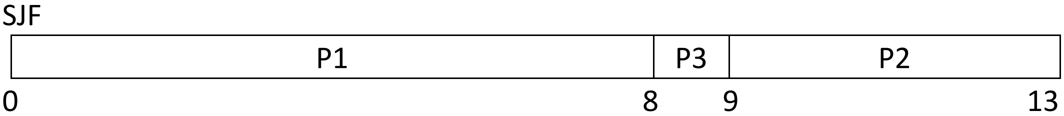
a. The average turnaround time of FCFS algorithm is ((8+4-0.4)+(8+4+1-1)+8)/3 = 10.53
b. The average turnaround time of SJF algorithm is (8+(8+1-1)+(8+4+1-0.4))/3 = 9.53
c. The average turnaround time if CPU left 1 unit time and use SJF is ((1+1-1)+(4+1+1-0.4)+(8+4+1+1))/3 = 6.87

The Gantt charts show as follows:

FCFS



SJF



SJF_Delay

	P3	P2	P1

0 1 2 6 14

Exercise 5.12

The turnaround time and waiting time of each process show as table 2.1 and 2.2.

Table 2.1 Turnaround time

	P1	P2	P3	P4	P5
FCFS	10	11	13	14	19
SJF	19	1	4	2	9
Nonpreemptive Priority	16	1	18	19	6
RR	19	2	7	4	14

Table 2.2 Waiting time

	P1	P2	P3	P4	P5
FCFS	0	10	11	13	14
SJF	9	0	2	1	4
Nonpreemptive Priority	6	0	16	18	1
RR	9	1	5	3	9

a. Average Turnaround time
   FCFS (10+11+13+14+19)/5 = 13.4
   SJF (1+2+4+9+19)/5 = 7
   Priority (1+6+16+18+19)/5 = 12
   RR (2+4+7+14+19)/5 = 9.2
b. Average Waiting Time
   FCFS (10+11+13+14)/5 = 9.6
   SJF (1+2+4+9)/5 = 3.2
   Priority (1+6+16+18)/5 = 8.2
   RR (1+3+5+9+9)/5 = 5.4
c. The minimum average waiting time algorithm is Shortest Job First scheduling whose average waiting time is 3.2.

The four Gantt charts is showed as follows:

FCFS

P1	P2	P3	P4	P5

0 10 11 13 14 19

SJF

P2	P4	P3	P5	P1

0 1 2 4 9 19

Nonpreemptive Priority

P2	P5	P1	P3	P4

0 1 6 16 18 19

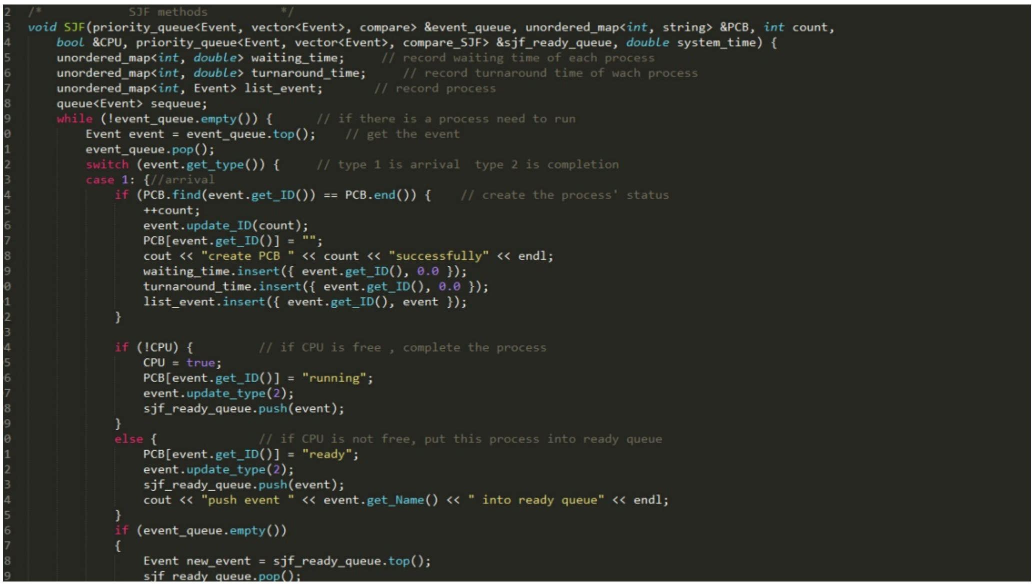
RR

P1	P2	P3	P4	P5	P1	P3	P5	P1	P5	P1	P5	P1	P5	P1

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 19

2.2 Simulation

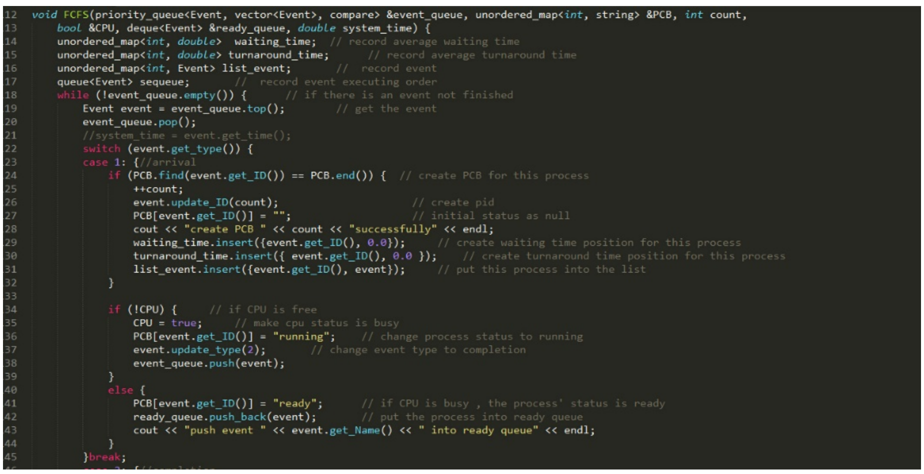
The driven code of FCFS shows as figure 2.1, 2.2.



```
/*              SJF methods            */
void SJF(priority_queue<Event, vector<Event>, compare> &event_queue, unordered_map<int, string> &PCB, int count,
    bool &CPU, priority_queue<Event, vector<Event>, compare_SJF> &sjf_ready_queue, double system_time) {
    unordered_map<int, double> waiting_time;      // record waiting time of each process
    unordered_map<int, double> turnaround_time;      // record turnaround time of wach process
    unordered_map<int, Event> list_event;       // record process
    queue<Event> sequeue;
    while (!event_queue.empty()) {       // if there is a process need to run
        Event event = event_queue.top();     // get the event
        event_queue.pop();
        switch (event.get_type()) {      // type 1 is arrival  type 2 is completion
        case 1: {//arrival
            if (PCB.find(event.get_ID()) == PCB.end()) {    // create the process' status
                ++count;
                event.update_ID(count);
                PCB[event.get_ID()] = "";
                cout << "create PCB " << count << "successfully" << endl;
                waiting_time.insert({ event.get_ID(), 0.0 });
                turnaround_time.insert({ event.get_ID(), 0.0 });
                list_event.insert({ event.get_ID(), event });
            }

            if (!CPU) {          // if CPU is free , complete the process
                CPU = true;
                PCB[event.get_ID()] = "running";
                event.update_type(2);
                sjf_ready_queue.push(event);
            }
            else {               // if CPU is not free, put this process into ready queue
                PCB[event.get_ID()] = "ready";
                event.update_type(2);
                sjf_ready_queue.push(event);
                cout << "push event " << event.get_Name() << " into ready queue" << endl;
            }
            if (event_queue.empty())
            {
                Event new_event = sjf_ready_queue.top();
                sjf_ready_queue.pop();
```

Figure 2.1



```
void FCFS(priority_queue<Event, vector<Event>, compare> &event_queue, unordered_map<int, string> &PCB, int count,
    bool &CPU, deque<Event> &ready_queue, double system_time) {
    unordered_map<int, double>  waiting_time;  // record average waiting time
    unordered_map<int, double> turnaround_time;      // record average turnaround time
    unordered_map<int, Event> list_event;       //  record event
    queue<Event> sequeue;       //  record event executing order
    while (!event_queue.empty()) {      // if there is an event not finished
        Event event = event_queue.top();        // get the event
        event_queue.pop();
        //system_time = event.get_time();
        switch (event.get_type()) {
        case 1: {//arrival
            if (PCB.find(event.get_ID()) == PCB.end()) {  // create PCB for this process
                ++count;
                event.update_ID(count);                       // create pid
                PCB[event.get_ID()] = "";                     // initial status as null
                cout << "create PCB " << count << "successfully" << endl;
                waiting_time.insert({event.get_ID(), 0.0});     // create waiting time position for this process
                turnaround_time.insert({ event.get_ID(), 0.0 });    // create turnaround time position for this process
                list_event.insert({event.get_ID(), event});     // put this process into the list
            }

            if (!CPU) {     // if CPU is free
                CPU = true;     // make cpu status is busy
                PCB[event.get_ID()] = "running";    // change process status to running
                event.update_type(2);       // change event type to completion
                event_queue.push(event);
            }
            else {
                PCB[event.get_ID()] = "ready";      // if CPU is busy , the process' status is ready
                ready_queue.push_back(event);       // put the process into ready queue
                cout << "push event " << event.get_Name() << " into ready queue" << endl;
            }
        }break;
```

Figure 2.2

The SJF driven code shows as figure 2.3, 2.4, 2.5.

```
/*             SJF methods          */
void SJF(priority_queue<Event, vector<Event>, compare> &event_queue, unordered_map<int, string> &PCB, int count,
    bool &CPU, priority_queue<Event, vector<Event>, compare_SJF> &sjf_ready_queue, double system_time) {
    unordered_map<int, double> waiting_time;     // record waiting time of each process
    unordered_map<int, double> turnaround_time;     // record turnaround time of wach process
    unordered_map<int, Event> list_event;       // record process
    queue<Event> sequeue;
    while (!event_queue.empty()) {      // if there is a process need to run
        Event event = event_queue.top();    // get the event
        event_queue.pop();
        switch (event.get_type()) {     // type 1 is arrival  type 2 is completion
        case 1: {//arrival
            if (PCB.find(event.get_ID()) == PCB.end()) {    // create the process' status
                ++count;
                event.update_ID(count);
                PCB[event.get_ID()] = "";
                cout << "create PCB " << count << "successfully" << endl;
                waiting_time.insert({ event.get_ID(), 0.0 });
                turnaround_time.insert({ event.get_ID(), 0.0 });
                list_event.insert({ event.get_ID(), event });
            }

            if (!CPU) {         // if CPU is free , complete the process
                CPU = true;
                PCB[event.get_ID()] = "running";
                event.update_type(2);
                sjf_ready_queue.push(event);
            }
            else {              // if CPU is not free, put this process into ready queue
                PCB[event.get_ID()] = "ready";
                event.update_type(2);
                sjf_ready_queue.push(event);
                cout << "push event " << event.get_Name() << " into ready queue" << endl;
            }
            if (event_queue.empty())
            {
                Event new_event = sjf_ready_queue.top();
                sjf_ready_queue.pop();
```

Figure 2.3

```
            if (event_queue.empty())
            {
                Event new_event = sjf_ready_queue.top();
                sjf_ready_queue.pop();
                event_queue.push(new_event);

                while (!sjf_ready_queue.empty()) {
                    Event temp = sjf_ready_queue.top();
                    if (temp.get_time() >= system_time + new_event.get_running_time())
                        break;
                    temp.update_time(system_time + new_event.get_running_time());
                    sjf_ready_queue.pop();
                    sjf_ready_queue.push(temp);
                }
            }
        }break;
        case 2: {//completion
            system_time += event.get_running_time(); // complete the process execution
            PCB[event.get_ID()].clear();  // destroy the PCB
            sequeue.push(event);
            if ((system_time - event.get_running_time()) == 0) { // record the waiting time and turnaround time
                waiting_time[event.get_ID()] = 0;
                turnaround_time[event.get_ID()] = event.get_running_time() - event.get_arrival_time();
            }
            else {
                waiting_time[event.get_ID()] = system_time - event.get_running_time() - event.get_arrival_time();
                turnaround_time[event.get_ID()] = system_time - event.get_arrival_time();
            }
            cout << "finish " << event.get_Name() << "at time" << system_time << endl;
            if (!sjf_ready_queue.empty()) { // get the top process of the queue and execute
                Event current = sjf_ready_queue.top();
                current.update_time(system_time);
                PCB[current.get_ID()] = "running";
                event_queue.push(current);
                sjf_ready_queue.pop();

                while (!sjf_ready_queue.empty()) {  // update the ready queue's process system time
                    Event temp = sjf_ready_queue.top();
                    if (temp.get_time() >= system_time + current.get_running_time())
                        break;
                    temp.update_time(system_time + current.get_running_time());
                    sjf_ready_queue.pop();
                    sjf_ready_queue.push(temp);
                }
```

Figure 2.4

```
                }
            }
            else
                CPU = false;
        }break;

        }

    }

    cout << system_time << endl;
    string algorithm = "SJF";
    calculate(waiting_time, turnaround_time, algorithm, list_event, sequeue);
}
```

Figure 2.5

The RR's driven code shows as figure 2.6, 2.7.

```cpp
/*  RR methods */
void RR(priority_queue<Event, vector<Event>, compare> &event_queue, unordered_map<int, string> &PCB, int count,
    bool &CPU, deque<Event> &ready_queue, double system_time, double &quantum) {
    unordered_map<int, double> waiting_time;
    unordered_map<int, double> turnaround_time;
    unordered_map<int, Event> list_event;
    queue<pair<Event, double> > sequeue;
    while (!event_queue.empty()) {  // if there is a process need to run
        Event event = event_queue.top();    // get the process
        event_queue.pop();
        switch (event.get_type()) {     // 1 arrival    2 completion   3 timer interrupt
        case 1: {//arrival
            if (PCB.find(event.get_ID()) == PCB.end()) {  // create the process' attributes
                ++count;
                event.update_ID(count);
                PCB[event.get_ID()] = "";
                cout << "create PCB " << count << "successfully" << endl;
                cout << event.get_ID() << " running time is " << event.get_running_time() << endl;
                waiting_time.insert({ event.get_ID(), 0.0 });
                turnaround_time.insert({ event.get_ID(), 0.0 });
                list_event.insert({ event.get_ID(), event });
            }
            if (!CPU) {     // if CPU is free , run this process as for a quantum time
                CPU = true;
                PCB[event.get_ID()] = "running";
                event.update_type(3);
                event_queue.push(event);
            }
            else {          // if CPU is busy, push the process to a ready queue
                PCB[event.get_ID()] = "ready";
                ready_queue.push_back(event);
                cout << "push event " << event.get_ID() << " into ready queue" << endl;
            }
        }break;
        case 2: { // complete
            system_time += event.get_running_time();  // exectue the process for the rest time
            PCB[event.get_ID()].clear();              // destroy its PCB
            if ((system_time - event.get_running_time()) == 0) {    //  record the waiting time and turnaround time
                turnaround_time[event.get_ID()] = event.get_running_time() - event.get_arrival_time();
            }
            else {
                turnaround_time[event.get_ID()] = system_time - event.get_arrival_time();
                waiting_time[event.get_ID()] -= event.get_arrival_time();
            }
            cout << "finish " << event.get_ID() << "at time" << system_time << endl;
```

Figure 2.6

```cpp
            if (ready_queue.empty())
                CPU = false;
            else {  // get another process from the ready queue and execute
                Event current = ready_queue.front();
                PCB[current.get_ID()] = "running";
                current.update_type(3);
                event_queue.push(current);
                ready_queue.pop_front();
            }
        }break;
        case 3: {//timer interrupt
            if (event.get_running_time() <= quantum) // if rest time less than quantum, then complete the process , otherwise call timer agai
                sequeue.push({ event,event.get_running_time() });
            else
                sequeue.push({event, quantum});
            if ((system_time ) == 0) {
                waiting_time[event.get_ID()] += 0;
            }
            else {
                waiting_time[event.get_ID()] += (system_time - event.get_time());
            }
            if (event.get_running_time() <= quantum) {
                event.update_type(2);
                event_queue.push(event);
                continue;
            }
            double remain_time = event.get_running_time() - quantum;    // calculate the remaining time
            system_time += quantum;
            cout << "run " << event.get_ID() << " at time " << system_time << " and remaining  time is " << remain_time << endl;
            event.update_time(system_time);          // update the process system time
            event.update_running_time(remain_time);     // update the process remaining time
            PCB[event.get_ID()] = "ready";              // change the status to ready
            ready_queue.push_back(event);               // put this process to the ready queue
            if (!ready_queue.empty()) {
                Event current = ready_queue.front();
                PCB[current.get_ID()] = "running";
                current.update_type(3);
                event_queue.push(current);
                ready_queue.pop_front();
            }

        }break;
        }
    }
    string algorithm = "RR";
    calculate_RR(waiting_time, turnaround_time, algorithm, list_event, sequeue);
```

Figure 2.7

The Nonpreemptive Priority driven code shows as figure 2.8, 2.9.

```cpp
/*  Nonpreemptive Priority methods */
void Priority_Sche(priority_queue<Event, vector<Event>, compare> &event_queue, unordered_map<int, string> &PCB, int count, bool &CPU,
    priority_queue<Event, vector<Event>, compare_Priority> &priority_ready_queue, double system_time) {
    unordered_map<int, double> waiting_time;
    unordered_map<int, double> turnaround_time;
    unordered_map<int, Event> list_event;
    queue<Event> sequeue;
    while (!event_queue.empty()) {      // if there is a process need to run
        Event event = event_queue.top();        // get the event
        event_queue.pop();
        //system_time = event.get_time();
        switch (event.get_type()) {
        case 1: {//arrival
            if (PCB.find(event.get_ID()) == PCB.end()) {// create the process' attributes
                ++count;
                event.update_ID(count);
                PCB[event.get_ID()] = "";
                cout << "create PCB " << event.get_Name() << "successfully" << endl;
                waiting_time.insert({ event.get_ID(), 0.0 });
                turnaround_time.insert({ event.get_ID(), 0.0 });
                list_event.insert({ event.get_ID(), event });
            }

            if (!CPU) { // if CPU is free run the process
                CPU = true;
                PCB[event.get_ID()] = "running";
                event.update_type(2);
                priority_ready_queue.push(event);
            }
            else {  // otherwise push this process to the ready queue
                PCB[event.get_ID()] = "ready";
                event.update_type(2);
                priority_ready_queue.push(event);
                cout << "push event " << event.get_Name() << " into ready queue" << endl;
            }

            if (event_queue.empty())
            {
                Event new_event = priority_ready_queue.top();
                priority_ready_queue.pop();
                event_queue.push(new_event);
                while (!priority_ready_queue.empty()) {
                    Event temp = priority_ready_queue.top();
                    if (temp.get_time() >= system_time + new_event.get_running_time())
                        break;
```

Figure 2.8

```cpp
                while (!priority_ready_queue.empty()) {
                    Event temp = priority_ready_queue.top();
                    if (temp.get_time() >= system_time + new_event.get_running_time())
                        break;
                    temp.update_time(system_time + new_event.get_running_time());
                    priority_ready_queue.pop();
                    priority_ready_queue.push(temp);
                }
            }
        }break;
        case 2: {//completion
            system_time += event.get_running_time();    // execute the process
            PCB[event.get_ID()].clear();                // destroy the PCB
            sequeue.push(event);
            if ((system_time - event.get_running_time()) == 0) {    // record the waiting time and turnaround time
                waiting_time[event.get_ID()] = 0;
                turnaround_time[event.get_ID()] = event.get_running_time() - event.get_arrival_time();
            }
            else {
                waiting_time[event.get_ID()] = system_time - event.get_running_time() - event.get_arrival_time();
                turnaround_time[event.get_ID()] = system_time - event.get_arrival_time();
            }
            cout << "finish " << event.get_Name() << "at time" << system_time << endl;
            if (!priority_ready_queue.empty()) {    // continue execute the event in the ready queue
                Event current = priority_ready_queue.top();
                current.update_time(system_time);
                PCB[current.get_ID()] = "running";
                event_queue.push(current);
                priority_ready_queue.pop();

                while (!priority_ready_queue.empty()) {
                    Event temp = priority_ready_queue.top();
                    if (temp.get_time() >= system_time + current.get_running_time())
                        break;
                    temp.update_time(system_time + current.get_running_time());
                    priority_ready_queue.pop();
                    priority_ready_queue.push(temp);
                }
            }
            else {
                CPU = false;
            }
        }break;

        }
```

Figure 2.9

The 5.3 and 5.12 output shows as figure 2.10 ~ 2.13.

```
create PCB 1successfully
create PCB 2successfully
push event P2 into ready queue
create PCB 3successfully
push event P3 into ready queue
finish P1at time8
finish P2at time12
finish P3at time13
13
The Average waiting time of FCFS is 6.2
The Average turnaround time of FCFS is 10.5333
 | P1________8 | P2____12 | P3_13 |

create PCB 1successfully
create PCB 2successfully
push event P2 into ready queue
create PCB 3successfully
push event P3 into ready queue
finish P1at time8
finish P3at time9
finish P2at time13
13
The Average waiting time of SJF is 5.2
The Average turnaround time of SJF is 9.53333
 | P1________8 | P3_9 | P2____13 |

create PCB 1successfully
create PCB 2successfully
push event P2 into ready queue
create PCB 3successfully
push event P3 into ready queue
finish P3at time2
finish P2at time6
finish P1at time14
14
The Average waiting time of delay_SJF is 2.86667
The Average turnaround time of delay_SJF is 6.86667
| delay_1 | P3_2 | P2____6 | P1________14 |
```

Figure 2.10

```
create PCB 3successfully
push event P3 into ready queue
create PCB 4successfully
push event P4 into ready queue
create PCB 5successfully
push event P5 into ready queue
finish P1at time10
finish P2at time11
finish P3at time13
finish P4at time14
finish P5at time19
19
The Average waiting time of FCFS is 9.6
The Average turnaround time of FCFS is 13.4
 | P1__________10 | P2_11 | P3__13 | P4_14 | P5_____19 |

create PCB 1successfully
create PCB 2successfully
push event P2 into ready queue
create PCB 3successfully
push event P3 into ready queue
create PCB 4successfully
push event P4 into ready queue
create PCB 5successfully
push event P5 into ready queue
finish P2at time1
finish P4at time2
finish P3at time4
finish P5at time9
finish P1at time19
19
The Average waiting time of SJF is 3.2
The Average turnaround time of SJF is 7
 | P2_1 | P4_2 | P3__4 | P5_____9 | P1__________19 |
```

Figure 2.11

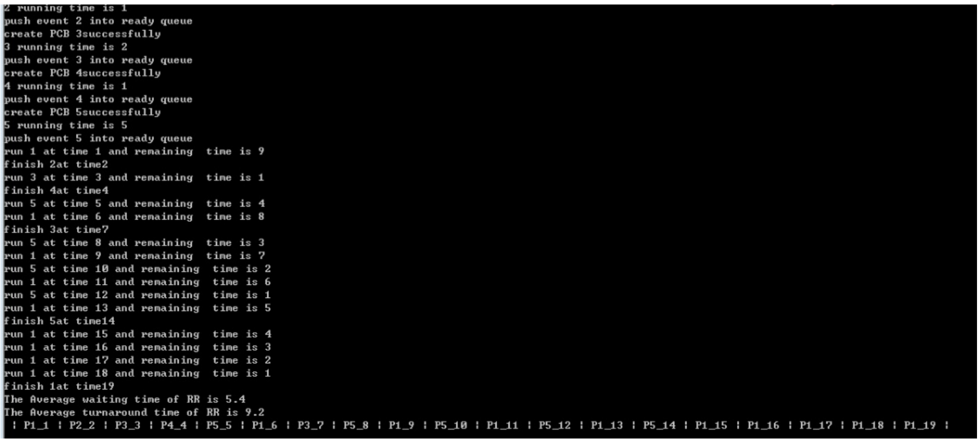


Figure 2.12

Figure 2.13

3

The simulation uses random exponential generator to generate the I/O interrupt randomly.

a) Observation of FCFS and SJF

Because CPU does not know the processes' burst time. Thus, CPU will predict the CPU burst time to run SJF methods. The equation is as follows:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n \qquad (0 \le \alpha \le 1)$$

The results show as table 3.1.

Table 3.1 average waiting time

	Average waiting time(min)	Average turnaround time(min)	Throughput (hour)	CPU utilization	Total simulation time(min)
FCFS	24.3727	27.1323	17.8685	0.821821	33.578667
SJF(α=1)	24.3165	27.076	17.9139	0.823907	33.493667
SJF(α=0.5)	24.3354	27.0949	17.8107	0.819162	33.687667
SJF(α=1/3)	24.2546	27.0142	17.8451	0.820746	33.622667

From table 3.1, when α is 0.5, the system has the lowest average waiting time, however, the simulation time is the longest. And all SJF simulations' average time is smaller than FCFS but the FCFS simulation has a good CPU utilization compared with SJF α is 0.5 and α is 1/3. The average waiting time is good, but the CPU utilization is a little different because the random generator makes the IO switching times more or less to results the FCFS has a good CPU utilization.

b) Statement:
   1) When quantum is small enough, the CPU utilization time will increase when quantum increase. And the average waiting time will decrease when quantum increase.
   2) With increasing the quantum close to the IO requests time, the CPU utilization and average waiting time will vibrate follow the time.

c) Based on the statement at b), the system results show as figure 3.1 ~ 3.6.

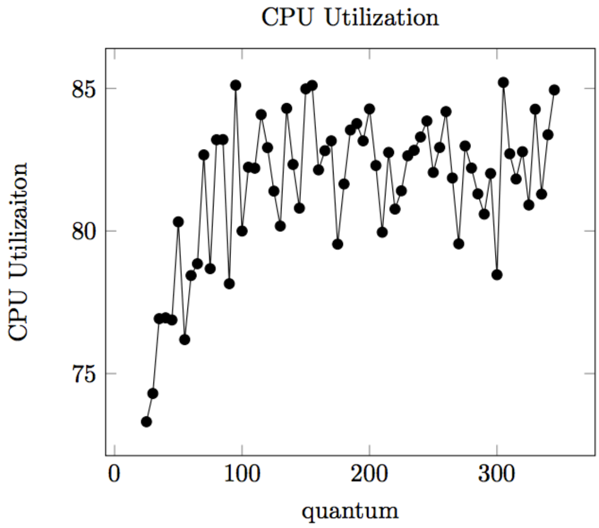


Figure 3.1

From figure 3.1, we know that the quantum from 0 to 50, the CPU utilization is increasing. After 50, the CPU utilization has a vibrate data. It's because when quantum is small enough (i.e. 25), the system timer interruption is almost determined by quantum. And the quantum is too small make the whole system switching too many times which results low CPU utilization. With the quantum increasing to the value which the timer interruption determined not only by quantum, but also IO requests, the CPU utilization determined by the interrupt time which is better to perform the CPU utilization.

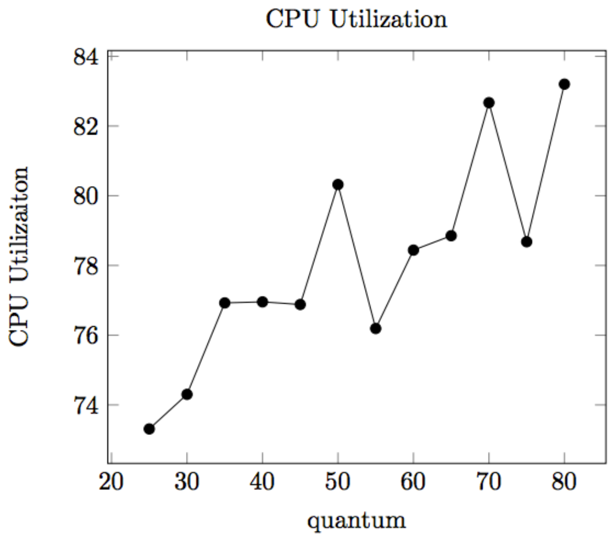


Figure 3.2

From the figure 3.2, the conclusion is that the CPU utilization's trend is increasing with the quantum from 20 to 80 because the the system switching times reduces. Every process has more time to run in every single timer.

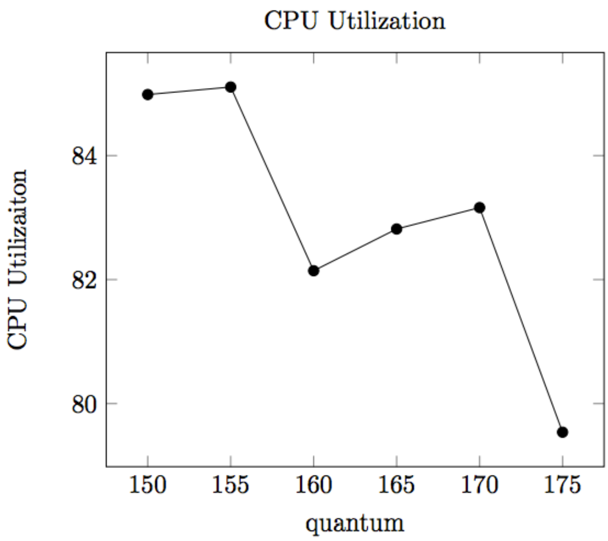


Figure 3.3

From figure 3.3, the CPU utilization decreases with quantum from 155 to 175. Because of the timer interruption decreases, the system efficiency is growing down. Consequently, the timer interruption length is important to the system efficiency. If it is too short, the system will have too many switching times, which results CPU utilization low. If it is too long, the system will have bad performance like FCFS. The system CPU utilization is vibrating because it the IO requests is random generated and this is not stable.

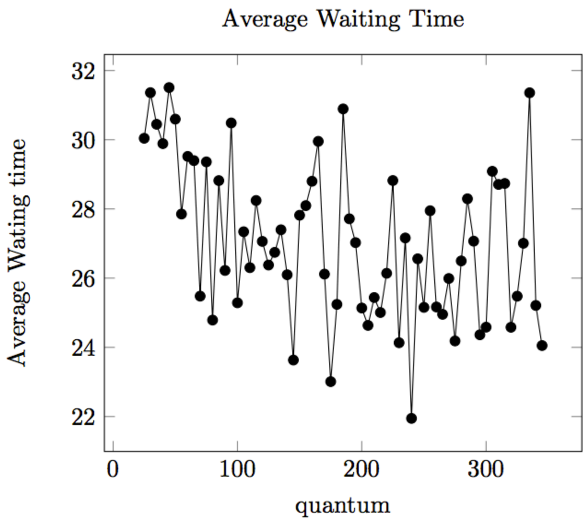


Figure 3.4

From figure 3.4, the average waiting time tendency is decrease with quantum from 0 to 80. Because during this quantum values, the system switching times decreases and increases the CPU utilization which makes the system is more efficiency.

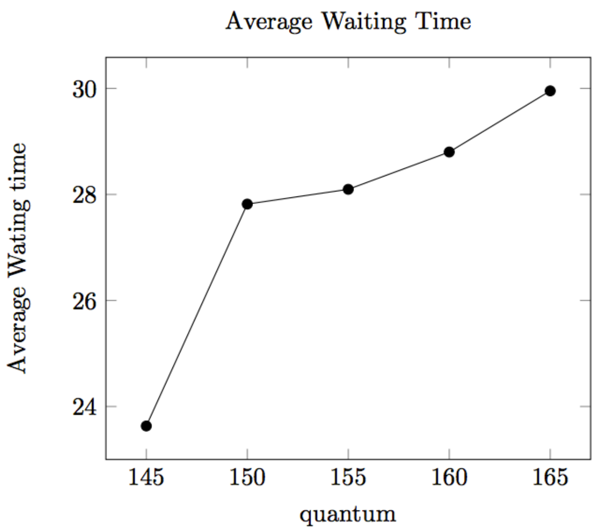


Figure 3.5

From figure 3.5, the average waiting time increase because the quantum becomes large and make the system degenerate to FCFS. Thus, the system average waiting time is larger and larger.

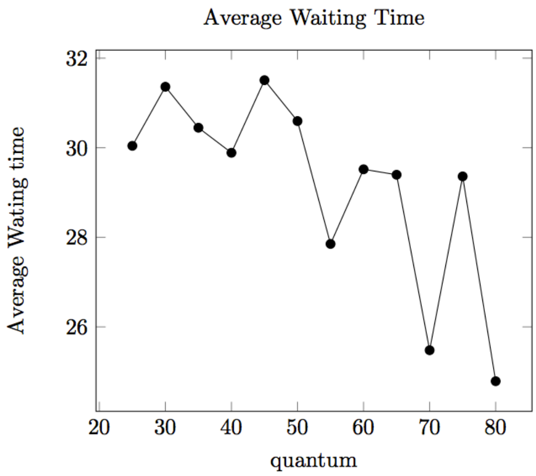


Figure 3.6

From figure 3.6, the average waiting time has a decrease trend because the with the quantum increasing the context switching times decreases in a reasonable scale. And this will make system reduce large amount of switching time and waiting time.