

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

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

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.

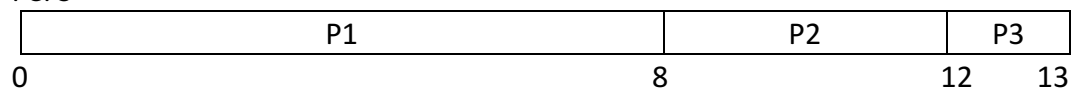
2

2.1 Exercise 5.3

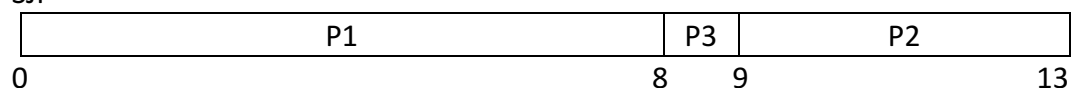
- The average turnaround time of FCFS algorithm is $((8+4-0.4)+(8+4+1-1)+8)/3 = 10.53$
- The average turnaround time of SJF algorithm is $(8+(8+1-1)+(8+4+1-0.4))/3 = 9.53$
- The average turnaround time if CPU left 1 unit time and use SJF is $((1+1-1)+(4+1+1-0.4)+(8+4+1+1))/3 = 6.87$

The Gantt charts show as follows:

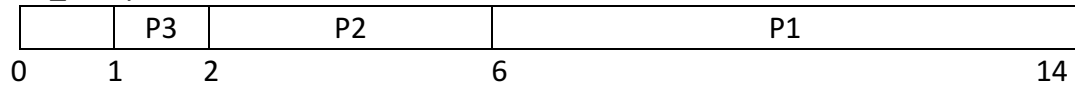
FCFS



SJF



SJF_Delay



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 $(1+2+4+9)/5 = 3.2$

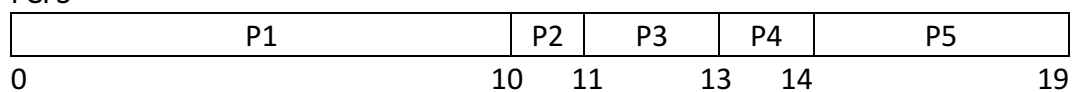
Priority $(1+6+16+18)/5 = 8.2$

RR $(1+3+5+9+9)/5 = 5.4$

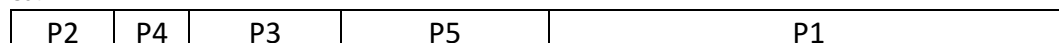
c. The minimum average waiting time algorithm is Shortest Job First scheduling whose average waiting time is 3.2.

The four Gantt charts is showed as follows:

FCFS



SJF



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.

```

1  /* SJF methods */
2
3  void SJF(priority_queue<Event, vector<Event>, compare> &event_queue, unordered_map<int, string> &PCB, int count,
4  bool &CPU, priority_queue<Event, vector<Event>, compare> SJF> &sjf_ready_queue, double system_time) {
5      unordered_map<int, double> waiting_time; // record waiting time of each process
6      unordered_map<int, double> turnaround_time; // record turnaround time of each process
7      unordered_map<int, Event> list_event; // record process
8      queue<Event> sequeque;
9      while (!event_queue.empty()) { // if there is a process need to run
10         Event event = event_queue.top(); // get the event
11         event_queue.pop();
12         switch (event.get_type()) { // type 1 is arrival type 2 is completion
13             case 1: { // arrival
14                 if (PCB.find(event.get_ID()) == PCB.end()) { // create the process' status
15                     ++count;
16                     event.update_ID(count);
17                     PCB[event.get_ID()] = "";
18                     cout << "create PCB " << count << "successfully" << endl;
19                     waiting_time.insert({ event.get_ID(), 0.0 });
20                     turnaround_time.insert({ event.get_ID(), 0.0 });
21                     list_event.insert({ event.get_ID(), event });
22                 }
23             }
24             if (!CPU) { // if CPU is free , complete the process
25                 CPU = true;
26                 PCB[event.get_ID()] = "running";
27                 event.update_type(2);
28                 sjf_ready_queue.push(event);
29             }
30             else { // if CPU is not free, put this process into ready queue
31                 PCB[event.get_ID()] = "ready";
32                 event.update_type(2);
33                 sjf_ready_queue.push(event);
34                 cout << "push event " << event.get_Name() << " into ready queue" << endl;
35             }
36         }
37         if (event_queue.empty())
38         {
39             Event new_event = sjf_ready_queue.top();
40             sjf_ready_queue.pop();

```

Figure 2.1

```

1 void PCBs(priority_queue<Event> vector<Event> &compare, unordered_map<int, string> &PCB, int count,
2 bool &CPU, deque<Event> &ready_queue, double system_time) {
3     unordered_map<int, double> waiting_time; // record average waiting time
4     unordered_map<int, double> turnaround_time; // record average turnaround time
5     unordered_map<int, Event> list_event; // record event
6     queue<Event> sequeuse; // record event executing order
7     while (!event_queue.empty()) { // if there is an event not finished
8         Event event = event_queue.top(); // get the event
9         event_queue.pop();
10        //system_time = event.get_time();
11        switch (event.get_type()) {
12        case 1: { //arrival
13            if (PCB.find(event.get_ID()) == PCB.end()) { // create PCB for this process
14                ++count;
15                event.update_ID(count); // create pid
16                PCB[event.get_ID()] = ""; // initial status as null
17                cout << "create PCB = " << count << "successfully" << endl;
18                waiting_time.insert({event.get_ID(), 0.0}); // create waiting time position for this process
19                turnaround_time.insert({event.get_ID(), 0.0}); // create turnaround time position for this process
20                list_event.insert({event.get_ID(), event}); // put this process into the list
21            }
22        }
23        case 1: { // CPU is free
24            CPU = true; // make cpu status is busy
25            PCB[event.get_ID()] = "running"; // change process status to running
26            event.update_type(2); // change event type to completion
27            event_queue.push(event);
28        }
29        else {
30            PCB[event.get_ID()] = "ready"; // if CPU is busy , the process' status is ready
31            ready_queue.push_back(event); // put the process into ready queue
32            cout << "push event " << event.get_Name() << " into ready queue" << endl;
33        }
34    }
35    }break;
36 }

```

Figure 2.2

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

```

2  /* SJF methods */
3  void SJF(priority_queue<Event, vector<Event>, compare> &event_queue, unordered_map<int, string> &PCB, int count,
4  bool &CPU, priority_queue<Event, vector<Event>, compare_SJF> &sjf_ready_queue, double system_time) {
5  unordered_map<int, double> waiting_time; // record waiting time of each process
6  unordered_map<int, double> turnaround_time; // record turnaround time of each process
7  unordered_map<int, Event> list_event; // record process
8  queue<Event> sequeue;
9  while (!event_queue.empty()) { // if there is a process need to run
10     Event event = event_queue.top(); // get the event
11     event_queue.pop();
12     switch (event.get_type()) { // type 1 is arrival type 2 is completion
13     case 1: { // arrival
14         if (PCB.find(event.get_ID()) == PCB.end()) { // create the process' status
15             ++count;
16             event.update_ID(count);
17             PCB[event.get_ID()] = "";
18             cout << "create PCB " << count << "successfully" << endl;
19             waiting_time.insert({ event.get_ID(), 0.0 });
20             turnaround_time.insert({ event.get_ID(), 0.0 });
21             list_event.insert({ event.get_ID(), event });
22         }
23
24         if (!CPU) { // if CPU is free , complete the process
25             CPU = true;
26             PCB[event.get_ID()] = "running";
27             event.update_type(2);
28             sjf_ready_queue.push(event);
29         }
30         else { // if CPU is not free, put this process into ready queue
31             PCB[event.get_ID()] = "ready";
32             event.update_type(2);
33             sjf_ready_queue.push(event);
34             cout << "push event " << event.get_Name() << " into ready queue" << endl;
35         }
36     }
37     if (event_queue.empty())
38     {
39         Event new_event = sjf_ready_queue.top();
40         sjf_ready_queue.pop();

```

Figure 2.3

```

16     if (event_queue.empty())
17     {
18         Event new_event = sjf_ready_queue.top();
19         sjf_ready_queue.pop();
20         event_queue.push(new_event);
21
22         while (!sjf_ready_queue.empty()) {
23             Event temp = sjf_ready_queue.top();
24             if (temp.get_time() >= system_time + new_event.get_running_time())
25                 break;
26             temp.update_time(system_time + new_event.get_running_time());
27             sjf_ready_queue.pop();
28             sjf_ready_queue.push(temp);
29         }
30     }
31 }break;
32 case 2: { //completion
33     system_time += event.get_running_time(); // complete the process execution
34     PCB[event.get_ID()].clear(); // destroy the PCB
35     sequeue.push(event);
36     if ((system_time - event.get_running_time()) == 0) { // record the waiting time and turnaround time
37         waiting_time[event.get_ID()] = 0;
38         turnaround_time[event.get_ID()] = event.get_running_time() - event.get_arrival_time();
39     }
40     else {
41         waiting_time[event.get_ID()] = system_time - event.get_running_time() - event.get_arrival_time();
42         turnaround_time[event.get_ID()] = system_time - event.get_arrival_time();
43     }
44     cout << "finish " << event.get_Name() << "at time" << system_time << endl;
45     if (!sjf_ready_queue.empty()) { // get the top process of the queue and execute
46         Event current = sjf_ready_queue.top();
47         current.update_time(system_time);
48         PCB[current.get_ID()] = "running";
49         event_queue.push(current);
50         sjf_ready_queue.pop();
51
52         while (!sjf_ready_queue.empty()) { // update the ready queue's process system time
53             Event temp = sjf_ready_queue.top();
54             if (temp.get_time() >= system_time + current.get_running_time())
55                 break;
56             temp.update_time(system_time + current.get_running_time());
57             sjf_ready_queue.pop();
58             sjf_ready_queue.push(temp);
59         }

```

Figure 2.4

```

59     }
60     }
61     else
62         CPU = false;
63 }break;
64 }
65 }
66 }
67 }
68 }
69 cout << system_time << endl;
70 string algorithm = "SJF";
71 calculate(waiting_time, turnaround_time, algorithm, list_event, sequeue);
72 }

```

Figure 2.5

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

```

74  /* RR method */
75  void RR(priority_queue<Event, vector<Event>, compare> &event_queue, unordered_map<int, string> &PCB, int count,
76  bool &CPU, deque<Event> &ready_queue, double system_time, double &quantum) {
77  unordered_map<int, double> waiting_time;
78  unordered_map<int, double> turnaround_time;
79  unordered_map<int, Event> list_event;
80  queue<pair<Event, double>> sequeue;
81  while (!event_queue.empty()) { // if there is a process need to run
82  Event event = event_queue.top(); // get the process
83  event_queue.pop();
84  switch (event.get_type()) { // 1 arrival 2 completion 3 timer interrupt
85  case 1: { // arrival
86  if (PCB.find(event.get_ID()) == PCB.end()) { // create the process' attributes
87  ++count;
88  event.update_ID(count);
89  PCB[event.get_ID()] = "";
90  cout << "create PCB " << count << "successfully" << endl;
91  cout << event.get_ID() << " running time is " << event.get_running_time() << endl;
92  waiting_time.insert({ event.get_ID(), 0.0 });
93  turnaround_time.insert({ event.get_ID(), 0.0 });
94  list_event.insert({ event.get_ID(), event });
95  }
96  if (!CPU) { // if CPU is free , run this process as for a quantum time
97  CPU = true;
98  PCB[event.get_ID()] = "running";
99  event.update_type(3);
100  event_queue.push(event);
101  }
102  else { // if CPU is busy, push the process to a ready queue
103  PCB[event.get_ID()] = "ready";
104  ready_queue.push_back(event);
105  cout << "push event " << event.get_ID() << " into ready queue" << endl;
106  }
107  }break;
108  case 2: { // complete
109  system_time += event.get_running_time(); // execute the process for the rest time
110  PCB[event.get_ID()].clear(); // destroy its PCB
111  if ((system_time - event.get_running_time()) == 0) { // record the waiting time and turnaround time
112  turnaround_time[event.get_ID()] = event.get_running_time() - event.get_arrival_time();
113  }
114  else {
115  turnaround_time[event.get_ID()] = system_time - event.get_arrival_time();
116  waiting_time[event.get_ID()] += event.get_running_time();
117  }
118  cout << "finish " << event.get_ID() << "at time" << system_time << endl;

```

Figure 2.6

```

19  if (ready_queue.empty())
20  CPU = false;
21  else { // get another process from the ready queue and execute
22  Event current = ready_queue.front();
23  PCB[current.get_ID()] = "running";
24  current.update_type(3);
25  event_queue.push(current);
26  ready_queue.pop_front();
27  }
28  }break;
29  case 3: { // timer interrupt
30  if (event.get_running_time() <= quantum) // if rest time less than quantum, then complete the process , otherwise call timer again
31  sequeue.push({ event, event.get_running_time() });
32  else
33  sequeue.push({ event, quantum });
34  if ((system_time) == 0) {
35  waiting_time[event.get_ID()] += 0;
36  }
37  else {
38  waiting_time[event.get_ID()] += (system_time - event.get_time());
39  }
40  if (event.get_running_time() <= quantum) {
41  event.update_type(2);
42  event_queue.push(event);
43  continue;
44  }
45  double remain_time = event.get_running_time() - quantum; // calculate the remaining time
46  system_time += quantum;
47  cout << "run " << event.get_ID() << " at time " << system_time << " and remaining time is " << remain_time << endl;
48  event.update_time(system_time); // update the process system time
49  event.update_running_time(remain_time); // update the process remaining time
50  PCB[event.get_ID()] = "ready"; // change the status to ready
51  ready_queue.push_back(event); // put this process to the ready queue
52  if (!ready_queue.empty()) {
53  Event current = ready_queue.front();
54  PCB[current.get_ID()] = "running";
55  current.update_type(3);
56  event_queue.push(current);
57  ready_queue.pop_front();
58  }
59  }
60  }break;
61  }
62  }
63  string algorithm = "RR";
64  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.

```

68  /* Nonpreemptive Priority methods */
69  void Priority_Sche(priority_queue<Event>, vector<Event>, compare) &event_queue, unordered_map<int, string> &PCB, int count, bool &CPU,
70  priority_queue<Event>, vector<Event>, compare_Priority> &priority_ready_queue, double system_time) {
71      unordered_map<int, double> waiting_time;
72      unordered_map<int, double> turnaround_time;
73      unordered_map<int, Event> list_event;
74      queue<Event> sequeue;
75      while (!event_queue.empty()) { // if there is a process need to run
76          Event event = event_queue.top(); // get the event
77          event_queue.pop();
78          //system_time = event.get_time();
79          switch (event.get_type()) {
80              case 1: //arrival
81                  if (PCB.find(event.get_ID()) == PCB.end()) { // create the process' attributes
82                      ++count;
83                      event.update_ID(count);
84                      PCB[event.get_ID()] = "";
85                      cout << "create PCB " << event.get_Name() << "successfully" << endl;
86                      waiting_time.insert({ event.get_ID(), 0.0 });
87                      turnaround_time.insert({ event.get_ID(), 0.0 });
88                      list_event.insert({ event.get_ID(), event });
89                  }
90
91                  if (!CPU) { // if CPU is free run the process
92                      CPU = true;
93                      PCB[event.get_ID()] = "running";
94                      event.update_type(2);
95                      priority_ready_queue.push(event);
96                  }
97                  else { // otherwise push this process to the ready queue
98                      PCB[event.get_ID()] = "ready";
99                      event.update_type(2);
100                      priority_ready_queue.push(event);
101                      cout << "push event " << event.get_Name() << " into ready queue" << endl;
102                  }
103
104                  if (event_queue.empty())
105                  {
106                      Event new_event = priority_ready_queue.top();
107                      priority_ready_queue.pop();
108                      event_queue.push(new_event);
109                      while (!priority_ready_queue.empty()) {
110                          Event temp = priority_ready_queue.top();
111                          if (temp.get_time() >= system_time + new_event.get_running_time())
112                              break;

```

Figure 2.8

```

99          while (!priority_ready_queue.empty()) {
100              Event temp = priority_ready_queue.top();
101              if (temp.get_time() >= system_time + new_event.get_running_time())
102                  break;
103              temp.update_time(system_time + new_event.get_running_time());
104              priority_ready_queue.pop();
105              priority_ready_queue.push(temp);
106          }
107      }break;
108      case 2: //completion
109          system_time += event.get_running_time(); // execute the process
110          PCB[event.get_ID()].clear(); // destroy the PCB
111          sequeue.push(event);
112          if ((system_time - event.get_running_time()) == 0) { // record the waiting time and turnaround time
113              waiting_time[event.get_ID()] = 0;
114              turnaround_time[event.get_ID()] = event.get_running_time() - event.get_arrival_time();
115          }
116          else {
117              waiting_time[event.get_ID()] = system_time - event.get_running_time() - event.get_arrival_time();
118              turnaround_time[event.get_ID()] = system_time - event.get_arrival_time();
119          }
120          cout << "finish " << event.get_Name() << "at time" << system_time << endl;
121          if (!priority_ready_queue.empty()) { // continue execute the event in the ready queue
122              Event current = priority_ready_queue.top();
123              current.update_time(system_time);
124              PCB[current.get_ID()] = "running";
125              event_queue.push(current);
126              priority_ready_queue.pop();
127          }
128          while (!priority_ready_queue.empty()) {
129              Event temp = priority_ready_queue.top();
130              if (temp.get_time() >= system_time + current.get_running_time())
131                  break;
132              temp.update_time(system_time + current.get_running_time());
133              priority_ready_queue.pop();
134              priority_ready_queue.push(temp);
135          }
136          else {
137              CPU = false;
138          }
139      }break;
140  }

```

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
i3
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
i3
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
i4
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
i9
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
i9
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


```

2 running time is 1
push event 2 into ready queue
create PCB 3 successfully
3 running time is 2
push event 3 into ready queue
create PCB 4 successfully
4 running time is 1
push event 4 into ready queue
create PCB 5 successfully
5 running time is 5
push event 5 into ready queue
run 1 at time 1 and remaining time is 9
finish 2 at time 2
run 3 at time 3 and remaining time is 1
finish 4 at time 4
run 5 at time 5 and remaining time is 4
run 1 at time 6 and remaining time is 8
finish 3 at time 7
run 5 at time 8 and remaining time is 3
run 1 at time 9 and remaining time is 7
run 5 at time 10 and remaining time is 2
run 1 at time 11 and remaining time is 6
run 5 at time 12 and remaining time is 1
run 1 at time 13 and remaining time is 5
finish 5 at time 14
run 1 at time 15 and remaining time is 4
run 1 at time 16 and remaining time is 3
run 1 at time 17 and remaining time is 2
run 1 at time 18 and remaining time is 1
finish 1 at time 19
The Average waiting time of RR is 5.4
The Average turnaround time of RR is 9.2
: P1_1 : P2_2 : P3_3 : P4_4 : P5_5 : P1_6 : P3_7 : P5_8 : P1_9 : P5_10 : P1_11 : P5_12 : P1_13 : P5_14 : P1_15 : P1_16 : P1_17 : P1_18 : P1_19 :

```

Figure 2.12

```

create PCB P1 successfully
create PCB P2 successfully
push event P2 into ready queue
create PCB P3 successfully
push event P3 into ready queue
create PCB P4 successfully
push event P4 into ready queue
create PCB P5 successfully
push event P5 into ready queue
finish P2 at time 1
finish P5 at time 6
finish P1 at time 16
finish P3 at time 18
finish P4 at time 19
19
The Average waiting time of Priority is 8.2
The Average turnaround time of Priority is 12
: P2_1 : P5_6 : P1_16 : P3_18 : P4_19 :

```

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:

$$\tau_{n+1} = \alpha \tau_n + (1 - \alpha) \tau_n \quad (0 \leq \alpha \leq 1)$$

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($\alpha=1$)	24.3165	27.076	17.9139	0.823907	33.493667
SJF($\alpha=0.5$)	24.3354	27.0949	17.8107	0.819162	33.687667
SJF($\alpha=1/3$)	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.

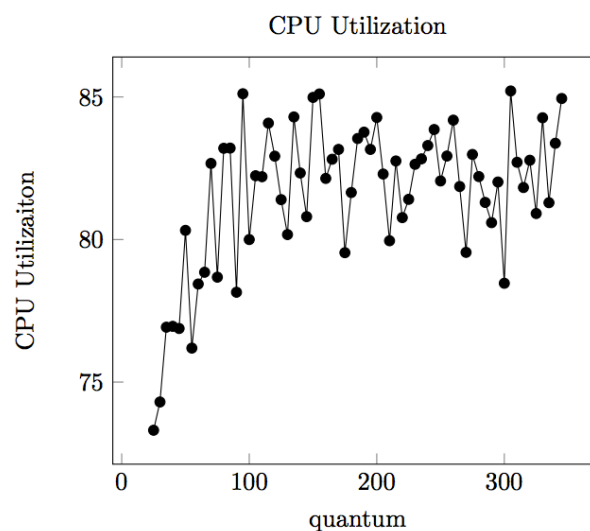


Figure 3.1

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.

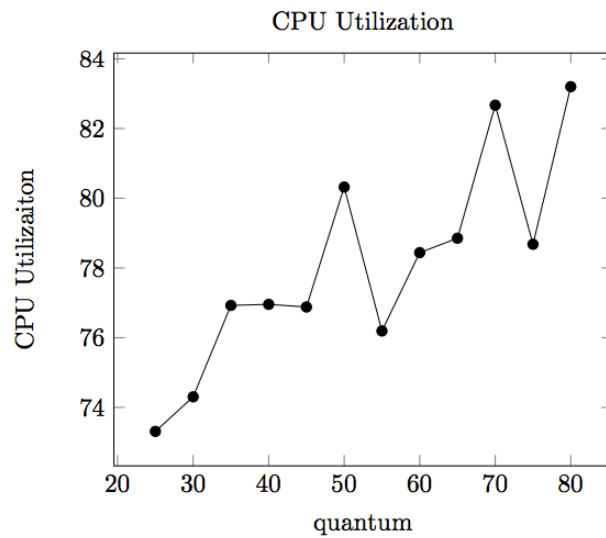


Figure 3.2

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.

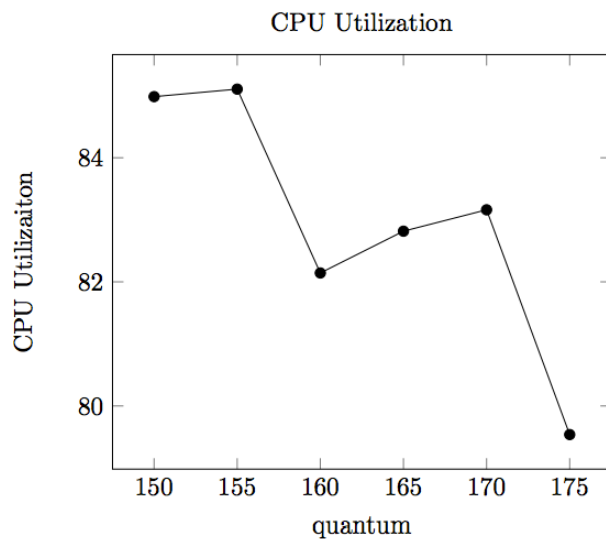


Figure 3.3

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.

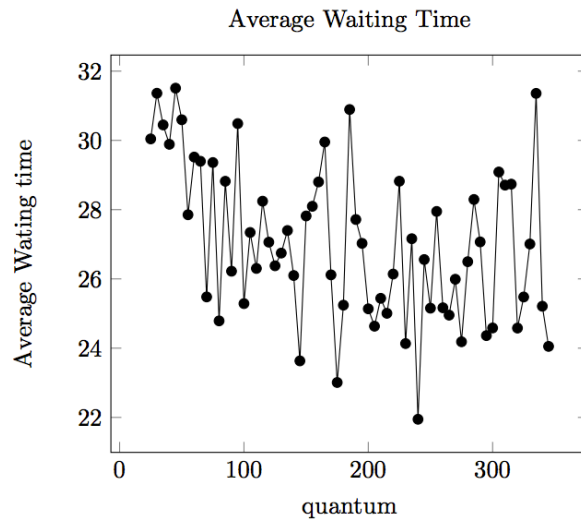


Figure 3.4

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.

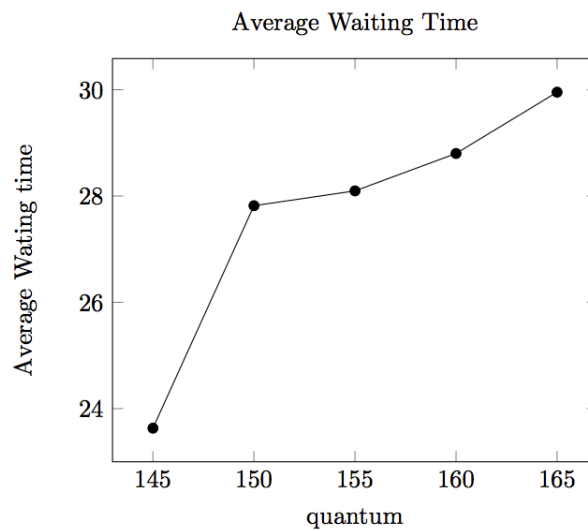


Figure 3.5

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.

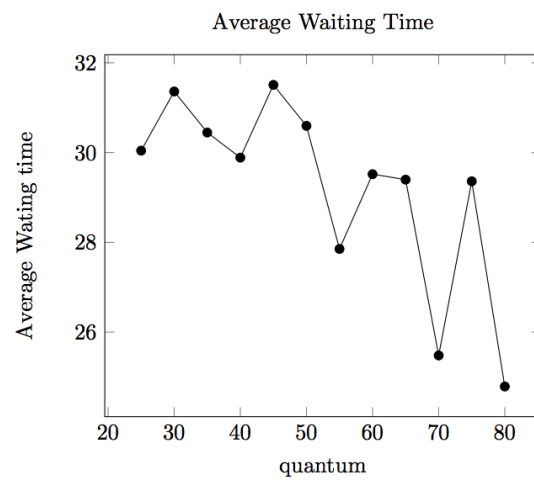


Figure 3.6

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.