# Round Robin report

软英1602 陈格平 20165187

Purpose

The goal of this assignment is to explore the differences in normalized turn-around time performance when using the round robin scheduling algorithm with various time quantum. The primary task is to implement a Round Robin scheduler and experiment to find what time quantum works best with the given input sequence of task arrival times and duration.

Methodology

## The process of round robin

* Firstly, open the file to load the data in.
* set the counter as 0, in order that we can use it to judge whether the running process should be changed.
* In a while statement, new deques for new process, ready to be loaded and the exited process.
* Increase the simulation Time every time the program finish the loop, so as the counter.
* Every time the process arrived, put the process into the ready queue, and pop it out in the new process queue.
* While there is no process running, but we have process ready at the ready queue, run the first ready process in the queue.
* While running, if the counter equals to the slice, that means we should change next process in the ready queue, put the running process at the back of the ready queue, and terminated the process. Reset the counter as 0.
* While running, if the job has completed, then push the job into the exit queue to calculate the normalized time at the end, reset the counter as 0 again.

## Pseudocode

1. Increase simulation time

|  |
| --- |
| simTime++; |

1. Insert the process into the ready queue when new process arrived

|  |
| --- |
| while (simTime == newProcs.front().getArrival()) {  readyProcs.push\_back(newProcs.front());  newProcs.pop\_front();  } |

1. If the counter is equal to the time quantum, set isRunning=false, reset the counter, push the running process into the ready process. And increase the counter .

|  |
| --- |
| if(isRunning && counter==q){  readyProcs.push\_back(runningProc);  isRunning=false;  counter=0;  }  counter++; |

1. If no process is running and we still have process ready at the ready queue, set the running process and pop out the terminated process

|  |
| --- |
| if (!isRunning && !readyProcs.empty()) {  runningProc=readyProcs.front(); readyProcs.pop\_front();  isRunning = true;  } |

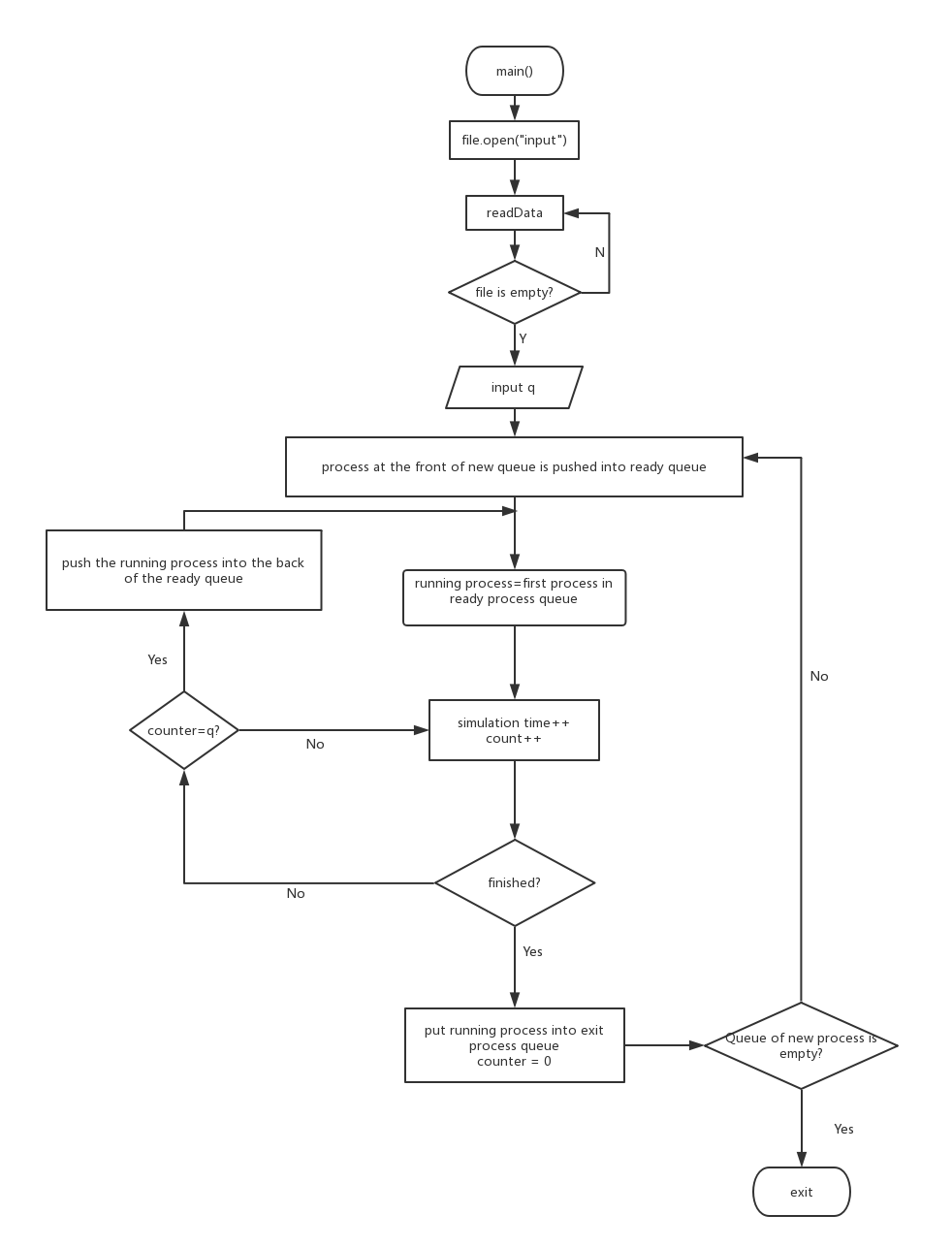
1. For the running process, if the job has completed, push the running process into the exit process, reset the counter and set the isRunning to false.

|  |
| --- |
| if (isRunning) {  if(runningProc.process()){  runningProc.setFinished(simTime+1); exitProcs.push\_back(runningProc);  isRunning = false; //prepare for the next process  counter=0;  }  } |

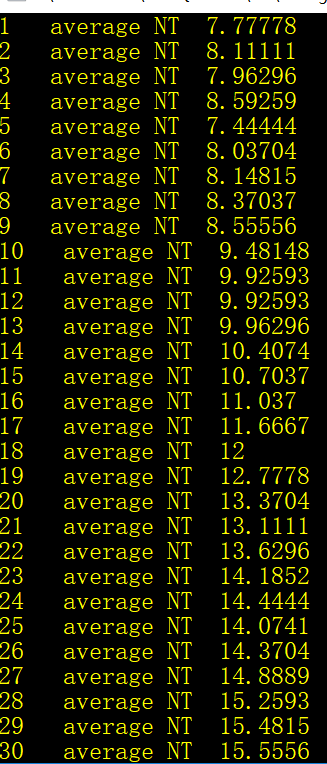
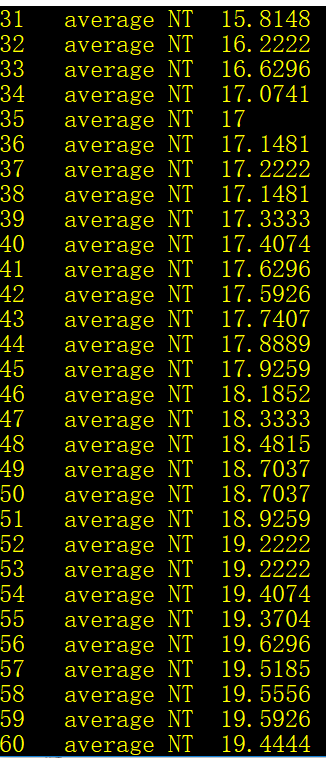
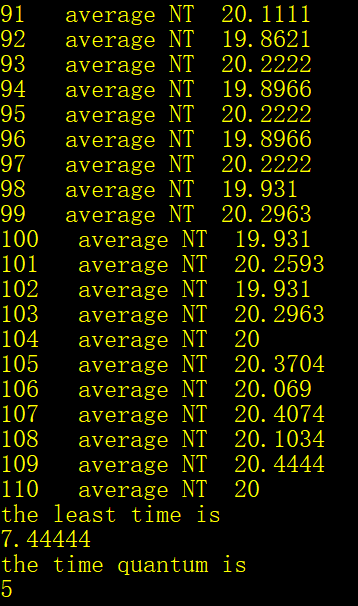
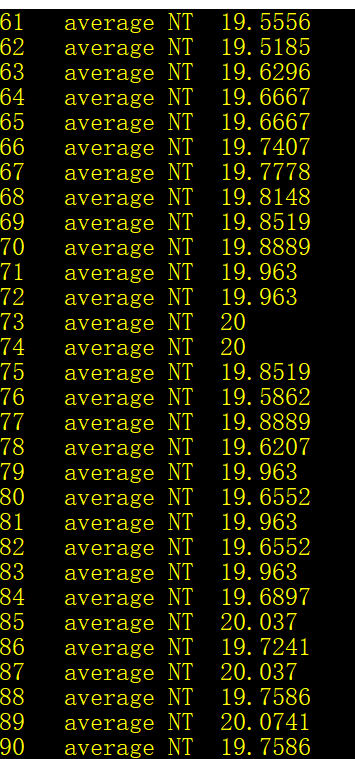
1. Calculate the normalized time, and find the best time quantum.

|  |
| --- |
| cout<<q  unsigned int avgNTaT = 0;  for (deque<simProcess>::iterator it =exitProcs.begin(); it != exitProcs.end(); it++) {  avgNTaT += (\*it).getNormalizedTurnaround();  }  time[q]=static\_cast<double>(avgNTaT) / exitProcs.size();  cout << " average NT "  << time[q] << endl;  exitProcs.clear();  }  //find the time quantum that use least time  double minTime=time[1];  int i;  for(i=1;i<=110;i++){  if(minTime>time[i]){  minTime=time[i];  q=i;  }  }  cout<<"the least time is"<<endl  <<minTime<<endl  <<"the time quantum is"<<endl  <<q<<endl; |

## Flow Chart



Results

Conclusion

By analyzing the result of the time quantum algorithm, we can find it that, the best time quantum of these data is 5. The average of the normalized time decreased at the beginning, however increase afterwards. And we can find that the best time quantum shown on the screen is 5, the least average time is 7.4444s.

By comparing with the FIFO algorithm, we can see that round robin algorithm is good at some time. Choosing the best time quantum is important, If the quantum is very short, then short processes will move through the system relatively quickly. On the other hand, there is processing overhead involved in handling the clock interrupt and performing the scheduling and dispatching function. Thus, very short time quanta should be avoided.