Austin Brown

CPE 434-01

1/28/2021

Lab 7

**Theory**

Process scheduling is what allows multiple processes to run on one CPU. This is important because it is what allows the operating system and applications to be run at the same time. We cover two types of scheduling in this lab. They are priority scheduling and round robin scheduling.

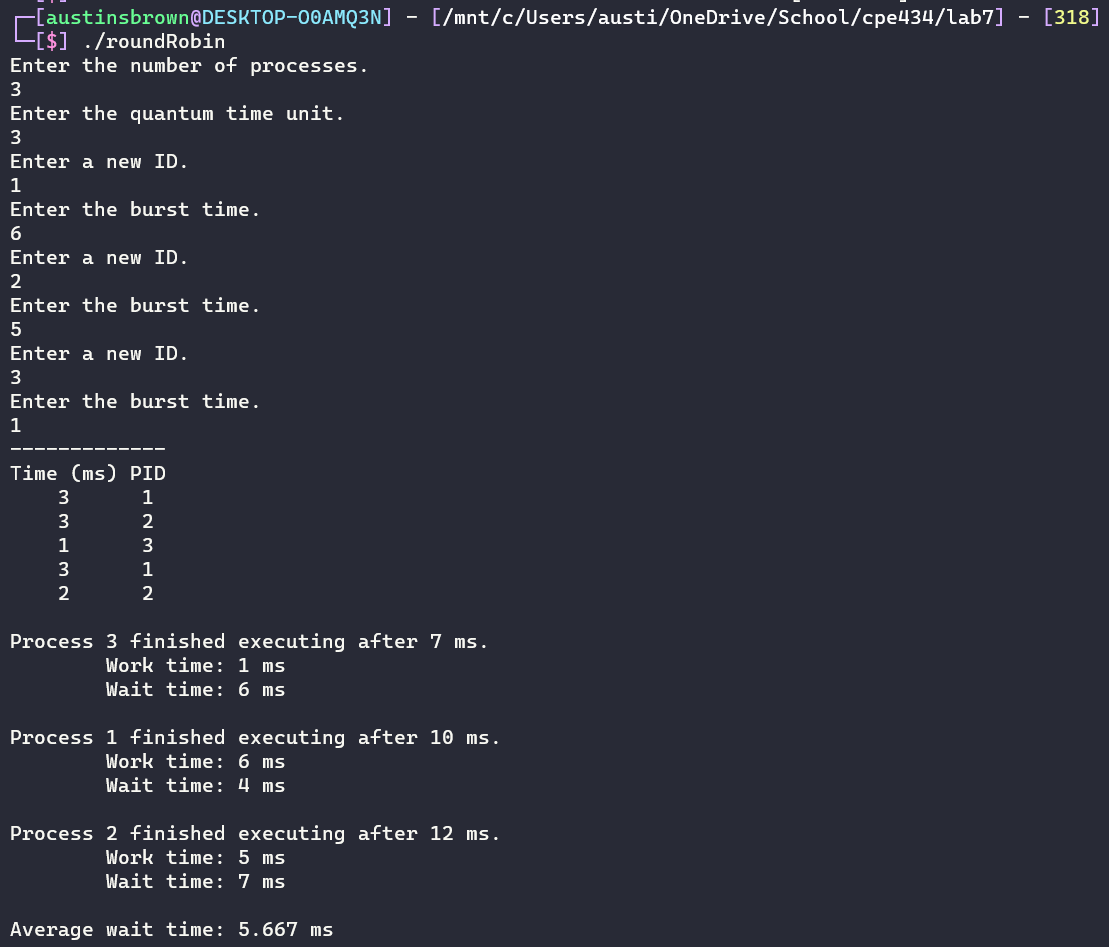
In round robin scheduling, every process is given the same quantum time. This is how long the process must execute. This is an example of preemptive scheduling. This just means that if a process does not finish in the correct amount of time, then it will be paused and will continue in the next time slot.

Priority scheduling does not give processes a limited amount of time to execute. They can take all the time that they need. Processes with higher priorities will be executed first. If a new process comes in that has a higher priority than the one being currently executed, then the CPU will begin executing the new process.

Preemptive scheduling is used when a process transitions from running to ready or from waiting to ready. Processes run for a certain amount of time and then you transition to a different process. Non-preemptive is where a process executes until it ends.

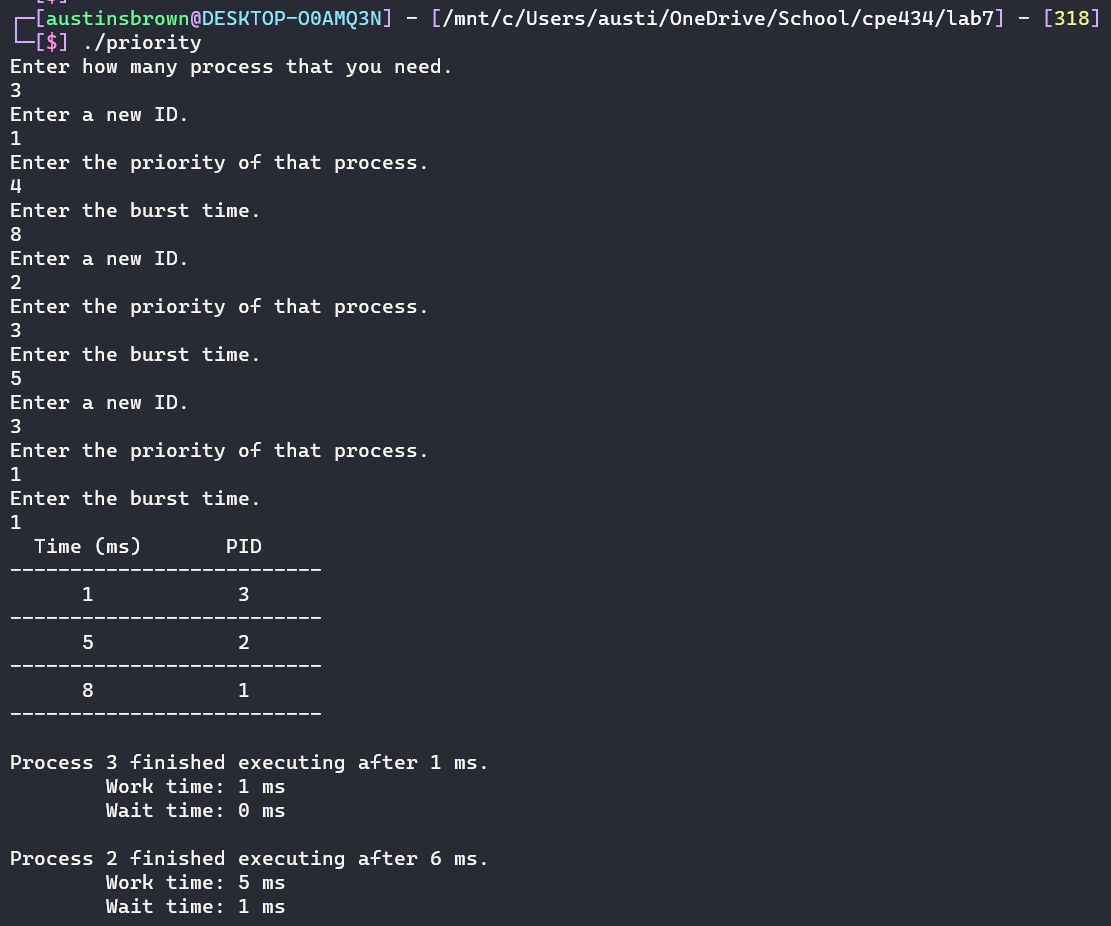
**Results**

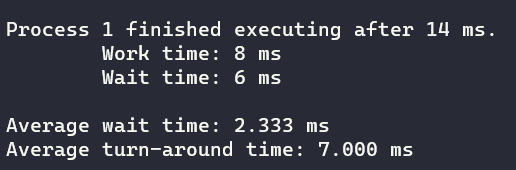
Round Robin

****

****

Priority





**Appendix**

**priority.c**

|  |
| --- |
| #include <stdio.h>  #include <string.h>  #include <stdlib.h>  typedef *struct* Process  {  *int* pid; *// process ID*  *int* priority;  *int* burst\_time; *// CPU burst time*  *int* working\_time; *// time this process executed, if working\_time==burst\_time, process is complete*  *int* t\_round; *// turn around time, time needed for the process to complete*  }Process;  *int* promptUser(*char* \*); *// prompts the user for input*  *int* checkID(*int*, Process *[]*); *// returns 0 if the process processList does not have a repeat, returns -1 otherwise*  *void* sortPriorityprocessList(Process *[]*, *int*); *// sorts the priority processList based on priority*  *void* getResults(Process *[]*, *int*); *// calculates results and prints them in table format*  *int* main()  {  *int* processNum = promptUser("Enter how many process that you need.\n");      Process processList[processNum]; *// processList of all of the processes*        for(*int* i=0; i<processNum; i++)      {          processList[i].pid = promptUser("Enter a new ID.\n");          while(checkID(i, processList) == -1) *// keeps user from entering a repeat ID*              processList[i].pid = promptUser("Enter a new ID.\n");            processList[i].priority = promptUser("Enter the priority of that process.\n");          processList[i].burst\_time = promptUser("Enter the burst time.\n");          processList[i].working\_time = 0;          processList[i].t\_round = 0;      }      sortPriorityprocessList(processList, processNum); *// sort the processList*      getResults(processList, processNum);      return 0;  }  *int* promptUser(*char* \*prompt)  {      printf("%s", prompt);  *char* string[200];      fgets(string, 200, stdin); *// get the user input*      string[strcspn(string, "\n")] = 0; *// remove th endline character*      return atoi(string); *// return an interger version of the input*  }  *int* checkID(*int* index, Process processList*[]*)  {  *int* process = processList[index].pid; *// get the pid at the specified index*      for(*int* i=0; i<index; i++)      {          if(process == processList[i].pid)          {              printf("%d has already been used.\n Cannot have two of the same PID.\n", process);              return -1;          }      }      return 0;  }  *void* sortPriorityprocessList(Process processList*[]*, *int* size)  {      Process temp; *// used for sorting*      for(*int* i=0; i<size-1; i++)      {          for(*int* ii=0; ii<size-1; ii++)          {              if(processList[ii].priority > processList[ii+1].priority) *// if left element is greater than right element*              {                  temp = processList[ii]; *// store i in a temp variable*                  processList[ii] = processList[ii+1]; *// swap ii and ii+1*                  processList[ii+1] = temp;              }          }      }  }  *void* getResults(Process list*[]*, *int* num)  {      printf("  Time (ms)       PID     \n");      printf("--------------------------\n");  *int* elapsedTime = 0;      for (*int* i=0; i<num; i++)      {          list[i].working\_time += list[i].burst\_time;          list[i].t\_round += list[i].burst\_time;          list[i].t\_round += elapsedTime;          elapsedTime += list[i].working\_time;          printf("  %5d", list[i].burst\_time);          printf(" %12d\n", list[i].pid);          printf("--------------------------\n");      }  *double* averageWait = 0;  *double* averageTurn = 0;      for (*int* i=0; i<num; i++)      {          averageWait += list[i].t\_round-list[i].working\_time;          averageTurn += list[i].t\_round;          printf("\nProcess %d finished executing after %d ms.\n", list[i].pid, list[i].t\_round);          printf("    Work time: %d ms\n", list[i].working\_time);          printf("    Wait time: %d ms\n", list[i].t\_round-list[i].working\_time);      }      averageWait = averageWait/num;      averageTurn = averageTurn/num;      printf("\nAverage wait time: %.3f ms\n", averageWait);      printf("Average turn-around time: %.3f ms\n\n", averageTurn);  } |

**roundRobin.c**

|  |
| --- |
| #include <stdio.h>  #include <string.h>  #include <stdlib.h>  typedef *struct* *Process*  {  *int* pid; *// process ID*  *int* burst\_time; *// CPU Burst Time*  *int* working\_time; *// time this process executed, if working\_time == burst\_time, process is complete*  *int* t\_round; *// turn around time, time needed for the process to complete*  *int* sleep\_time;  }*Process*;  *int* promptUser(*char* \*); *// prompt the user for input, return it as an integer*  *int* populateArray(*Process* *[]*, *int*); *// popuate the process array*  *int* checkID(*Process* *[]*, *int*); *// check for duplicate id's in the process array*  *void* removeFromList(*Process* *[]*, *int*, *int*);  *int* main()  {  *int* processNum = promptUser("Enter the number of processes.\n");  *int* quantumTimeUnit = promptUser("Enter the quantum time unit.\n");  *Process* todo[processNum];  *Process* done[processNum];  *int* processListSize = populateArray(todo, processNum);      printf("-------------\n");      printf("Time (ms) PID\n");  *int* elapsedTime = 0;  *int* i, ii;      while(processListSize != 0)      {  *int* remainingTime = todo[i].burst\_time - todo[i].working\_time;          if(remainingTime < quantumTimeUnit)          {              elapsedTime += remainingTime;              todo[i].working\_time += remainingTime;              todo[i].t\_round += elapsedTime - todo[i].sleep\_time;              printf("%5d", remainingTime);          }          else          {              elapsedTime += quantumTimeUnit;              todo[i].working\_time += quantumTimeUnit;              todo[i].t\_round += elapsedTime - todo[i].sleep\_time;              printf("%5d", quantumTimeUnit);          }          printf("%7d\n", todo[i].pid);          if(todo[i].working\_time >= todo[i].burst\_time)          {              done[ii] = todo [i];              ii++;              removeFromList(todo, i+1, processListSize);              processListSize--;              i--;          }          else              todo[i].sleep\_time = elapsedTime;          if(++i >= processListSize)              i = 0;      }  *double* averageWait = 0;  *double* averageTurnaround = 0;      for (*int* i = 0; i < ii; i++)      {          averageWait += done[i].t\_round - done[i].working\_time;          averageTurnaround += done[i].t\_round;          printf("\nProcess %d finished executing after %d ms.\n", done[i].pid, done[i].t\_round);          printf("    Work time: %d ms\n", done[i].working\_time);          printf("    Wait time: %d ms\n", done[i].t\_round - done[i].working\_time);      }      averageWait = averageWait/ii;      averageTurnaround = averageTurnaround/ii;      printf("\nAverage wait time: %.3f ms\n", averageWait);      printf("Average turn-around time: %.3f ms\n\n", averageTurnaround);        return 0;  }  *int* promptUser(*char* \*prompt)  {      printf("%s", prompt);  *char* string[200];      fgets(string, 200, stdin); *// get the user input*      string[strcspn(string, "\n")] = 0; *// remove th endline character*      return atoi(string); *// return an interger version of the input*  }  *int* checkID(*Process* processList*[]*, *int* index)  {  *int* process = processList[index].pid; *// get the pid at the specified index*      for(*int* i=0; i<index; i++)      {          if(process == processList[i].pid)          {              printf("%d has already been used.\n Cannot have two of the same PID.\n", process);              return -1;          }      }      return 0;  }  *int* populateArray(*Process* list*[]*, *int* num)  {  *int* size = 0;      for(*int* i=0; i<num; i++)      {          list[i].pid = promptUser("Enter a new ID.\n");          while(checkID(list, i) == -1) *// keeps user from entering a repeat ID*              list[i].pid = promptUser("Enter a new ID.\n");            size++;          list[i].burst\_time = promptUser("Enter the burst time.\n");          list[i].working\_time = 0;          list[i].t\_round = 0;          list[i].sleep\_time = 0;      }      return size;  }  *void* removeFromList(*Process* processList*[]*, *int* index, *int* n)  {      for(*int* i=index-1; i<n; i++)          processList[i] = processList[i+1];  } |