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

```
┌──[austinsbrown@DESKTOP-O0AMQ3N] - [/mnt/c/Users/austi/OneDrive/School/cpe434/lab7] - [318]
└─[$] ./roundRobin
Enter the number of processes.
3
Enter the quantum time unit.
3
Enter a new ID.
1
Enter the burst time.
6
Enter a new ID.
2
Enter the burst time.
5
Enter a new ID.
3
Enter the burst time.
1
-------------
Time (ms) PID
    3      1
    3      2
    1      3
    3      1
    2      2

Process 3 finished executing after 7 ms.
        Work time: 1 ms
        Wait time: 6 ms

Process 1 finished executing after 10 ms.
        Work time: 6 ms
        Wait time: 4 ms

Process 2 finished executing after 12 ms.
        Work time: 5 ms
        Wait time: 7 ms

Average wait time: 5.667 ms
```

```
Average wait time: 5.667 ms
Average turn-around time: 9.667 ms
```

Priority

```
┌──[austinsbrown@DESKTOP-O0AMQ3N] - [/mnt/c/Users/austi/OneDrive/School/cpe434/lab7] - [318]
└─[$] ./priority
Enter how many process that you need.
3
Enter a new ID.
1
Enter the priority of that process.
4
Enter the burst time.
8
Enter a new ID.
2
Enter the priority of that process.
3
Enter the burst time.
5
Enter a new ID.
3
Enter the priority of that process.
1
Enter the burst time.
1
  Time (ms)       PID
--------------------------
      1            3
--------------------------
      5            2
--------------------------
      8            1
--------------------------

Process 3 finished executing after 1 ms.
        Work time: 1 ms
        Wait time: 0 ms

Process 2 finished executing after 6 ms.
        Work time: 5 ms
        Wait time: 1 ms
```

```
Process 1 finished executing after 14 ms.
        Work time: 8 ms
        Wait time: 6 ms

Average wait time: 2.333 ms
Average turn-around time: 7.000 ms
```

Appendix

priority.c

```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 t
hat 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 pro
cessList
    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 speci
fied index
    for(int i=0; i<index; i++)
    {
        if(process == processList[i].pid)
        {
            printf("%d has already been used.\n Cannot have two of th
e 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 i
i+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>
```

```c
#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 proc
ess 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].wo
rking_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 speci
fied index
    for(int i=0; i<index; i++)
    {
        if(process == processList[i].pid)
        {
            printf("%d has already been used.\n Cannot have two of th
e 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];
}
```