Name: Ningyuan Zhang

Lab Section: G

**Lab 5 A: Process Scheduling Report**

**Summary**

In this lab, I learned the three different process scheduling methods that the Linux Operating System often uses in process scheduling. They are First Come First Serve (FCFS), Shortest First (SF) and Robin Round(RR). Each of them is organized by their own rules. As covered in class, FCFS will process the process that with the first arrive time, SF will focus on the process with shortest process time, and RR will give each process a certain time to process but after the certain time, the current process must yield to the next coming process according to the arrive times. I implement the three scheduling methods with C programming language as well as a required modified RR method. All source codes were included in the file scheduling.c, along with a make file called *makefile*.

Source code:

// to ensure if you could not see the file

/\*\*

\*    Filename: scheduling.c

\* Description: four different process scheduling algorithms implement

\* Version: 1.0

\* Created: 10.26.2017 22h05min23s

\* Author: Ningyuan Zhang （狮子劫博丽）(elithz), elithz@iastate.edu

\* Company: NERVE Software

\*/

#include <stdio.h>

#include <string.h>

#define NUM\_PROCESSES 20

struct process

{

/\* Values initialized for each process \*/

int arrivaltime; /\* Time process arrives and wishes to start \*/

int runtime; /\* Time process requires to complete job \*/

int priority; /\* Priority of the process \*/

/\* Values algorithm may use to track processes \*/

int starttime;

int endtime;

int flag;

int remainingtime;

};

/\* Forward declarations of Scheduling algorithms \*/

void first\_come\_first\_served(struct process \*proc);

void shortestRemainTime(struct process \*proc);

void round\_robin(struct process \*proc);

void round\_robin\_priority(struct process \*proc);

int main()

{

int i;

struct process proc[NUM\_PROCESSES], /\* List of processes \*/

proc\_copy[NUM\_PROCESSES]; /\* Backup copy of processes \*/

/\* Seed random number generator \*/

/\*srand(time(0));\*/ /\* Use this seed to test different scenarios \*/

srand(0xC0FFEE); /\* Used for test to be printed out \*/

/\* Initialize process structures \*/

for (i = 0; i < NUM\_PROCESSES; i++)

{

proc[i].arrivaltime = rand() % 100;

proc[i].runtime = (rand() % 30) + 10;

proc[i].priority = rand() % 3;

proc[i].starttime = 0;

proc[i].endtime = 0;

proc[i].flag = 0;

proc[i].remainingtime = 0;

}

/\* Show process values \*/

printf("Process\tarrival\truntime\tpriority\n");

for (i = 0; i < NUM\_PROCESSES; i++)

printf("%d\t%d\t%d\t%d\n", i, proc[i].arrivaltime, proc[i].runtime,

proc[i].priority);

/\* Run scheduling algorithms \*/

printf("\n\nFirst come first served\n");

memcpy(proc\_copy, proc, NUM\_PROCESSES \* sizeof(struct process));

first\_come\_first\_served(proc\_copy);

printf("\n\nShortest remaining time\n");

memcpy(proc\_copy, proc, NUM\_PROCESSES \* sizeof(struct process));

shortest\_remaining\_time(proc\_copy);

printf("\n\nRound Robin\n");

memcpy(proc\_copy, proc, NUM\_PROCESSES \* sizeof(struct process));

round\_robin(proc\_copy);

printf("\n\nRound Robin with priority\n");

memcpy(proc\_copy, proc, NUM\_PROCESSES \* sizeof(struct process));

round\_robin\_priority(proc\_copy);

return 0;

}

void first\_come\_first\_served(struct process \*proc)

{

/\* Implement scheduling algorithm here \*/

//counters

int i, j;

//int to store running total of completion time

int totalComRunTime = 0;

//int to store average completion time

int avgComRunTime;

//int to store system time

int sysTime = 0;

//keep track of first come

int firstCome;

//loop through num processes to execute them

for (i = 0; i < NUM\_PROCESSES; i++)

{

//initialize firstCome

firstCome = i;

//loop through and find first come

for (j = 0; j < NUM\_PROCESSES; j++)

{

//if firstCome is already complete (flag = 1) then overwrite it

if (proc[firstCome].flag)

firstCome = j;

//if firstCome arrival time is after current process

//and current process hasn't been executed (flag = 0)

//then overwrite it

else if (proc[firstCome].arrivaltime > proc[j].arrivaltime && !proc[j].flag)

firstCome = j;

}

//advance system time if it hasn't caught up to arrival time

if (sysTime < proc[firstCome].arrivaltime)

sysTime = proc[firstCome].arrivaltime;

//execute process

//set firstCome start time

proc[firstCome].starttime = sysTime;

//advance system time

sysTime += proc[firstCome].runtime;

//set firstCome end time

proc[firstCome].endtime = sysTime;

//keep track of completion time running total

totalComRunTime += (proc[firstCome].endtime - proc[firstCome].arrivaltime);

//mark firstCome as completed

proc[firstCome].flag = 1;

//print process star and finish

printf("Process %d started at time %d\n", firstCome, proc[firstCome].starttime);

printf("Process %d finished at time %d\n", firstCome, proc[firstCome].endtime);

}

//calculate average completion time

avgComRunTime = totalComRunTime / NUM\_PROCESSES;

//print out average arrival to finish time

printf("Average time from arrival to completion is %d seconds\n", avgComRunTime);

}

void shortest\_remaining\_time(struct process \*proc)

{

/\* Implement scheduling algorithm here \*/

//counters

int i, j;

//int to store running total of completion time

int totalComRunTime = 0;

//int to store average completion time

int avgComRunTime;

//int to store system time

int sysTime = 0;

//keep track of first come

int shortestRemainTime;

//loop through num processes to execute them

for (i = 0; i < NUM\_PROCESSES; i++)

{

//initialize shortestRemainTime

shortestRemainTime = -1;

//loop through and find shortest remaining time

for (j = 0; j < NUM\_PROCESSES; j++)

{

//if shortest\_remaining time hasn't been set, the

//current process has arrived (arrival time <= system

//time) then overwrite it, and the current process

//hasn't finished\*/

if (shortestRemainTime < 0 && proc[j].arrivaltime <= sysTime && !proc[j].flag)

shortestRemainTime = j;

//else if shortest remaining time has been set, the

//current process has arrived, the current process

//runtime is shorter than shortest remaining time, and

//current process hasn't finished than overwrite it\*/

else if (shortestRemainTime >= 0 && proc[j].arrivaltime <= sysTime && proc[j].runtime < proc[shortestRemainTime].runtime && !proc[j].flag)

shortestRemainTime = j;

}

//if no process was found advance system time and continue

if (shortestRemainTime < 0)

{

sysTime++;

i--;

continue;

}

//execute process

//set shortestRemainTime start time

proc[shortestRemainTime].starttime = sysTime;

//advance system time

sysTime += proc[shortestRemainTime].runtime;

//set shortestRemainTime end time

proc[shortestRemainTime].endtime = sysTime;

//keep track of completion time running total

totalComRunTime += (proc[shortestRemainTime].endtime - proc[shortestRemainTime].arrivaltime);

//mark shortestRemainTime as completed

proc[shortestRemainTime].flag = 1;

//print process star and finish

printf("Process %d started at time %d\n", shortestRemainTime, proc[shortestRemainTime].starttime);

printf("Process %d finished at time %d\n", shortestRemainTime, proc[shortestRemainTime].endtime);

}

//calculate average completion time

avgComRunTime = totalComRunTime / NUM\_PROCESSES;

//print out average arrival to finish time

printf("Average time from arrival to completion is %d seconds\n", avgComRunTime);

}

void round\_robin(struct process \*proc)

{

/\* Implement scheduling algorithm here \*/

//counters

int i, j = 0;

//int to store which process id we searched first at current system time

int start\_j = 0;

//int to store running total of completion time

int totalComRunTime = 0;

//int to store average completion time

int avgComRunTime;

//int to store system time

int sysTime = 0;

//int that is 0 until a job completes

int procFinish;

//loop through till all processes have completed

for (i = 0; i < NUM\_PROCESSES; i++)

{

//initialize procFinish

procFinish = 0;

while (!procFinish)

{

//if proc[j] has arrived and has not completed then run it for 1 second

if (proc[j].arrivaltime <= sysTime && proc[j].flag != 2)

{

//if proc[j] just started running initialize it

if (!proc[j].flag)

{

proc[j].flag = 1; //process started

proc[j].starttime = sysTime;

proc[j].remainingtime = proc[j].runtime - 1;

}

//else update process

else

{

proc[j].remainingtime--;

//if proc is finished update proc

if (!proc[j].remainingtime)

{

//process has completed

proc[j].flag = 2;

proc[j].endtime = sysTime + 1;

procFinish = 1;

totalComRunTime += (proc[j].endtime - proc[j].arrivaltime);

printf("Process %d started at "

"time %d\n",

j, proc[j].starttime);

printf("Process %d finished "

"at time %d\n",

j, proc[j].endtime);

}

}

//update j and increment system time

j = (j < (NUM\_PROCESSES - 1)) ? (j + 1) : 0;

sysTime++;

start\_j = j;

}

//if proc[j] can't be ran

else

{

//update j

j = (j < (NUM\_PROCESSES - 1)) ? (j + 1) : 0;

//if j = start\_j then increment system time since no process could be run at this time

if (j == start\_j)

sysTime++;

}

}

}

//calculate average completion time

avgComRunTime = totalComRunTime / NUM\_PROCESSES;

//print out average arrival to finish time

printf("Average time from arrival to completion is %d seconds\n", avgComRunTime);

}

void round\_robin\_priority(struct process \*proc)

{

/\* Implement scheduling algorithm here \*/

//counters

int i, j = 0;

//int to store which process id we searched first at current system time

int start\_j = 0;

//int to store running total of completion time

int totalComRunTime = 0;

//int to store average completion time

int avgComRunTime;

//int to store system time

int sysTime = 0;

//int that is 0 until a job completes

int procFinish;

//int that stores highest priority

int greatPrior;

//int that stores last executed process

int last\_executed = 0;

//loop through till all processes have completed

for (i = 0; i < NUM\_PROCESSES; i++)

{

//initialize procFinish

procFinish = 0;

while (!procFinish)

{

//initialize greatPrior

greatPrior = -1;

do

{

//if proc[j] has arrived, is not done, and highest priority has not been set, set highest priority

if (proc[j].arrivaltime <= sysTime && greatPrior < 0 && proc[j].flag < 2)

greatPrior = j;

//if proc[j] has arrived, is not done, and is of higher priority than highest priority, update greatPrior

else if (proc[j].arrivaltime <= sysTime && proc[j].flag < 2 && proc[j].priority > proc[greatPrior].priority)

greatPrior = j;

j = (j < (NUM\_PROCESSES - 1)) ? (j + 1) : 0;

} while (j != start\_j);

//if highest\_priority has been set execute

if (greatPrior > -1)

{

last\_executed = greatPrior;

//if greatPrior just started running initialize it

if (!proc[greatPrior].flag)

{

//process started

proc[greatPrior].flag = 1;

proc[greatPrior].starttime = sysTime;

proc[greatPrior].remainingtime = proc[greatPrior].runtime - 1;

}

//else update process

else

{

proc[greatPrior].remainingtime--;

//if proc is finished update proc

if (!proc[greatPrior].remainingtime)

{

//process has completed

proc[greatPrior].flag = 2;

proc[greatPrior].endtime = sysTime + 1;

procFinish = 1;

totalComRunTime += (proc[greatPrior].endtime - proc[greatPrior].arrivaltime);

printf("Process %d started at "

"time %d\n",

greatPrior, proc[greatPrior].starttime);

printf("Process %d finished "

"at time %d\n",

greatPrior, proc[greatPrior].endtime);

}

}

}

//increment system time and set j to search from current highest priority job

sysTime++;

if (last\_executed == NUM\_PROCESSES - 1)

start\_j = j = 0;

else

start\_j = j = (last\_executed + 1);

}

}

//calculate average completion time

avgComRunTime = totalComRunTime / NUM\_PROCESSES;

//print out average arrival to finish time

printf("Average time from arrival to completion is %d seconds\n", avgComRunTime);

}

//Along with make file:

CC=gcc

CFLAGS=-g

ALL=scheduling

all: $(ALL)

scheduling: scheduling.c

    $(CC) $(CFLAGS) -o scheduling scheduling.c

clean:

    rm -f $(ALL) \*.o