//Prac 1

/\*

a) same program, same code.

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main()

{

int a;

a = fork();

if (a < 0)

{

printf("child process could not be created");

exit(-1);

}

else

{

printf("My ID is %d, My parent is %d\n", getpid(), getppid());

}

return 0;

}

/\*

b) same program, different code

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main()

{

int a;

a = fork();

if (a < 0)

{

printf("child process could not be created");

exit(-1);

}

else if (a == 0)

{

printf("Child process : My ID is %d\n", getpid());

}

else

{

printf("Parent process : My ID is %d\n", getpid());

}

return 0;

}

/\*

c) different programs

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main()

{

int a;

a = fork();

if (a < 0)

{

printf("child process could not be created");

exit(-1);

}

else if (a == 0)

{

execl("/bin/ls", "ls", NULL);

}

else

{

printf("Parent process : My ID is %d\n", getpid());

}

return 0;

}

/\*

d) before terminating, the parent waits for the child to finish

its task

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

int main() {

int a;

a=fork();

if(a<0)

{

printf("child process could not be created");

exit(-1);

}

else if(a==0)

{

printf("Child process : My ID is %d\n",getpid());

}

else

{

wait(NULL);

printf("Parent process : My ID is %d\n",getpid());

}

return 0;

}

/\*

2. Write a program to report behaviour of Linux kernel including kernel version, CPU type and model. (CPU Information)

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(){

printf("\nKernel Version:\n");

system("cat /proc/sys/kernel/osrelease");

printf("\n-- CPU ----\n");

system("cat /proc/cpuinfo | awk 'NR==3{print}'");

system("cat /proc/cpuinfo | awk 'NR==4{print}'");

system("cat /proc/cpuinfo | awk 'NR==5{print}'");

printf("\n");

return 0;

}

/\*

3. Write a program to report behaviour of Linux kernel including information on configured memory, amount of free and used memory. (Memory Information)

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(){

printf("\nKernel Version:\n");

system("cat /proc/sys/kernel/osrelease");

printf("\nConfigured Memory:\n");

system("cat /proc/meminfo | awk 'NR==1{print $2}'");

printf("\nAmount of Free Memory:\n");

system("cat /proc/meminfo | awk 'NR==2{print $2}'");

printf("\nAmount of Used Memory:\n");

system("cat /proc/meminfo | awk '{if(NR==1) a=$2; if(NR==2) b=$2} END {print a-b}'");

return 0;

}

/\*

Write a program to print file details including owner access permissions,file access time, where file name is given as argument.

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

int main(int argc, char \*\*argv)

{

if (argc < 2)

{

fprintf(stderr, "Correct Usage: ./main <filename>\n");

return -1;

}

struct stat dt;

printf("\nFile Info\n");

printf("--------------------\n");

printf("Name: %s\n", argv[1]);

printf("UserID: %d\n", dt.st\_uid);

printf("GroupID: %d\n", dt.st\_gid);

printf("File Type: %d\n", S\_IFMT);

printf("Directory: %s\n", S\_IFDIR ? "Yes" : "No");

printf("Regular File: %s\n", S\_IFREG ? "Yes" : "No");

printf("Last access time: %ld\n", dt.st\_atime);

printf("Last modified time: %ld\n", dt.st\_mtime);

printf("User Permissions:\n");

printf(" Read: %s\n", S\_IRUSR ? "Yes" : "No");

printf(" Write: %s\n", S\_IWUSR ? "Yes" : "No");

printf(" Execute: %s\n", S\_IXUSR ? "Yes" : "No");

printf("Group Permissions:\n");

printf(" Read: %s\n", S\_IRGRP ? "Yes" : "No");

printf(" Write: %s\n", S\_IWGRP ? "Yes" : "No");

printf(" Execute: %s\n", S\_IXGRP ? "Yes" : "No");

printf("Others Permissions:\n");

printf(" Read: %s\n", S\_IROTH ? "Yes" : "No");

printf(" Write: %s\n", S\_IWOTH ? "Yes" : "No");

printf(" Execute: %s\n\n", S\_IXOTH ? "Yes" : "No");

}

/\*

5. Write a program to copy a source file into the target file and display the target file using system calls.

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

int main(int argc, char \*argv[])

{

if (argc < 3)

{

fprintf(stderr, "Correct Usage: ./main <src\_filename> <src\_filename>\n");

return -1;

}

char buf;

int fd1, fd2, n;

if ((fd1 = open(argv[1], O\_RDONLY)) < 0)

{

fprintf(stderr, "Could not read %s\n", argv[1]);

return 2;

}

if ((fd2 = creat(argv[2], 0666)) < 0)

{

fprintf(stderr, "Could not write to %s\n", argv[2]);

return 2;

}

while ((n = read(fd1, &buf, 1)) > 0)

write(fd2, &buf, 1);

printf("Copied contents of %s to %s\n", argv[1], argv[2]);

close(fd1);

close(fd2);

return 0;

}

/\*

6. Write a program to implement FCFS scheduling algorithm.

\*/

#include <iostream>

#include <stdlib.h>

using namespace std;

struct Process {

int pID;

float arrivalTime;

float burstTime;

float completionTime;

float waitingTime;

float turnAroundTime;

};

void calcCompletionTime(struct Process \*p, int n)

{

p[0].completionTime = p[0].burstTime;

for(int i=1; i<n; ++i)

{

p[i].completionTime = p[i-1].completionTime + p[i].burstTime;

}

return;

}

void calcTurnAroundTime(struct Process \*p, int n)

{

for(int i=0; i<n; ++i)

{

p[i].turnAroundTime = p[i].completionTime - p[i].arrivalTime;

}

return;

}

void calcWaitingTime(struct Process \*p, int n)

{

for(int i=0; i<n; ++i)

{

p[i].waitingTime = p[i].turnAroundTime - p[i].burstTime;

}

return;

}

void printAvgTime(struct Process \*p, int n)

{

calcCompletionTime(p, n);

calcTurnAroundTime(p, n);

calcWaitingTime(p, n);

// Printing Process Info

cout << " ----------------------------" << endl;

cout << "\n process -> { arrivalTime, burstTime, completionTime, turnAroundTime, waitingTime }\n";

for (int i=0; i<n; ++i)

{

cout << " " << p[i].pID << " -> { " << p[i].arrivalTime << " , " << p[i].burstTime << " , " << p[i].completionTime << " , " << p[i].turnAroundTime << " , " << p[i].waitingTime << " }\n";

}

// Calculating sum of waitingTime and turnAroundTime

float sumW = 0.0;

float sumT = 0.0;

for (int i=0; i<n; ++i)

{

sumW += p[i].waitingTime;

sumT += p[i].turnAroundTime;

}

// Printing average waitingTime and turnAroundTime

cout << "\n Average Waiting Time: " << sumW/n;

cout << "\n Average Turn Around Time: " << sumT/n << endl;

return;

}

int main()

{

int n;

cout << "\n Enter number of Processes: ";

cin >> n;

cout << endl;

struct Process p[n];

for(int i=0; i<n; ++i)

{

p[i].pID = i+1;

cout << " Enter Arrival Time of Process " << i+1 << ": ";

cin >> p[i].arrivalTime;

cout << " Enter Burst Time of Process " << i+1 << ": ";

cin >> p[i].burstTime;

cout << endl;

}

printAvgTime(p, n);

cout << endl;

return 0;

}

/\*

7. Write a program to implement Round Robin scheduling algorithm.

\*/

#include <iostream>

#include <stdlib.h>

using namespace std;

struct Process {

int pID;

float arrivalTime;

float burstTime;

float waitingTime;

float turnAroundTime;

};

void swapProcess(struct Process \*a, struct Process \*b)

{

struct Process temp = \*a;

\*a = \*b;

\*b = temp;

}

void sortAccArrivalTime(struct Process \*p, int n)

{

for(int i=0; i < n-1; ++i)

{

if(p[i].arrivalTime > p[i+1].arrivalTime)

{

swapProcess(&p[i], &p[i+1]);

}

}

return;

}

void sortAccPID(struct Process \*p, int n)

{

for(int i=0; i < n-1; ++i)

{

if(p[i].pID > p[i+1].pID)

{

swapProcess(&p[i], &p[i+1]);

}

}

return;

}

void calcWaitingTime(struct Process \*p, int n, float quantum)

{

float \*remainingTime = new float[n];

for(int i=0; i<n; ++i)

remainingTime[i] = p[i].burstTime;

float temp = 0;

while(true)

{

int flag = 1;

for(int i=0; i<n; ++i)

{

if (remainingTime[i] > 0)

{

flag = 0;

if (remainingTime[i] > quantum)

{

temp += quantum;

remainingTime[i] -= quantum;

}

else

{

temp += remainingTime[i];

p[i].waitingTime += temp - p[i].arrivalTime - p[i].burstTime;

remainingTime[i] = 0;

}

}

}

if (flag ==1) break;

}

return;

}

void calcTurnAroundTime(struct Process \*p, int n)

{

for(int i=0; i<n; ++i)

{

p[i].turnAroundTime = p[i].waitingTime + p[i].burstTime - p[i].arrivalTime;

}

return;

}

void printAvgTime(struct Process \*p, int n, int quantum)

{

sortAccArrivalTime(p, n);

calcWaitingTime(p, n, quantum);

calcTurnAroundTime(p, n);

sortAccPID(p, n);

// Printing Process Info

cout << " Round Robin CPU Scheduling" << endl;

cout << " ------------------------------" << endl;

cout << "\n process -> { arrivalTime, burstTime, turnAroundTime, waitingTime }\n";

for (int i=0; i<n; ++i)

{

cout << " P" << p[i].pID << " -> { " << p[i].arrivalTime << " , " << p[i].burstTime << " , " << p[i].turnAroundTime << " , " << p[i].waitingTime << " }\n";

}

// Calculating sum of waitingTime and turnAroundTime

float sumW = 0.0;

float sumT = 0.0;

for (int i=0; i<n; ++i)

{

sumW += p[i].waitingTime;

sumT += p[i].turnAroundTime;

}

// Printing average waitingTime and turnAroundTime

cout << "\n Average Waiting Time: " << sumW/n;

cout << "\n Average Turn Around Time: " << sumT/n << endl;

return;

}

int main()

{

int n;

float quantum;

cout << "\n Enter number of Processes: ";

cin >> n;

cout << endl;

struct Process p[n];

for(int i=0; i<n; ++i)

{

p[i].pID = i+1;

cout << " Enter Arrival Time of Process " << i+1 << ": ";

cin >> p[i].arrivalTime;

cout << " Enter Burst Time of Process " << i+1 << ": ";

cin >> p[i].burstTime;

cout << endl;

}

cout << " Enter quantum time: ";

cin >> quantum;

cout << endl;

printAvgTime(p, n, quantum);

cout << endl;

return 0;

}

/\*

8. Write a program to implement SJF scheduling algorithm.

\*/

#include <iostream>

#include <stdlib.h>

using namespace std;

struct Process {

int pID;

float arrivalTime;

float burstTime;

float completionTime;

float waitingTime;

float turnAroundTime;

};

void swapProcess(struct Process \*a, struct Process \*b)

{

struct Process temp = \*a;

\*a = \*b;

\*b = temp;

}

void sortAccArrivalTime(struct Process \*p, int n)

{

for(int i=0; i < n-1; ++i)

{

if(p[i].arrivalTime > p[i+1].arrivalTime)

{

swapProcess(&p[i], &p[i+1]);

}

}

return;

}

void sortAccPID(struct Process \*p, int n)

{

for(int i=0; i < n-1; ++i)

{

if(p[i].pID > p[i+1].pID)

{

swapProcess(&p[i], &p[i+1]);

}

}

return;

}

void calcCompletionTime(struct Process \*p, int n)

{

float minBurstTime = p[0].burstTime;

p[0].completionTime = minBurstTime + p[0].arrivalTime;

int temp;

for(int i=1; i < n; ++i)

{

temp = i;

minBurstTime = p[i].burstTime;

for(int j=i; j < n; j++)

{

if(p[i-1].completionTime >= p[j].arrivalTime && minBurstTime > p[j].burstTime)

{

minBurstTime = p[j].burstTime;

temp = j;

}

}

p[temp].completionTime = p[i-1].completionTime + p[temp].burstTime;

swapProcess(&p[temp], &p[i]);

}

return;

}

void calcTurnAroundTime(struct Process \*p, int n)

{

for(int i=0; i<n; ++i)

{

p[i].turnAroundTime = p[i].completionTime - p[i].arrivalTime;

}

return;

}

void calcWaitingTime(struct Process \*p, int n)

{

for(int i=0; i<n; ++i)

{

p[i].waitingTime = p[i].turnAroundTime - p[i].burstTime;

}

return;

}

void printAvgTime(struct Process \*p, int n)

{

sortAccArrivalTime(p, n);

calcCompletionTime(p, n);

calcTurnAroundTime(p, n);

calcWaitingTime(p, n);

sortAccPID(p, n);

// Printing Process Info

cout << " SJF CPU Scheduling" << endl;

cout << " ------------------------------" << endl;

cout << "\n process -> { arrivalTime, burstTime, completionTime, turnAroundTime, waitingTime }\n";

for (int i=0; i<n; ++i)

{

cout << " P" << p[i].pID << " -> { " << p[i].arrivalTime << " , " << p[i].burstTime << " , " << p[i].completionTime << " , " << p[i].turnAroundTime << " , " << p[i].waitingTime << " }\n";

}

// Calculating sum of waitingTime and turnAroundTime

float sumW = 0.0;

float sumT = 0.0;

for (int i=0; i<n; ++i)

{

sumW += p[i].waitingTime;

sumT += p[i].turnAroundTime;

}

// Printing average waitingTime and turnAroundTime

cout << "\n Average Waiting Time: " << sumW/n;

cout << "\n Average Turn Around Time: " << sumT/n << endl;

return;

}

int main()

{

int n;

cout << "\n Enter number of Processes: ";

cin >> n;

cout << endl;

struct Process p[n];

for(int i=0; i<n; ++i)

{

p[i].pID = i+1;

cout << " Enter Arrival Time of Process " << i+1 << ": ";

cin >> p[i].arrivalTime;

cout << " Enter Burst Time of Process " << i+1 << ": ";

cin >> p[i].burstTime;

cout << endl;

}

printAvgTime(p, n);

cout << endl;

return 0;

}

/\*

9. Write a program to implement non-preemptive priority based scheduling algorithm.

\*/

#include <iostream>

#include <stdlib.h>

using namespace std;

struct Process {

int pID;

int priority;

float arrivalTime;

float burstTime;

float completionTime;

float waitingTime;

float turnAroundTime;

};

void swapProcess(struct Process \*a, struct Process \*b)

{

struct Process temp = \*a;

\*a = \*b;

\*b = temp;

}

void sortForExec(struct Process \*p, int n)

{

for(int i=0; i < n-1; ++i)

{

if(p[i].arrivalTime > p[i+1].arrivalTime)

{

swapProcess(&p[i], &p[i+1]);

}

else if(p[i].arrivalTime == p[i+1].arrivalTime)

{

if(p[i].priority > p[i+1].priority)

swapProcess(&p[i], &p[i+1]);

else if(p[i].priority == p[i+1].priority)

{

if(p[i].pID > p[i+1].pID)

swapProcess(&p[i], &p[i+1]);

}

}

}

return;

}

void sortAccPID(struct Process \*p, int n)

{

for(int i=0; i < n-1; ++i)

{

if(p[i].pID > p[i+1].pID)

{

swapProcess(&p[i], &p[i+1]);

}

}

return;

}

void calcCompletionTime(struct Process \*p, int n)

{

p[0].completionTime = p[0].burstTime;

for(int i=1; i<n; ++i)

{

p[i].completionTime = p[i-1].completionTime + p[i].burstTime;

}

return;

}

void calcTurnAroundTime(struct Process \*p, int n)

{

for(int i=0; i<n; ++i)

{

p[i].turnAroundTime = p[i].completionTime - p[i].arrivalTime;

}

return;

}

void calcWaitingTime(struct Process \*p, int n)

{

for(int i=0; i<n; ++i)

{

p[i].waitingTime = p[i].turnAroundTime - p[i].burstTime;

}

return;

}

void printAvgTime(struct Process \*p, int n)

{

sortForExec(p, n);

calcCompletionTime(p, n);

sortAccPID(p, n);

calcTurnAroundTime(p, n);

calcWaitingTime(p, n);

// Printing Process Info

cout << " Non-preemptive Priority Based CPU Scheduling" << endl;

cout << " ------------------------------------------------" << endl;

cout << "\n process -> { priority, arrivalTime, burstTime, completionTime, turnAroundTime, waitingTime }\n";

for (int i=0; i<n; ++i)

{

cout << " P" << p[i].pID << " -> { " << p[i].priority << " , " << p[i].arrivalTime << " , " << p[i].burstTime << " , " << p[i].completionTime << " , " << p[i].turnAroundTime << " , " << p[i].waitingTime << " }\n";

}

// Calculating sum of waitingTime and turnAroundTime

float sumW = 0.0;

float sumT = 0.0;

for (int i=0; i<n; ++i)

{

sumW += p[i].waitingTime;

sumT += p[i].turnAroundTime;

}

// Printing average waitingTime and turnAroundTime

cout << "\n Average Waiting Time: " << sumW/n;

cout << "\n Average Turn Around Time: " << sumT/n << endl;

return;

}

int main()

{

int n;

cout << "\n Enter number of Processes: ";

cin >> n;

cout << endl;

struct Process p[n];

for(int i=0; i<n; ++i)

{

p[i].pID = i+1;

cout << " Enter Priority of Process " << i+1 << ": ";

cin >> p[i].priority;

cout << " Enter Arrival Time of Process " << i+1 << ": ";

cin >> p[i].arrivalTime;

cout << " Enter Burst Time of Process " << i+1 << ": ";

cin >> p[i].burstTime;

cout << endl;

}

printAvgTime(p, n);

cout << endl;

return 0;

}

/\*

10. Write a program to implement preemptive priority based scheduling algorithm.

\*/

#include <iostream>

#include <stdlib.h>

using namespace std;

struct Process {

int pID;

int priority;

int arrivalTime;

int burstTime;

int completionTime;

int waitingTime;

int turnAroundTime;

};

void swapProcess(struct Process \*a, struct Process \*b)

{

struct Process temp = \*a;

\*a = \*b;

\*b = temp;

}

void sortForExec(struct Process \*p, int n)

{

for(int i=0; i < n-1; ++i)

{

if(p[i].arrivalTime > p[i+1].arrivalTime)

{

swapProcess(&p[i], &p[i+1]);

}

else if(p[i].arrivalTime == p[i+1].arrivalTime)

{

if(p[i].priority > p[i+1].priority)

swapProcess(&p[i], &p[i+1]);

else if(p[i].priority == p[i+1].priority)

{

if(p[i].pID > p[i+1].pID)

swapProcess(&p[i], &p[i+1]);

}

}

}

return;

}

void sortAccPID(struct Process \*p, int n)

{

for(int i=0; i < n-1; ++i)

{

if(p[i].pID > p[i+1].pID)

{

swapProcess(&p[i], &p[i+1]);

}

}

return;

}

void calcCompletionTime(struct Process \*p, int n)

{

int remainingTime[n];

for(int i=0; i < n; ++i)

remainingTime[i] = p[i].burstTime;

int minIndex, time=0, count=0;

for(time=0; count!=n; time++)

{

remainingTime[9] = 999;

minIndex = 9;

for(int i = 0; i < n; ++i)

{

if(p[i].arrivalTime <= time && remainingTime[i] > 0 && p[i].priority <= p[minIndex].priority)

{

minIndex = i;

}

}

if(remainingTime[minIndex] <= 0)

continue;

remainingTime[minIndex]--;

if(remainingTime[minIndex] == 0)

{

count++;

p[minIndex].completionTime = time + 1;

}

}

return;

}

void calcTurnAroundTime(struct Process \*p, int n)

{

for(int i=0; i<n; ++i)

{

p[i].turnAroundTime = p[i].completionTime - p[i].arrivalTime;

}

return;

}

void calcWaitingTime(struct Process \*p, int n)

{

for(int i=0; i<n; ++i)

{

p[i].waitingTime = p[i].turnAroundTime - p[i].burstTime;

}

return;

}

void printAvgTime(struct Process \*p, int n)

{

sortForExec(p, n);

calcCompletionTime(p, n);

calcTurnAroundTime(p, n);

calcWaitingTime(p, n);

sortAccPID(p, n);

// Printing Process Info

cout << " Preemptive Priority Based CPU Scheduling" << endl;

cout << " ------------------------------" << endl;

cout << "\n process -> { priority, arrivalTime, burstTime, completionTime, turnAroundTime, waitingTime }\n";

for (int i=0; i<n; ++i)

{

cout << " P" << p[i].pID << " -> { " << p[i].priority << " , " << p[i].arrivalTime << " , " << p[i].burstTime << " , " << p[i].completionTime << " , " << p[i].turnAroundTime << " , " << p[i].waitingTime << " }\n";

}

// Calculating sum of waitingTime and turnAroundTime

int sumW = 0.0;

int sumT = 0.0;

for (int i=0; i<n; ++i)

{

sumW += p[i].waitingTime;

sumT += p[i].turnAroundTime;

}

// Printing average waitingTime and turnAroundTime

cout << "\n Average Waiting Time: " << sumW/n;

cout << "\n Average Turn Around Time: " << sumT/n << endl;

return;

}

int main()

{

int n;

cout << "\n Enter number of Processes: ";

cin >> n;

cout << endl;

struct Process p[n];

for(int i=0; i<n; ++i)

{

p[i].pID = i+1;

cout << " Enter Priority of Process " << i+1 << ": ";

cin >> p[i].priority;

cout << " Enter Arrival Time of Process " << i+1 << ": ";

cin >> p[i].arrivalTime;

cout << " Enter Burst Time of Process " << i+1 << ": ";

cin >> p[i].burstTime;

cout << endl;

}

printAvgTime(p, n);

cout << endl;

return 0;

}

/\*

11. Write a program to implement SRJF scheduling algorithm.

\*/

#include <iostream>

#include <stdlib.h>

using namespace std;

struct Process {

int pID;

int arrivalTime;

int burstTime;

int completionTime;

int waitingTime;

int turnAroundTime;

};

void swapProcess(struct Process \*a, struct Process \*b)

{

struct Process temp = \*a;

\*a = \*b;

\*b = temp;

}

void sortAccArrivalTime(struct Process \*p, int n)

{

for(int i=0; i < n-1; ++i)

{

if(p[i].arrivalTime > p[i+1].arrivalTime)

{

swapProcess(&p[i], &p[i+1]);

}

}

return;

}

void sortAccPID(struct Process \*p, int n)

{

for(int i=0; i < n-1; ++i)

{

if(p[i].pID > p[i+1].pID)

{

swapProcess(&p[i], &p[i+1]);

}

}

return;

}

void calcCompletionTime(struct Process \*p, int n)

{

int remainingTime[n];

for(int i=0; i < n; ++i)

remainingTime[i] = p[i].burstTime;

int minIndex, time=0, count=0;

for(time=0; count!=n; time++)

{

minIndex = 0;

for(int i = 0; i < n; ++i)

if(p[i].arrivalTime <= time && remainingTime[i] < remainingTime[minIndex] && remainingTime[i] > 0)

minIndex = i;

if(remainingTime[minIndex] <= 0)

continue;

remainingTime[minIndex]--;

if(remainingTime[minIndex] == 0)

{

count++;

p[minIndex].completionTime = time + 1;

}

}

return;

}

void calcTurnAroundTime(struct Process \*p, int n)

{

for(int i=0; i<n; ++i)

{

p[i].turnAroundTime = p[i].completionTime - p[i].arrivalTime;

}

return;

}

void calcWaitingTime(struct Process \*p, int n)

{

for(int i=0; i<n; ++i)

{

p[i].waitingTime = p[i].turnAroundTime - p[i].burstTime;

}

return;

}

void printAvgTime(struct Process \*p, int n)

{

sortAccArrivalTime(p, n);

calcCompletionTime(p, n);

calcTurnAroundTime(p, n);

calcWaitingTime(p, n);

sortAccPID(p, n);

// double end;

// int smallest, count = 0;

// double totalWaitingTime = 0.0;

// double totalTurnAroundTime = 0.0;

// struct Process temp[n + 1];

// for (int i = 0; i < n; i++)

// temp[i] = p[i];

// smallest = n + 1;

// temp[smallest].burstTime = 999;

// for (double time = 0; count != n; time++) {

// for (int i = 0; i < n; i++) {

// if (p[i].arrivalTime <= time &&

// p[i].burstTime < temp[smallest].burstTime &&

// p[i].burstTime > 0) {

// smallest = i;

// }

// }

// temp[smallest].burstTime--;

// if (temp[smallest].burstTime == 0) {

// count++;

// end = time + 1;

// p[count].waitingTime +=

// end - p[smallest].arrivalTime - temp[smallest].burstTime;

// p[count].turnAroundTime += end - p[smallest].arrivalTime;

// }

// }

// Printing Process Info

cout << " SRTF CPU Scheduling" << endl;

cout << " ------------------------------" << endl;

cout << "\n process -> { arrivalTime, burstTime, completionTime, turnAroundTime, waitingTime }\n";

for (int i=0; i<n; ++i)

{

cout << " P" << p[i].pID << " -> { " << p[i].arrivalTime << " , " << p[i].burstTime << " , " << p[i].completionTime << " , " << p[i].turnAroundTime << " , " << p[i].waitingTime << " }\n";

}

// Calculating sum of waitingTime and turnAroundTime

int sumW = 0.0;

int sumT = 0.0;

for (int i=0; i<n; ++i)

{

sumW += p[i].waitingTime;

sumT += p[i].turnAroundTime;

}

// Printing average waitingTime and turnAroundTime

cout << "\n Average Waiting Time: " << sumW/n;

cout << "\n Average Turn Around Time: " << sumT/n << endl;

return;

}

int main()

{

int n;

cout << "\n Enter number of Processes: ";

cin >> n;

cout << endl;

struct Process p[n];

for(int i=0; i<n; ++i)

{

p[i].pID = i+1;

cout << " Enter Arrival Time of Process " << i+1 << ": ";

cin >> p[i].arrivalTime;

cout << " Enter Burst Time of Process " << i+1 << ": ";

cin >> p[i].burstTime;

cout << endl;

}

printAvgTime(p, n);

cout << endl;

return 0;

}

/\*\*

12. Write a program to calculate sum of n numbers using thread library.

\*

\*/

#include <cstdlib>

#include <iostream>

#include <pthread.h>

using namespace std;

long long sum;

void \*runner(void \*number);

int main(int argc, char \*\*argv)

{

if (argc != 2)

{

cerr << "Usage: ./main <upper>" << endl;

exit(1);

}

if (atoi(argv[1]) < 0)

{

cerr << "Argument must be non-negative." << endl;

exit(1);

}

pthread\_t tid;

pthread\_attr\_t attr;

pthread\_attr\_init(&attr);

pthread\_create(&tid, &attr, runner, (void \*)argv[1]);

pthread\_join(tid, NULL);

cout << "Sum from 1 to " << atoi(argv[1])

<< " is " << sum << endl;

return 0;

}

void \*runner(void \*upper)

{

int num = atoi((const char \*)(upper));

for (int i = 1; i <= num; i++)

sum += i;

pthread\_exit(0);

return nullptr;

}

//PRAC 13

/\*\*

\* Write a program to implement best-fit allocation strategy

\*

\*/

#include <cstring>

#include <iostream>

#define MAX\_SIZE 100

using namespace std;

void bestFit(int blockSize[], int m,

int processSize[], int n)

{

int allocation[n];

for (int i = 0; i < n; i++)

allocation[i] = -1;

for (int i = 0; i < n; i++)

{

int bestIdx = -1;

for (int j = 0; j < m; j++)

{

if (blockSize[j] >= processSize[i])

{

if (bestIdx == -1)

bestIdx = j;

else if (blockSize[bestIdx] > blockSize[j])

bestIdx = j;

}

}

if (bestIdx != -1)

{

allocation[i] = bestIdx;

blockSize[bestIdx] -= processSize[i];

}

}

cout << "\nBest-Fit Allocation Strategy\n";

cout << "=========================================\n";

cout << "Process No.\tProcess Size\tBlock No.\n";

cout << "=========================================\n";

for (int i = 0; i < n; i++)

{

cout << " " << i + 1 << "\t\t" << processSize[i] << "\t\t";

if (allocation[i] != -1)

cout << allocation[i] + 1;

else

cout << "Not Allocated";

cout << endl;

}

}

int main()

{

int holes, processes;

int holeSizes[MAX\_SIZE], processSizes[MAX\_SIZE];

cout << "Enter Number of Holes: ";

cin >> holes;

cout << "Enter Number of Processes: ";

cin >> processes;

for (int i = 0; i < holes; i++)

{

cout << "Enter Size of Hole " << (i + 1) << ": ";

cin >> holeSizes[i];

}

for (int i = 0; i < processes; i++)

{

cout << "Enter Size of Process " << (i + 1) << ": ";

cin >> processSizes[i];

}

bestFit(holeSizes, holes, processSizes, processes);

return 0;

}

/\*\*

\* Write a program to implement first-fit, best-fit and

\* worst-fit allocation strategies.

\*

\*/

#include <cstring>

#include <iostream>

#define MAX\_SIZE 100

using namespace std;

void firstFit(int blockSize[], int m,

int processSize[], int n)

{

int allocation[n];

for (int i = 0; i < n; i++)

allocation[i] = -1;

for (int i = 0; i < n; i++)

{

for (int j = 0; j < m; j++)

{

if (blockSize[j] >= processSize[i])

{

allocation[i] = j;

blockSize[j] -= processSize[i];

break;

}

}

}

cout << "\nFirst-Fit Allocation Strategy\n";

cout << "=========================================\n";

cout << "\nProcess No.\tProcess Size\tBlock No.\n";

cout << "=========================================\n";

for (int i = 0; i < n; i++)

{

cout << " " << i + 1 << "\t\t"

<< processSize[i] << "\t\t";

if (allocation[i] != -1)

cout << allocation[i] + 1;

else

cout << "Not Allocated";

cout << endl;

}

}

int main()

{

int holes, processes;

int holeSizes[MAX\_SIZE], processSizes[MAX\_SIZE];

cout << "\nEnter Number of Holes: ";

cin >> holes;

cout << "Enter Number of Processes: ";

cin >> processes;

for (int i = 0; i < holes; i++)

{

cout << "Enter Size of Hole " << (i + 1) << ": ";

cin >> holeSizes[i];

}

for (int i = 0; i < processes; i++)

{

cout << "Enter Size of Process " << (i + 1) << ": ";

cin >> processSizes[i];

}

firstFit(holeSizes, holes, processSizes, processes);

cout << endl;

return 0;

}

/\*\*

\* Write a program to implement first-fit, best-fit and

\* worst-fit allocation strategies.

\*

\*/

#include <cstring>

#include <iostream>

#define MAX\_SIZE 100

using namespace std;

void worstFit(int blockSize[], int m,

int processSize[], int n)

{

int allocation[n];

for (int i = 0; i < n; i++)

allocation[i] = -1;

for (int i = 0; i < n; i++)

{

int wstIdx = -1;

for (int j = 0; j < m; j++)

{

if (blockSize[j] >= processSize[i])

{

if (wstIdx == -1)

wstIdx = j;

else if (blockSize[wstIdx] < blockSize[j])

wstIdx = j;

}

}

if (wstIdx != -1)

{

allocation[i] = wstIdx;

blockSize[wstIdx] -= processSize[i];

}

}

cout << "\nWorst-Fit Allocation Strategy\n";

cout << "=========================================\n";

cout << "Process No.\tProcess Size\tBlock No.\n";

cout << "=========================================\n";

for (int i = 0; i < n; i++)

{

cout << " " << i + 1 << "\t\t" << processSize[i] << "\t\t";

if (allocation[i] != -1)

cout << allocation[i] + 1;

else

cout << "Not Allocated";

cout << endl;

}

}

int main()

{

int holes, processes;

int holeSizes[MAX\_SIZE], processSizes[MAX\_SIZE];

cout << "\nEnter Number of Holes: ";

cin >> holes;

cout << "Enter Number of Processes: ";

cin >> processes;

for (int i = 0; i < holes; i++)

{

cout << "Enter Size of Hole " << (i + 1) << ": ";

cin >> holeSizes[i];

}

for (int i = 0; i < processes; i++)

{

cout << "Enter Size of Process " << (i + 1) << ": ";

cin >> processSizes[i];

}

worstFit(holeSizes, holes, processSizes, processes);

cout << endl;

return 0;

}

//I deserve a Party for all this =(