

**设计一个有 N个进程共行的进程调度程序。**

 　进程调度算法：采用时间片轮转调度算法和先来先服务算法

　　每个进程有一个进程控制块（ PCB）表示。进程控制块可以包含如下信息：进程名(pid)、到达时间(arrival\_time)、需要运行时间(burst\_time)、剩余运行时间(remaing\_time)等等。

　　进程的各种信息例如运行时间可以事先人为地指定（也可以由随机数产生）。进程的到达时间为进程输入的时间。

　　进程的运行时间以时间片为单位进行计算。

　　就绪进程获得 CPU后都只能运行一个时间片。用已占用CPU时间加1来表示。

　　如果运行一个时间片后，进程的已占用 CPU时间已达到所需要的运行时间，则撤消该进程，如果运行一个时间片后进程的已占用CPU时间已达所需要的运行时间，则该进程已完成，下次轮转不再考虑率该进程 。

打印各进程的turnaround\_time和waiting\_time，并求出最后总体的Average

具体程序步骤：

1. 用户输入进程个数n和每个进程的到达时间、运行时间等信息。
2. 对于每个进程，将它们按照到达时间的先后顺序存储到一个结构体数组中，并计算总的运行时间total\_time。
3. 运行时间片轮转调度算法，其中时间片大小为TIME\_QUANTUM。程序会一直运行，直到所有进程都执行完毕。在每个时间片中，程序会依次检查每个进程，如果该进程还有剩余时间，则将该进程运行一个时间片的时间，并更新剩余时间。如果进程的剩余时间已经为0，则记录该进程的周转时间和等待时间，并累加到总的平均周转时间和平均等待时间中。
4. 运行完所有进程后，输出每个进程的详细信息和平均周转时间、平均等待时间。
5. 释放动态分配的内存空间，结束程序运行。

总体来说，通过时间片轮转调度算法来实现多进程的调度和执行，并计算每个进程的周转时间和等待时间，最后输出平均周转时间和平均等待时间。

代码

#include <stdio.h>

#include <stdlib.h>

#define TIME\_QUANTUM 4 // 时间片大小

struct Process {

int pid; // 进程ID

int arrival\_time; // 到达时间

int burst\_time; // 运行时间

int remaining\_time; // 剩余运行时间

int turnaround\_time; // 周转时间

int waiting\_time; // 等待时间

};

int main() {

int n, total\_time = 0;

float avg\_turnaround\_time = 0, avg\_waiting\_time = 0;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct Process \*processes = (struct Process\*) malloc(n \* sizeof(struct Process)); // 动态分配进程数组

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

printf("\nProcess %d\n", i + 1);

printf("Enter arrival time: ");

scanf("%d", &processes[i].arrival\_time);

printf("Enter burst time: ");

scanf("%d", &processes[i].burst\_time);

processes[i].remaining\_time = processes[i].burst\_time;

processes[i].pid = i + 1;

total\_time += processes[i].burst\_time;

}

int time = 0, time\_slice = 0;

printf("\n");

printf("Gantt Chart:\n");

while (time < total\_time) {

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

if (processes[i].remaining\_time > 0) {

if (processes[i].remaining\_time <= TIME\_QUANTUM) {

time\_slice = processes[i].remaining\_time;

} else {

time\_slice = TIME\_QUANTUM;

}

time += time\_slice;

processes[i].remaining\_time -= time\_slice;

printf("| P%d |", processes[i].pid);

if (processes[i].remaining\_time == 0) {

processes[i].turnaround\_time = time - processes[i].arrival\_time;

processes[i].waiting\_time = processes[i].turnaround\_time - processes[i].burst\_time;

avg\_turnaround\_time += processes[i].turnaround\_time;

avg\_waiting\_time += processes[i].waiting\_time;

}

}

}

}

printf("\n\nProcess Details:\n");

printf("PID\tArrival Time\tBurst Time\tTurnaround Time\tWaiting Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].arrival\_time, processes[i].burst\_time,

processes[i].turnaround\_time, processes[i].waiting\_time);

}

avg\_turnaround\_time /= n;

avg\_waiting\_time /= n;

printf("\nAverage Turnaround Time: %f\n", avg\_turnaround\_time);

printf("Average Waiting Time: %f\n", avg\_waiting\_time);

free(processes);

return 0;

}