به نام خدا
گزارش کار آزمایش ۸ آزمایشگاه درس سیستم های عامل

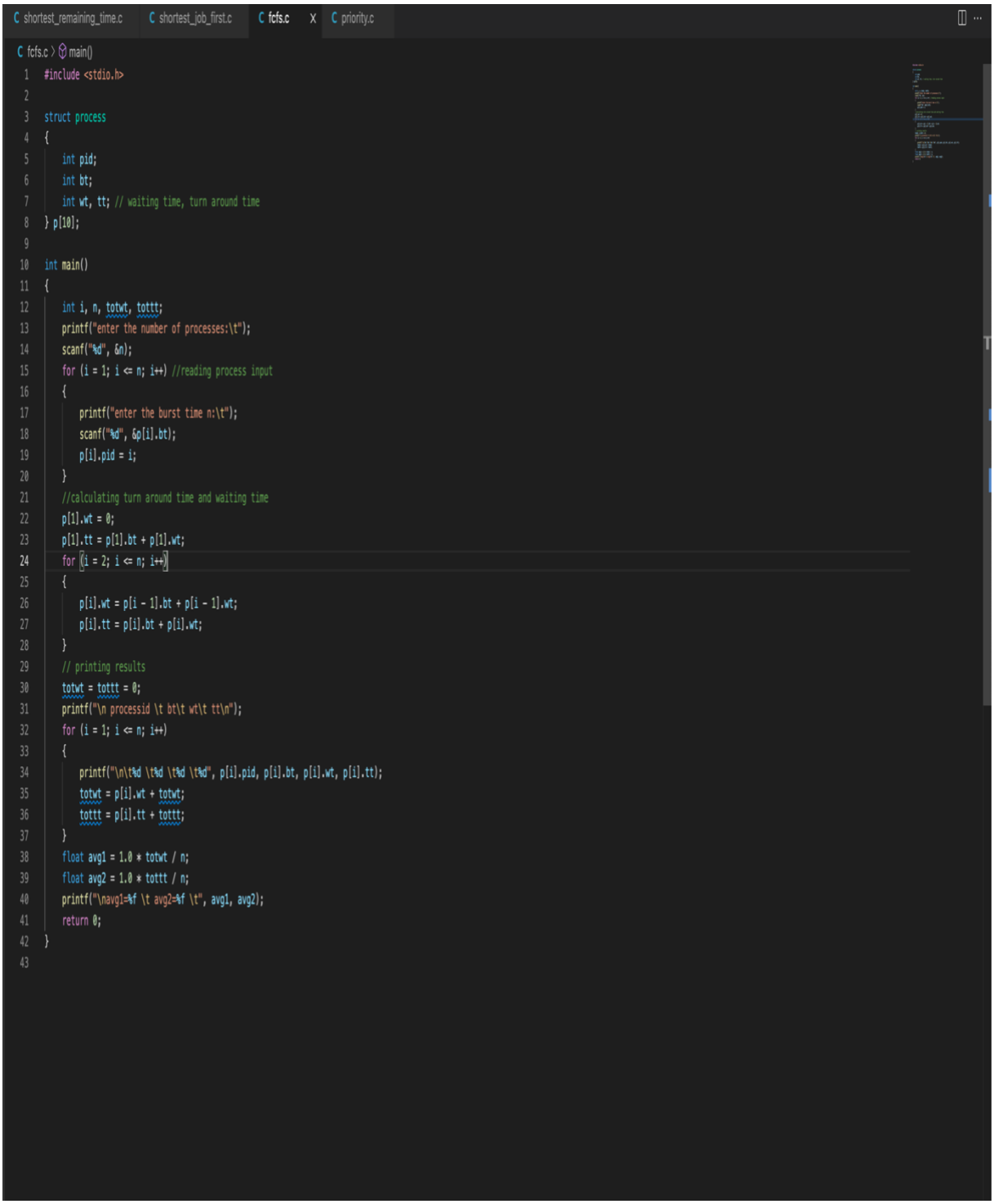
نام استاد: سرکار خانم حسینی

نام دانشجو: فرشید نوشی

شماره ی دانشجویی : ۹۸۳۱۰۶۸

قسمت اول:

الگوریتم FCFS در این قسمت پیاده شده است. در زیر تصویر کد آورده شده است.

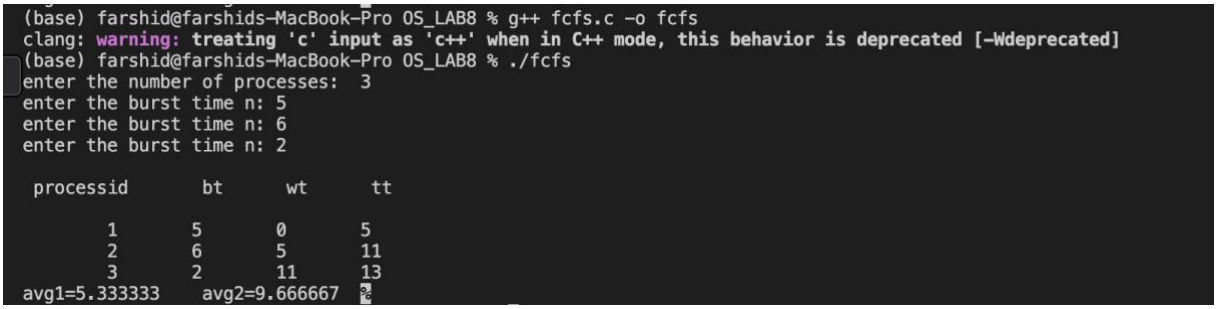


```c
#include <stdio.h>

struct process
{
    int pid;
    int bt;
    int wt, tt; // waiting time, turn around time
} p[10];

int main()
{
    int i, n, totwt, totttt;
    printf("enter the number of processes:\t");
    scanf("%d", &n);
    for (i = 1; i <= n; i++) //reading process input
    {
        printf("enter the burst time n:\t");
        scanf("%d", &p[i].bt);
        p[i].pid = i;
    }
    //calculating turn around time and waiting time
    p[1].wt = 0;
    p[1].tt = p[1].bt + p[1].wt;
    for (i = 2; i <= n; i++)
    {
        p[i].wt = p[i - 1].bt + p[i - 1].wt;
        p[i].tt = p[i].bt + p[i].wt;
    }
    // printing results
    totwt = totttt = 0;
    printf("\n processid \t bt\t wt\t tt\n");
    for (i = 1; i <= n; i++)
    {
        printf("\n\t%d \t%d \t%d \t%d", p[i].pid, p[i].bt, p[i].wt, p[i].tt);
        totwt = p[i].wt + totwt;
        totttt = p[i].tt + totttt;
    }
    float avg1 = 1.0 * totwt / n;
    float avg2 = 1.0 * totttt / n;
    printf("\navg1=%f \t avg2=%f \t", avg1, avg2);
    return 0;
}
```

با توجه به تصویر کد در ابتدا ورودی ها با پرسیدن تعداد پردازه ها و مدت زمان اجرای هر یک کار شروع میشود در ادامه چون به ترتیب داده شده پردازه ها اجرا خواهند شد زمان های turn around, waiting time را حساب میکنیم و در نهایت به صورت یک جدول خروجی را نمایش میدهیم.

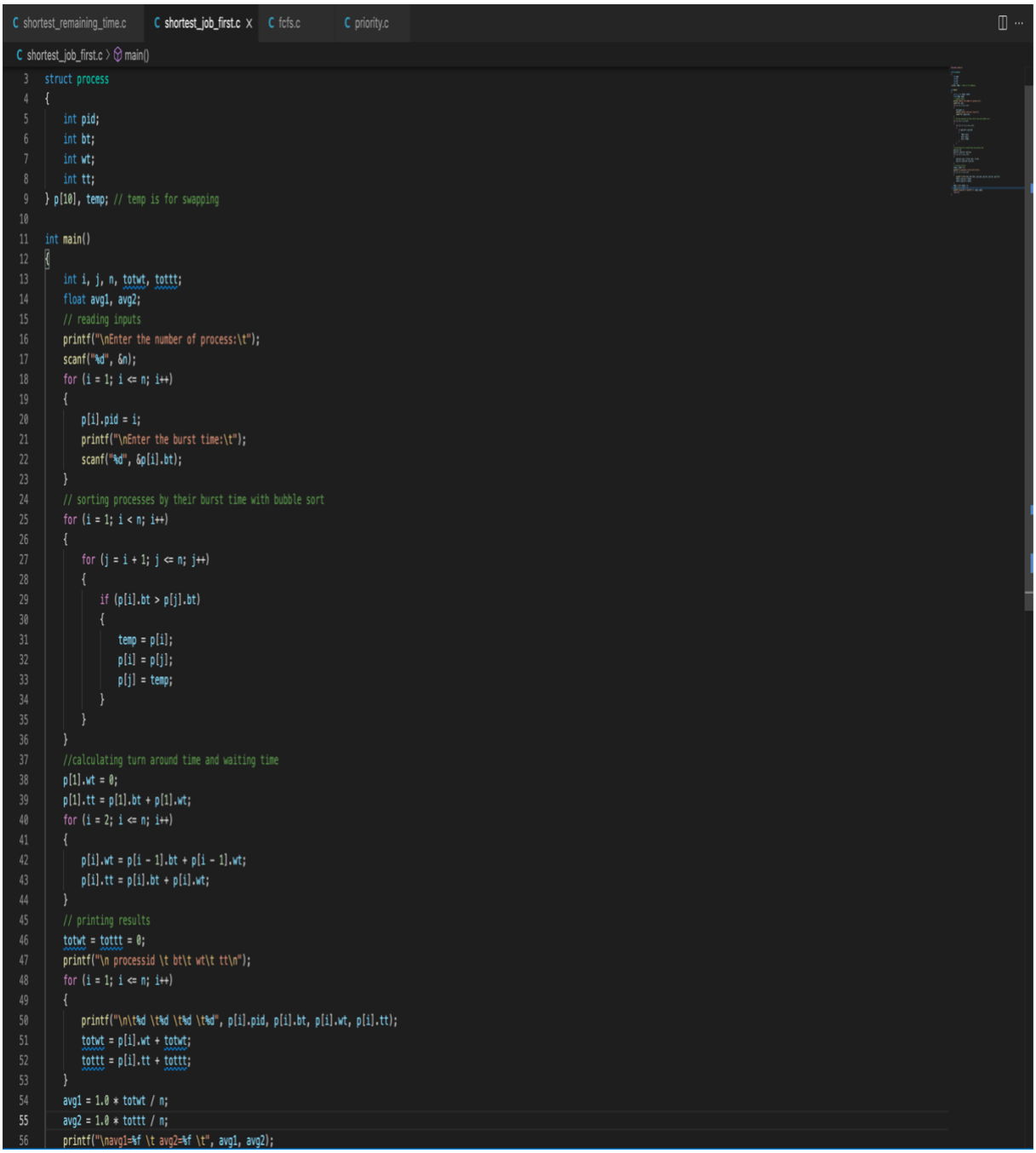
نحوه ی محاسبه ی waiting time به این صورت هست که برابر waiting time پردازه ی قبلی بعلاوه ی زمان اجرای پردازه ی قبلی هست هم چنین turn around تایم نیز برابر زمان پایان یافتن هر پردازه هست که برابر با waiting time + burst time آن پردازه میباشد.

نمونه خروجی:



قسمت دوم:

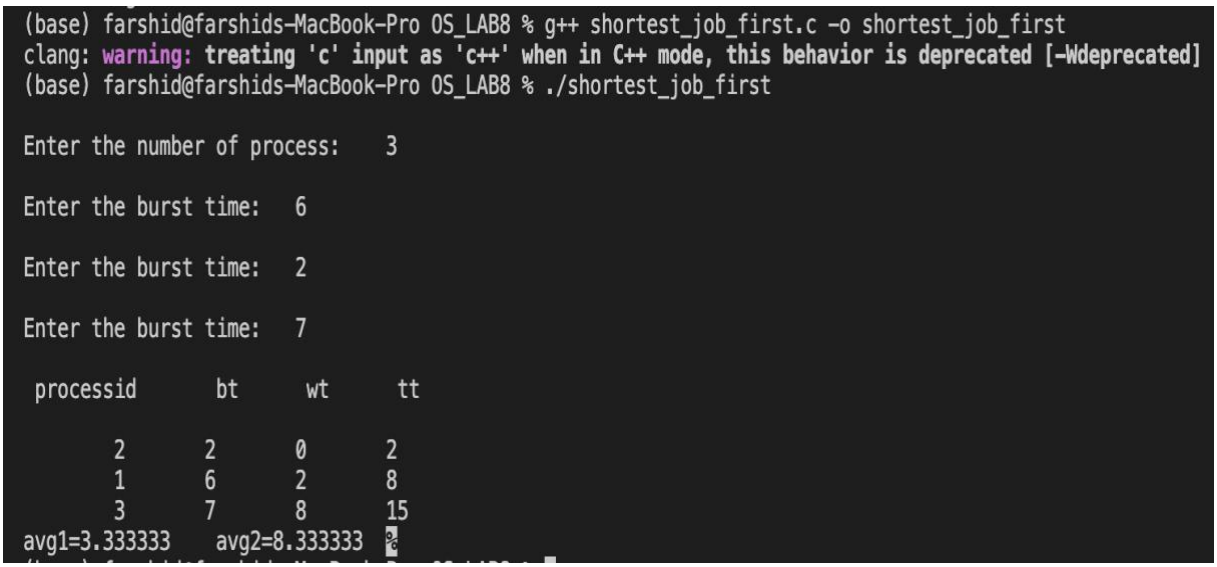
الگوریتم shortest job first در این قسمت پیاده سازی شده است. تصویر کد در زیر آورده شده است:



```c
struct process
{
    int pid;
    int bt;
    int wt;
    int tt;
} p[10], temp; // temp is for swapping

int main()
{
    int i, j, n, totwt, tottt;
    float avg1, avg2;
    // reading inputs
    printf("\nEnter the number of process:\t");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
    {
        p[i].pid = i;
        printf("\nEnter the burst time:\t");
        scanf("%d", &p[i].bt);
    }
    // sorting processes by their burst time with bubble sort
    for (i = 1; i < n; i++)
    {
        for (j = i + 1; j <= n; j++)
        {
            if (p[i].bt > p[j].bt)
            {
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }
    //calculating turn around time and waiting time
    p[1].wt = 0;
    p[1].tt = p[1].bt + p[1].wt;
    for (i = 2; i <= n; i++)
    {
        p[i].wt = p[i - 1].bt + p[i - 1].wt;
        p[i].tt = p[i].bt + p[i].wt;
    }
    // printing results
    totwt = tottt = 0;
    printf("\n processid \t bt\t wt\t tt\n");
    for (i = 1; i <= n; i++)
    {
        printf("\n\t%d \t%d \t%d \t%d", p[i].pid, p[i].bt, p[i].wt, p[i].tt);
        totwt = p[i].wt + totwt;
        tottt = p[i].tt + tottt;
    }
    avg1 = 1.0 * totwt / n;
    avg2 = 1.0 * tottt / n;
    printf("\navg1=%f \t avg2=%f \t", avg1, avg2);
```

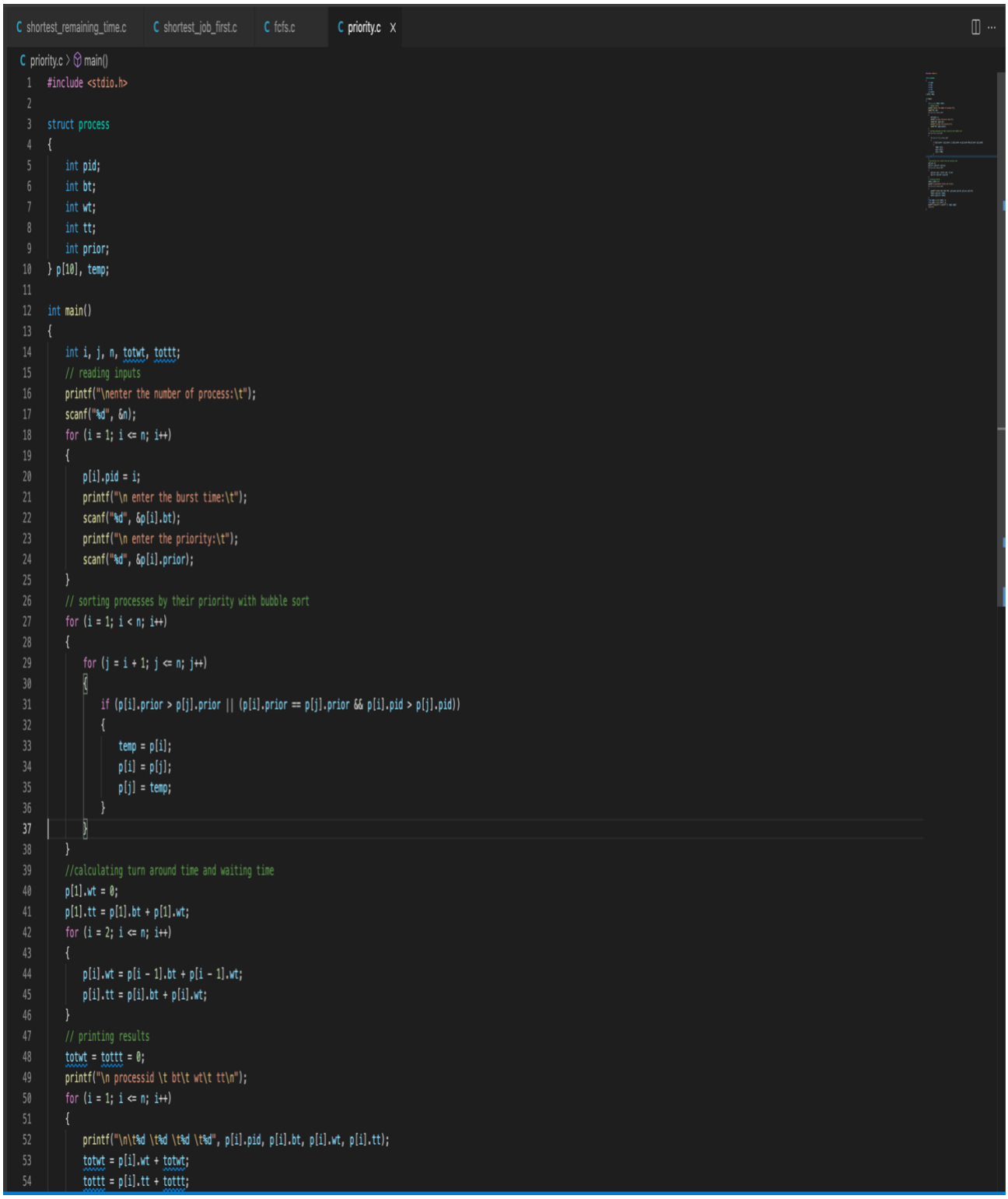
الگوریتم استفاده شده در این قسمت به این صورت هست که ابتدا پس از خواندن ورودی ها به مانند قسمت اول به روش bubble sort پردازه ها را بر حسب زمان اجرایشان مرتب میکنیم و در ادامه ترتیب نهایی اجرای آن ها نهایی خواهد شد.(در ابتدای خط ۳۷ ترتیب نهایی اجرا را داریم) در ادامه نیز به مانند قسمت اول turn around time, waiting time را حساب میکنیم و در نهایت نیز خروجی را چاپ میکنیم.

نمونه ی خروجی :

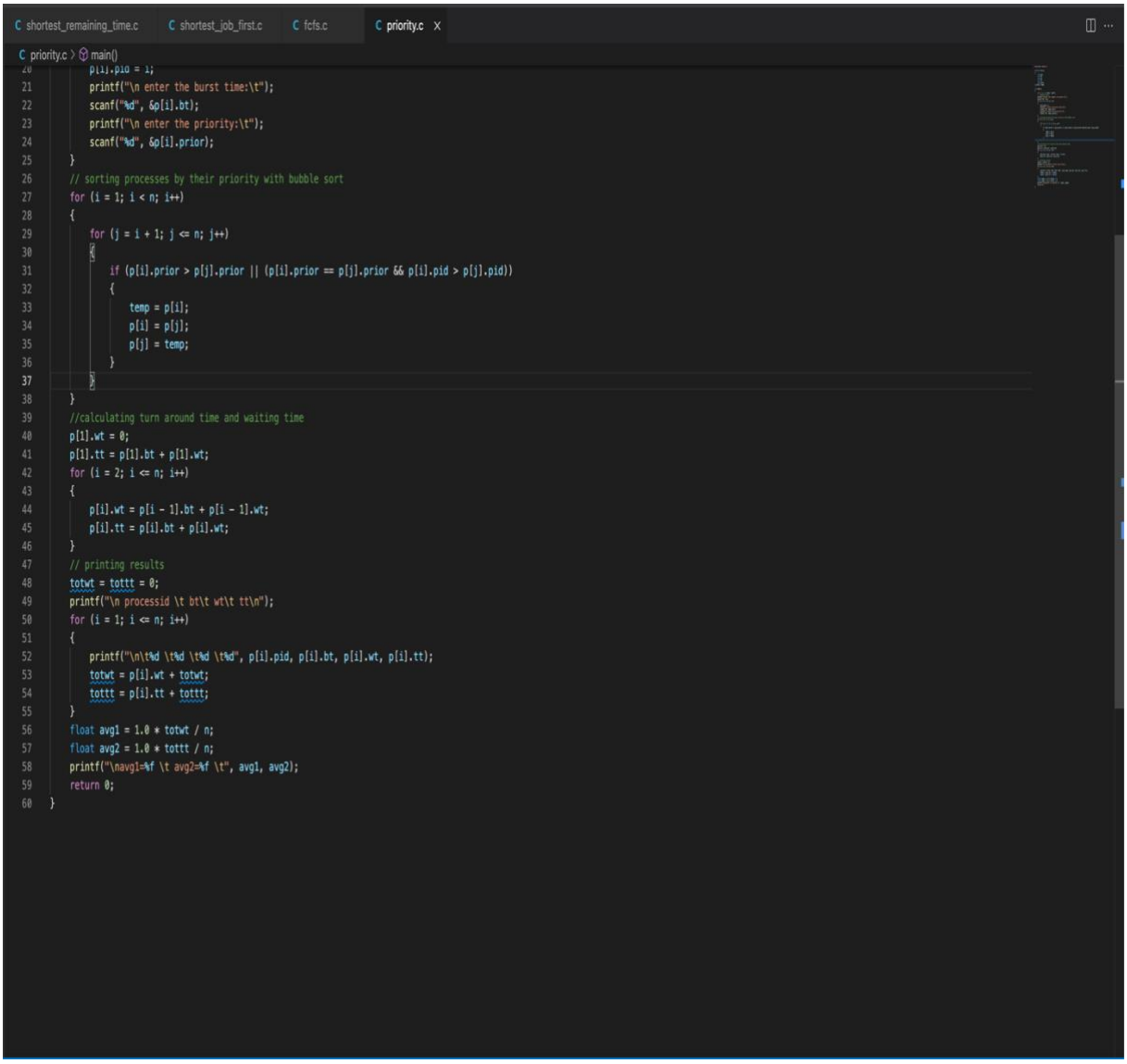


قسمت سوم:

در این قسمت الگوریتم اولویت دار(priority) پیاده سازی شده است در زیر تصاویر کد آمده اند.



```c
#include <stdio.h>

struct process
{
    int pid;
    int bt;
    int wt;
    int tt;
    int prior;
} p[10], temp;

int main()
{
    int i, j, n, totwt, tottt;
    // reading inputs
    printf("\nenter the number of process:\t");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
    {
        p[i].pid = i;
        printf("\n enter the burst time:\t");
        scanf("%d", &p[i].bt);
        printf("\n enter the priority:\t");
        scanf("%d", &p[i].prior);
    }
    // sorting processes by their priority with bubble sort
    for (i = 1; i < n; i++)
    {
        for (j = i + 1; j <= n; j++)
        {
            if (p[i].prior > p[j].prior || (p[i].prior == p[j].prior && p[i].pid > p[j].pid))
            {
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }
    //calculating turn around time and waiting time
    p[1].wt = 0;
    p[1].tt = p[1].bt + p[1].wt;
    for (i = 2; i <= n; i++)
    {
        p[i].wt = p[i - 1].bt + p[i - 1].wt;
        p[i].tt = p[i].bt + p[i].wt;
    }
    // printing results
    totwt = tottt = 0;
    printf("\n processid \t bt\t wt\t tt\n");
    for (i = 1; i <= n; i++)
    {
        printf("\n\t%d \t%d \t%d \t%d", p[i].pid, p[i].bt, p[i].wt, p[i].tt);
        totwt = p[i].wt + totwt;
        tottt = p[i].tt + tottt;
```



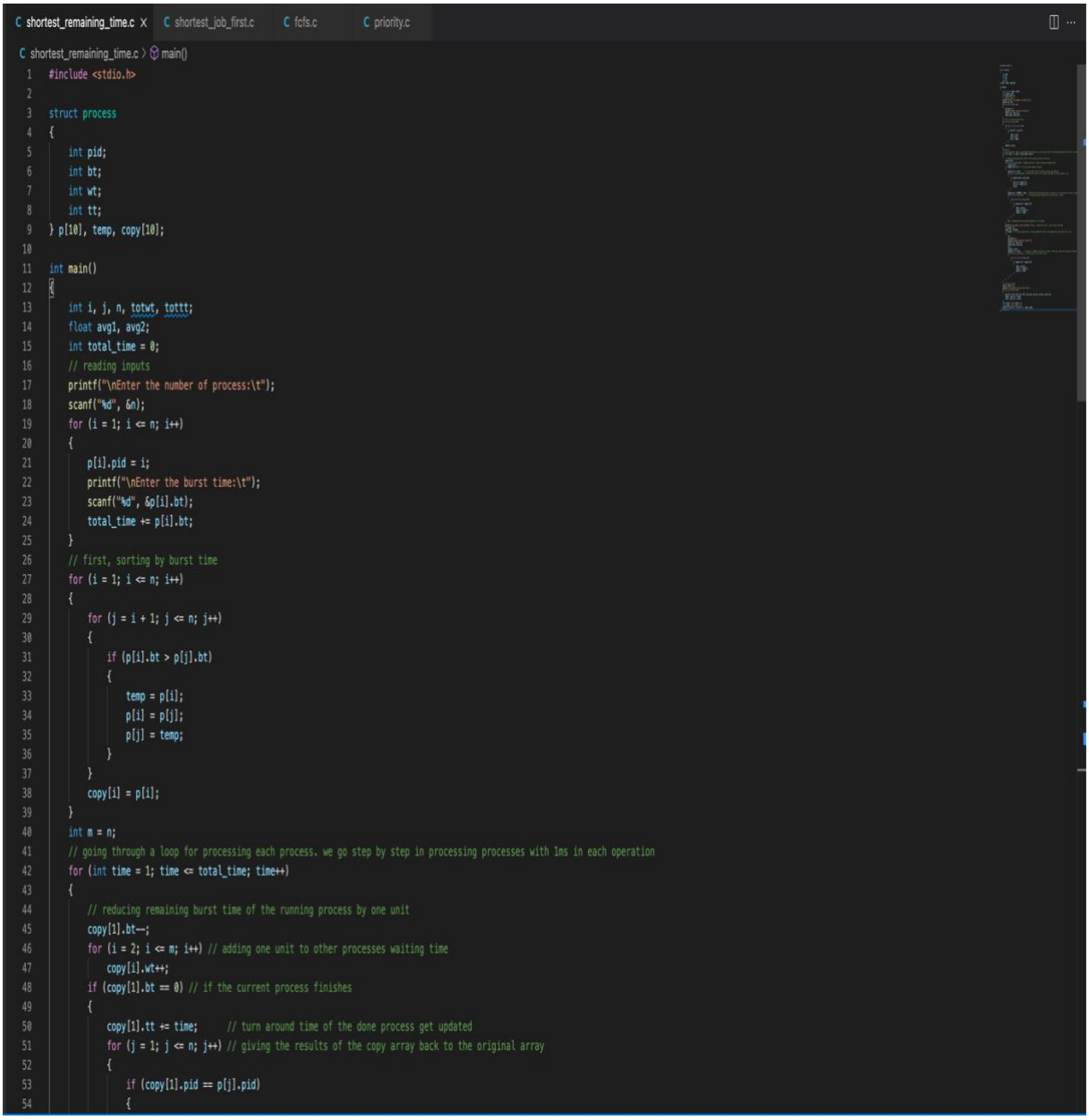
تفاوت این قسمت با قسمت قبل در قسمت مرتب سازی و گرفتن ورودی هست در این جا در ورودی متغیری برای اولویت را نیز به عنوان ورودی میگیریم و در قسمت مرتب سازی نیز ابتدا برحسب اولویت چک میکنیم که آیا دو اندیس i,j اولویت متفاوتی دارند اگر داشتند جابجا میشوند و اگر اولویت یکسان داشتند برحسب ترتیب ورودی مرتب میشوند. باقی کد به مانند قسمت دوم میباشد.(محاسبه ی خروجی ها و حساب کردن wt, tt)

نمونه خروجی:

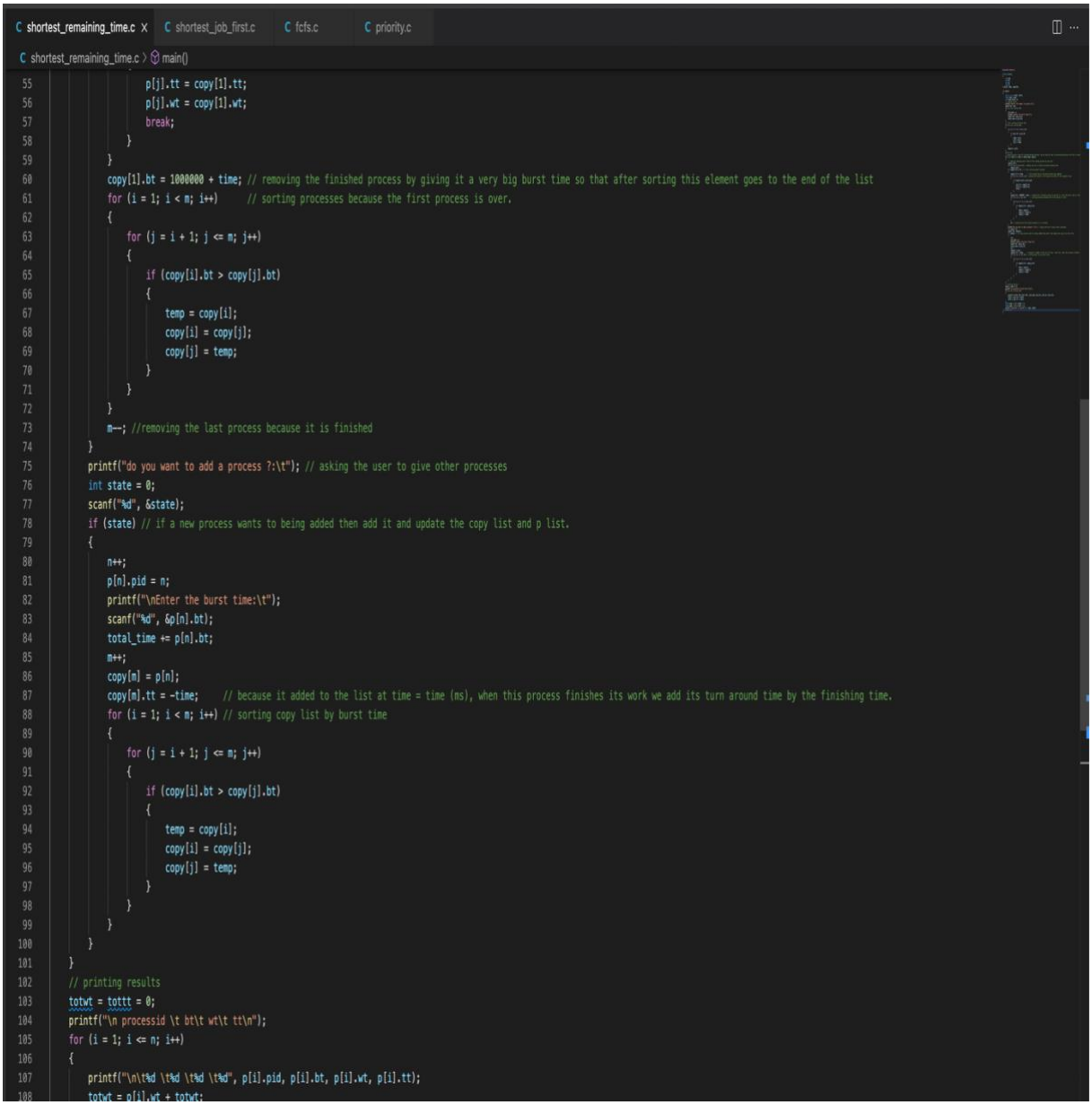
در این ورودی حالت یکسان بودن اولویت ها نیز آورده شده است.

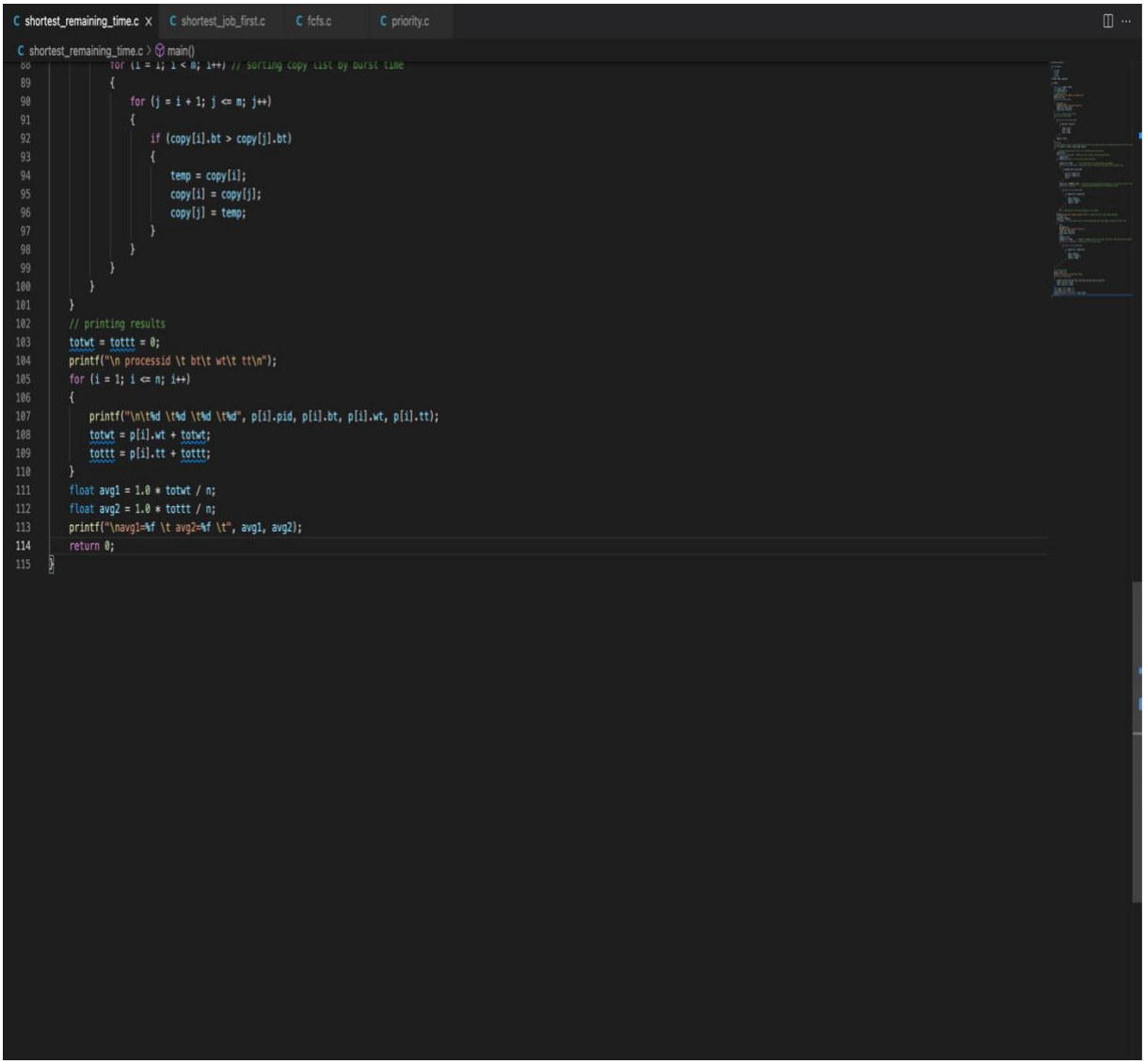
قسمت امتیازی:

در این قسمت الگوریتم SRT را پیاده سازی کرده ایم. در زیر تصاویر کد به همراه توضیح هر بخش به صورت کامنت آورده شده است.



```c
#include <stdio.h>

struct process
{
    int pid;
    int bt;
    int wt;
    int tt;
} p[10], temp, copy[10];

int main()
{
    int i, j, n, totwt, tottt;
    float avg1, avg2;
    int total_time = 0;
    // reading inputs
    printf("\nEnter the number of process:\t");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
    {
        p[i].pid = i;
        printf("\nEnter the burst time:\t");
        scanf("%d", &p[i].bt);
        total_time += p[i].bt;
    }
    // first, sorting by burst time
    for (i = 1; i <= n; i++)
    {
        for (j = i + 1; j <= n; j++)
        {
            if (p[i].bt > p[j].bt)
            {
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
        copy[i] = p[i];
    }
    int m = n;
    // going through a loop for processing each process. we go step by step in processing processes with 1ns in each operation
    for (int time = 1; time <= total_time; time++)
    {
        // reducing remaining burst time of the running process by one unit
        copy[1].bt--;
        for (i = 2; i <= m; i++) // adding one unit to other processes waiting time
            copy[i].wt++;
        if (copy[1].bt == 0) // if the current process finishes
        {
            copy[1].tt += time;      // turn around time of the done process get updated
            for (j = 1; j <= n; j++) // giving the results of the copy array back to the original array
            {
                if (copy[1].pid == p[j].pid)
                {
```


shortest_remaining_time.c
shortest_job_first.c
fcfs.c
priority.c
shortest_remaining_time.c > main()
p[j].tt = copy[1].tt;
p[j].wt = copy[1].wt;
break;
}
}
copy[1].bt = 1000000 + time; // removing the finished process by giving it a very big burst time so that after sorting this element goes to the end of the list
for (i = 1; i < m; i++) // sorting processes because the first process is over.
{
for (j = i + 1; j <= m; j++)
{
if (copy[i].bt > copy[j].bt)
{
temp = copy[i];
copy[i] = copy[j];
copy[j] = temp;
}
}
}
m--; //removing the last process because it is finished
}
printf("do you want to add a process ?:\t"); // asking the user to give other processes
int state = 0;
scanf("%d", &state);
if (state) // if a new process wants to being added then add it and update the copy list and p list.
{
n++;
p[n].pid = n;
printf("\nEnter the burst time:\t");
scanf("%d", &p[n].bt);
total_time += p[n].bt;
m++;
copy[m] = p[n];
copy[m].tt = -time; // because it added to the list at time = time (ms), when this process finishes its work we add its turn around time by the finishing time.
for (i = 1; i < m; i++) // sorting copy list by burst time
{
for (j = i + 1; j <= m; j++)
{
if (copy[i].bt > copy[j].bt)
{
temp = copy[i];
copy[i] = copy[j];
copy[j] = temp;
}
}
}
}
}
// printing results
totwt = totttt = 0;
printf("\n processid \t bt\t wt\t tt\n");
for (i = 1; i <= n; i++)
{
printf("\n\t%d \t%d \t%d \t%d", p[i].pid, p[i].bt, p[i].wt, p[i].tt);
totwt = p[i].wt + totwt;



در این قسمت کد اندکی از قسمت های دیگر پیچیده تر است، قسمت های مختلف کد سعی شده با استفاده از کامنت گذاری توضیح داده شوند. به طور کلی در این بخش یک آرایه ی کپی از پردازه ها میگیریم و پردازه ها را میکرو ثانیه به میکروثانیه اجرا میکنیم و جلو میرویم. در هر میکروثانیه از کاربر در مورد اضافه کردن یک پردازه ی جدید نیز سوال میکنیم در صورتی که پاسخ ۱ بدهد یک پردازه ورودی گرفته میشود. در غیر این صورت در ارایه ی مرتب شده ی copy بر حسب تایم باقی مانده، اولین پردازه یک میکروثانیه اجرا میشود و سپس برحسب تمام شدنش یا نشدنش تصمیم گیری میشود. در هر دو حالت waiting time و turn around time پردازه های باقی مانده در لیست بروز میشوند. هر پردازه هم که تمام میشود به روش توضیح داده شده در کد از به ته لیست برده میشود تا دیگر در پردازشمان نیاید.

نمونه خروجی:

```
(base) farshid@farshids-MacBook-Pro OS_LAB8 % g++ shortest_remaining_time.c -o shortest_remaining_time
clang: warning: treating 'c' input as 'c++' when in C++ mode, this behavior is deprecated [-Wdeprecated]
(base) farshid@farshids-MacBook-Pro OS_LAB8 % ./shortest_remaining_time

Enter the number of process:    2

Enter the burst time:   3

Enter the burst time:   2
do you want to add a process ?: 1

Enter the burst time:   3
do you want to add a process ?: 0
do you want to add a process ?: 0
do you want to add a process ?: 0
do you want to add a process ?: 0
do you want to add a process ?: 0
do you want to add a process ?: 0
do you want to add a process ?: 0

Process id      bt      wt      tt
        2       2       0       2

        1       3       2       5

        3       3       4       7

AVG1=2.000000    AVG2=4.000000
```

```
(base) farshid@farshids-MacBook-Pro OS_LAB8 % ./shortest_remaining_time

Enter the number of process:    1

Enter the burst time:   5
do you want to add a process ?: 1

Enter the burst time:   2
do you want to add a process ?: 0
do you want to add a process ?: 0
do you want to add a process ?: 0
do you want to add a process ?: 0
do you want to add a process ?: 0
do you want to add a process ?: 0

Process id      bt      wt      tt
        1       5       2       7

        2       2       0       2

AVG1=1.000000    AVG2=4.000000
```

در دو خروجی میبینیم که پردازه ها در هر ثانیه بر حسب زمان باقی مانده شان انتخاب برای اجرا میشوند.(در نمونه ی دوم این مورد واضح است جایی که ابتدا یک میکرو ثانیه پردازه ی شماره ی یک اجرا شده است در ادامه پردازه ی دو بلافاصله شروع به اجرا شده است و wt پردازه ی اول ۲ واحد زیاد شده است در ادامه پردازه ی اول آن قدر اجرا شده است تا تمام بشود tt پردازه ی دوم نیز چون از اخر اولین واحد آمده است و تا اخر سومین واحد اجرا شده است برابر ۲ شده است برای پردازه ی اول از لحظه ی ۰ وارد شده است تا اخر ۷)

قسمت چهارم:

برای توضیح کد به این صورت هست که یک ارایه ی x از پردازه های ورودی به همراه زمان اجرایشان داریم. در ادامه به اندازه ای که لازم هست تا همه ی پردازه ها اجرا شوند یعنی مجموع زمان اجرایشان در یک حلقه شروع به چرخش میکنیم(به نمایندگی از هر واحد زمانی) در داخل حلقه اولین پردازه ی باقی مانده مان را پیدا میکنیم با یک فور داخلی و در ادامه در دو حالتی که آیا این پردازه در این برهه ی m واحدی ما تمام میشود یا خیر حالت بندی کرده ایم. در هر دو حالت اطلاعات پردازه ای که الان میخواهد اجرا شود را در ارایه ی p به روز میکنیم مانند waiting time, turn around time که از روز اندیس های قبلی یا i ارایه های p, x محاسبه میشوند. در نهایت نیز به مانند بخش های قبلی خروجی را چاپ میکنیم.(میانگین waiting time, turn around time را نیز حساب میکنیم).

rr.c

```c
#include <stdio.h>

struct process
{
    int pid;
    int bt;
    int wt, tt; // waiting time, turn around time
} p[60], x[10];

int main()
{
    int i, j, k, tot = 0, m, n;
    float total_waiting_time = 0.0, total_turn_around_time = 0.0;
    printf("\nenter the number of process:\t");
    scanf("%d", &n);
    for (i = 1; i <= n; i++) // reading process input
    {
        x[i].pid = i;
        printf("\nenter the burst time:\t");
        scanf("%d", &x[i].bt);
        tot += x[i].bt;
    }
    printf("\nenter the time quantum:\t");
    scanf("%d", &m);
    // calculating turn around time and waiting time
    p[0].tt = 0;
    k = 1;
    for (j = 1; j <= tot; j++)
        for (i = 1; i <= n; i++)
            if (x[i].bt != 0)
            {
                p[k].pid = i;
                if (x[i].bt - m < 0)
                {
                    p[k].bt = x[i].bt;
                    p[k].wt = p[k - 1].tt;
                    p[k].tt = p[k].wt + p[k].bt;
                    x[i].bt = 0;
                    k++;
                }
                else
                {
                    x[i].bt = x[i].bt - m;
                    p[k].wt = p[k - 1].tt;
                    p[k].tt = p[k].wt + m;
                    k++;
                }
            }
    // printing results
    printf("\nsum of burst times(total burst time):\t%d", tot);
    printf("\nProcess id \twt \ttt");
    for (i = 1; i < k; i++)
    {
        printf("\n\t%d \t%d \t%d", p[i].pid, p[i].wt, p[i].tt);
```

rr.c

```c
    printf("\nenter the time quantum:\t");
    scanf("%d", &m);
    // calculating turn around time and waiting time
    p[0].tt = 0;
    k = 1;
    for (j = 1; j <= tot; j++)
        for (i = 1; i <= n; i++)
            if (x[i].bt != 0)
            {
                p[k].pid = i;
                if (x[i].bt - m < 0)
                {
                    p[k].bt = x[i].bt;
                    p[k].wt = p[k - 1].tt;
                    p[k].tt = p[k].wt + p[k].bt;
                    x[i].bt = 0;
                    k++;
                }
                else
                {
                    x[i].bt = x[i].bt - m;
                    p[k].wt = p[k - 1].tt;
                    p[k].tt = p[k].wt + m;
                    k++;
                }
            }
    // printing results
    printf("\nsum of burst times(total burst time):\t%d", tot);
    printf("\nProcess id \twt \ttt");
    for (i = 1; i < k; i++)
    {
        printf("\n\t%d \t%d \t%d", p[i].pid, p[i].wt, p[i].tt);
        total_waiting_time += p[i].wt;
        total_turn_around_time += p[i].tt;
    }
    total_waiting_time /= n;
    total_turn_around_time /= n;
    printf("\naverage waiting time:\t%f", total_waiting_time);
    printf("\naverage turn around time:\t%f\n", total_turn_around_time);
    return 0;
}
```

نمونه ی خروجی:

```
(base) farshid@farshids-MacBook-Pro az8_9831068 % g++ rr.c -o rr
clang: warning: treating 'c' input as 'c++' when in C++ mode, this behavior is deprecated [-Wdeprecated]
(base) farshid@farshids-MacBook-Pro az8_9831068 % ./rr

enter the number of process:    3

enter the burst time:   3

enter the burst time:   5

enter the burst time:   7

enter the time quantum: 2

sum of burst times(total burst time):   15
Process id      wt      tt
        1       0       2
        2       2       4
        3       4       6
        1       6       7
        2       7       9
        3       9       11
        2       11      12
        3       12      14
        3       14      15
average waiting time:   21.666666
average turn around time:       26.666666
(base) farshid@farshids-MacBook-Pro az8_9831068 %
```

در این نمونه سه پردازه داریم که زمان اجرای هر کدام ۳و۵و۷ واحد زمانی هست و کوانتوم زمانی ما نیز ۲ است. در آغاز پردازه ی اول و دوم و سوم هرکدام دو واحد زمانی اجرا میشوند. و زمان اجرایشان به ۱و۳و۵ کاهش میابد در ادامه پردازه ی اول یک واحد زمانی اجرا میشود و اجرایش تمام میشود. در ادامه ی آن پردازه ی دوم و سوم هر کدام دو واحد زمانی را مصرف میکنند و زمان باقی مانده ی پردازه ها به صورت ۱و ۳ واحد باقی میمانند در ادامه نیز یک واحد زمانی پردازه ی دوم مصرف میکند و سپس ابتدا یک دو واحد زمانی پردازه ی سوم مصرف میکند و در سری اخرش یک واحد زمانی مصرف میکند.

در نهایت میانگین waiting time, turn around time برای پردازه ها امده است.

قسمت پنجم:

الگوریتمFCFS :

مزایا: سادگی اجرا و حداقل بودن سربار اجرایی را دارد زیرا نیازی به اطلاعات قبلی یا اضافه در مورد پردازه‌ها ندارد و عدم وجود قحطی یا starvation

معایب: میانگین زمان انتظار برای فرآیندها بسیار بالا است و بالا بودن میانگین زمان برگشت و برای اجرا روی یک سیستم تک پردازنده‌ای، روش خوبی نیست.

کاربرد: این روش می‌تواند باعث استفاده ناکارآمد از CPU و همچنین دستگاه‌های ورودی/خروجی (I/O)شود.

الگوریتمSJF :

مزایا: کمترین زمان انتظار.

معایب: امکان گرسنگی یا starvation فرآیندهای طولانی وجود دارد و این الگوریتم برای محیطهای اشتراک زمانی، چندان مناسب نیست.

کاربرد: اگر پردازه ها به طور همزمان وارد سیستم شوند، میانگین زمان برگشت و زمان انتظار، حداقل میشود. یکی از بزرگ ترین مشکلات الگوریتم SJF نیاز به دانستن زمان پردازش هر فرآیند است. در حالت عادی و پیش از اجرای کامل یک فرآیند، زمان دقیق آن را نمی دانیم. به همین دلیل باید زمان اجرای پردازش را تخمین زد.

الگوریتم priority:

مزایا: اختیار اولویت دادن به عهده ی برنامه است.

معایب: ممکن است برنامه‌ای مخرب به خود الویت بالا بدهد.

کاربرد: در سیستم عامل‌های ما رایج نیست.

الگوریتمRR :

مزایا: تضمین زمان پاسخ مطلوب برای کارهای معمولی کوچک و نداشتن قحطی و گرسنگی در سیستم و سادگی اجرا و عملکرد عادلانه

معایب: سربار تعداد زیاد تعویض میان اجرای فرآیندها و میانگین زمان اجرای نسبتاً بالا در پردازش‌های طولانی

کاربرد: انتخاب برهه زمانی (مدت زمان کوانتوم) یکی از معیارهای مشخص‌کننده عملکرد الگوریتم RRاست. اگر کوانتوم زمانی خیلی کوچک باشد، توان عملیاتی (throughput) سیستم عامل کم خواهد شد.