دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده مهندسی برق

آزمایشگاه سیستم عامل

آزمایش ۸

شبیه ساز الگوریتم های زمان بندی

نگارش

علی بابالو

پویا شریفی

استاد راهنما

مهندس کیخا

خرداد ماه 1402

در ابتدا یک فایل بنام process.h تشـــکیل می‌دهیم که در آن اســـتراکت process و توابع مربوط به مقایســـه پردازه ها از نظر زمان ورود، الویت و زمان باقی مانده و همچنین تابع نوشتن جدول زمانی را در آن قرار می‌دهیم:

Process.h :

```c
#include <stdio.h>
#include <stdlib.h>

int n;           // number of processes
int i, j;        // used for iterations
int total_burst; // total cpu burst of processes

typedef struct process{
    int pid;
    int priority;
    int at; // arrival time
    int bt; // burst time
    int rmt; // remaining time
    int wt; // waiting time
    int tat; // turnaround time
} process;

/* compare two processes  by arrival time */
int compare_arrival(const void* process1, const void* process2){
    process* p1 = (process*) process1;
    process* p2 = (process*) process2;
    return p2->at < p1->at;
}

/* compare two processes  by priority */
int compare_priority(const void* process1, const void* process2){
    process* p1 = (process*) process1;
    process* p2 = (process*) process2;
    return p1->priority > p2->priority;
}

/* compare two processes  by remaining time */
int compare_RemainigTime(const void* process1, const void* process2){
    process* p1 = (process*) process1;
    process* p2 = (process*) process2;
    return p1->rmt > p2->rmt;
}

/* print all processes info */
void print_processes_info(process PCB[]){
```

```
    printf("\n%s%9s%10s%7s%6s%12s", "pid", "arrival", "priority", "burst",
"wait", "turnarround");
    for(i = 0; i < n; i++)
        printf("\n%d%8d%10d%8d%7d%8d\n",
        PCB[i].pid, PCB[i].at, PCB[i].priority, PCB[i].bt, PCB[i].wt,
PCB[i].tat);
    printf("\n");
}
```

بخش اول: پیاده سازی الگوریتم First Come First Serve:

در این الگوریتم به پردازه ای که زودتر وارد میشود cpu را تخصیص میدهیم. برای اینکار ابتدا از کاربر تعداد پردازه ها، مقدار burst time هر یک از پردازه ها و زمان ورود پردازه ها را می‌پرسیم. سپس پس از مقدار دهی اولیه به پردازه ها تابع FCFS() را صدا می‌کنیم که در این تابع با توجه به تابع compare_arrival() و استفاده از تابع qsort، استراکت مربوط به پردازه ها را sort می‌کنیم و با توجه به مقادیر سورت شده، cpu را به پردازه ها اختصاص می‌دهیم.

```
#include "./process.h"
#include <stdio.h>
#include <stdlib.h>

/* first come first serve policy */
void FCFS(process PCB[]){
    printf("\narrival time of processes: \n");
    for(i = 0; i < n; i++){
        int arrival;
        scanf("%d", &arrival);
        PCB[i].at = arrival;
    }

    int time = 0;

    qsort(PCB, n, sizeof(struct process), compare_arrival);

    for(i = 0; i < n; i++){
        PCB[i].wt = time;
        PCB[i].tat = PCB[i].wt + PCB[i].bt;
        PCB[i].rmt = 0;
        time += PCB[i].bt;
    }
    printf("\nFCFS:\n");
```

```
    print_processes_info(PCB);
}

int main(int argc, char const *argv[]){
    printf("number of processes: ");
    scanf("%d", &n);

    process PCB[n];
    total_burst = 0;

    printf("\nEnter Burst Time:\n");
    for(i = 0; i < n; i++){
        int burst;
        scanf("%d", &burst);
        total_burst += burst;
        PCB[i].pid = i;
        PCB[i].wt = 0;
        PCB[i].tat = 0;
        PCB[i].at = 0;
        PCB[i].priority = 0;
        PCB[i].bt = burst;
        PCB[i].rmt = burst;
    }

    FCFS(PCB);

    int total_wt = 0, total_tat = 0;
    for (int i = 0 ; i < n ; i++)
    {
        total_wt = total_wt + PCB[i].wt;
        total_tat = total_tat + PCB[i].tat;
    }

    printf("Average waiting time = %f", (float)total_wt / (float)n);
    printf("\nAverage turn around time = %f\n", (float)total_tat / (float)n);

    return 0;
}
```

نتایج مربوط به این الگوریتم را در زیر مشاهده می‌کنید:

```
cavendish@LAPTOP-J4FO4O81:/mnt/c/Users/98912/Desktop/uni/OS LAB/EXP 8$ gcc FCFS.c -o FCFS
cavendish@LAPTOP-J4FO4O81:/mnt/c/Users/98912/Desktop/uni/OS LAB/EXP 8$ ./FCFS
number of processes: 5

Enter Burst Time:
1 9 3 2 5

arrival time of processes:
0 1 2 3 4

FCFS:

pid  arrival  priority  burst  wait turnarround
0       0         0       1      0       1

1       1         0       9      1      10

2       2         0       3     10      13

3       3         0       2     13      15

4       4         0       5     15      20

Average waiting time = 7.800000
Average turn around time = 11.800000
cavendish@LAPTOP-J4FO4O81:/mnt/c/Users/98912/Desktop/uni/OS LAB/EXP 8$
```

بخش دوم: پیاده سازی الگوریتم Shortest Job First:

در این الگوریتم به پردازه ای که کمترین time burst را میخواهد اول cpu را اختصاص میدهیم)با توجه به اینکه در دستور کار گفته شده فقط زمان سرویس دهی هر فرایند را از کاربر بگیریم فرض شده است که زمان ورود هم پردازه ها صفر است.)

برای اینکار پس از گرفتن ورودی ها از کاربر و مقدار دهی اولیه تابع SJF را صدا می‌کنیم تا با توجه به مقدار RemainigTime پردازه ها (در مقدار دهی اولیه، زمان باقی مانده را برابر با Burst Time درنظر گرفتیم) را sort بکند و پس از این سورت شدن cpu را به پردازه ها اختصاص می‌دهیم.

```c
#include "./process.h"

void SJF(process PCB[]){
    int time = 0;

    qsort(PCB, n, sizeof(struct process), compare_RemainigTime);

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

```
        PCB[i].at = 0;
        PCB[i].wt = time;
        PCB[i].tat = PCB[i].wt + PCB[i].bt;
        PCB[i].rmt = 0;
        time += PCB[i].bt;
    }
    printf("\nSJF:\n");
    print_processes_info(PCB);
}

int main(int argc, char const *argv[]){
    printf("number of processes: ");
    scanf("%d", &n);

    process PCB[n];
    total_burst = 0;

    printf("\nEnter Burst Time:\n");
    for(i = 0; i < n; i++){
        int burst;
        scanf("%d", &burst);
        total_burst += burst;
        PCB[i].pid = i;
        PCB[i].wt = 0;
        PCB[i].tat = 0;
        PCB[i].at = 0;
        PCB[i].priority = 0;
        PCB[i].bt = burst;
        PCB[i].rmt = burst;
    }

    SJF(PCB);

    int total_wt = 0, total_tat = 0;
    for (int i = 0 ; i < n ; i++)
    {
        total_wt = total_wt + PCB[i].wt;
        total_tat = total_tat + PCB[i].tat;
    }

    printf("Average waiting time = %f", (float)total_wt / (float)n);
    printf("\nAverage turn around time = %f\n", (float)total_tat / (float)n);

    return 0;
}
```

نتایج مربوط به این الگوریتم را در زیر مشاهده می‌کنید:

```
cavendish@LAPTOP-J4FO4O81:/mnt/c/Users/98912/Desktop/uni/OS LAB/EXP 8$ gcc SJF.c -o SJF
cavendish@LAPTOP-J4FO4O81:/mnt/c/Users/98912/Desktop/uni/OS LAB/EXP 8$ ./SJF
number of processes: 5

Enter Burst Time:
1 9 5 3 2

SJF:

pid  arrival  priority  burst  wait turnarround
0       0         0       1      0       1

4       0         0       2      1       3

3       0         0       3      3       6

2       0         0       5      6      11

1       0         0       9     11      20

Average waiting time = 4.200000
Average turn around time = 8.200000
cavendish@LAPTOP-J4FO4O81:/mnt/c/Users/98912/Desktop/uni/OS LAB/EXP 8$
```

بخش سوم: پیاده سازی الگوریتم Priority:

در این الگوریتم برای هر پردازه یک اولویت در نظر میگیریم)پردازه با عدد priority کمتر اولویت بیشتری دارد(و هر بار به پردازه ای که بیشترین اولویت را دارد و هنوز انجام نشده است cpu را تخصیص میدهیم.

مانند ۲ الگوریتم قبل این بار نیز مقدار Burst Time, Priority و تعداد پردازه ها را از کاربر می‌گیریم و پس از مقدار دهی اولیه تابع priorityQueue را صدا می‌کنیم. در این تابع استراکت پردازه های ورودی را بر اساس مقدار priority سورت می‌کنیم و cpu را به پردازه ها اختصاص می‌دهیم.

```c
#include "./process.h"

/* priority queue policy */
void PriorityQueue(process PCB[]){
    printf("\npriority of processes: \n");
    for(i = 0; i < n; i++){
        int priority;
        scanf("%d", &priority);
        PCB[i].priority = priority;
    }
```

```
    int time = 0;

    qsort(PCB, n, sizeof(struct process), compare_priority);

    for(i = 0; i < n; i++){
        PCB[i].wt = time;
        PCB[i].tat = PCB[i].wt + PCB[i].bt;
        PCB[i].rmt = 0;
        time += PCB[i].bt;
    }
    printf("\nPriority:\n");
    print_processes_info(PCB);
}

int main(int argc, char const *argv[]){
    printf("number of processes: ");
    scanf("%d", &n);

    process PCB[n];
    total_burst = 0;

    printf("\nEnter Burst Time:\n");
    for(i = 0; i < n; i++){
        int burst;
        scanf("%d", &burst);
        total_burst += burst;
        PCB[i].pid = i;
        PCB[i].wt = 0;
        PCB[i].tat = 0;
        PCB[i].at = 0;
        PCB[i].priority = 0;
        PCB[i].bt = burst;
        PCB[i].rmt = burst;
    }

    PriorityQueue(PCB);


    int total_wt = 0, total_tat = 0;
    for (int i = 0 ; i < n ; i++)
    {
        total_wt = total_wt + PCB[i].wt;
        total_tat = total_tat + PCB[i].tat;
    }
```

```
    printf("Average waiting time = %f", (float)total_wt / (float)n);
    printf("\nAverage turn around time = %f\n", (float)total_tat / (float)n);

    return 0;
}
```

نتایج مربوط به این الگوریتم در زیر قابل مشاهده هستند:

```
cavendish@LAPTOP-J4FO4O81:/mnt/c/Users/98912/Desktop/uni/OS LAB/EXP 8$ gcc priority.c -o priority
cavendish@LAPTOP-J4FO4O81:/mnt/c/Users/98912/Desktop/uni/OS LAB/EXP 8$ ./priority
number of processes: 5

Enter Burst Time:
1 9 3 2 4

priority of processes:
1 3 3 2 4

Priority:

pid  arrival  priority  burst  wait turnarround
0       0         1       1      0       1

3       0         2       2      1       3

1       0         3       9      3      12

2       0         3       3     12      15

4       0         4       4     15      19

Average waiting time = 6.200000
Average turn around time = 10.000000
cavendish@LAPTOP-J4FO4O81:/mnt/c/Users/98912/Desktop/uni/OS LAB/EXP 8$
```

بخش چهارم: پیاده سازی الگوریتم RoundRobin:

در این الگوریتم یک تایم کوانتوم در نظر میگیریم و به هر پردازه حداکثر به اندازه کوانتوم زمانی تعیین شده cpuرا اختصاص میدهیم و سپس نوبت را از آن میگیریم و به پردازه بعدی میدهیم. ترتیب دادن نوبت پس از اتمام کوانتوم زمانی همانند الگوریتم های قبل است و زمانی که یک دور به تمامی پردازه ها نوبت برسد دوباره از ابتدای لیست پردازه های باقی مانده به همان ترتیب قبلی حداکثر به اندازه کوانتوم زمانی نوبت دهی میکنیم و این کار را تا زمانی ادامه میدهیم که همه پردازه ها تمام شوند. برای اینکار ابتدا چک می‌کنیم زمان باقی مانده هر پردازه از quantum time بیشتر است یا خیر؟ اگر بیشتر بود از burst, remaining time مقدار تایم کوانتوم را کم می‌کنیم و به waiting time مقدار کوانتوم زمانی را اضافه می‌کنیم. اگر مقدار زمان باقی مانده از

کوانتوم زمانی کمتر و از ۰ بیشتر بود کل زمان باقی مانده را به waiting اضافه و مقدار burst, remaining time را ۰ می‌کنیم. اگر هم مقدار زمان باقی مانده ۰ بود که نوبت را به بعدی می‌دهیم.

```
#include "./process.h"

/* round robin policy */
void RoundRobin(process PCB[]){
    int q = 1;
    printf("\ntime quantum: ");
    scanf("%d", &q);

    int time = 0;
    i = 0;

    while(total_burst != 0){
        if(PCB[i].rmt >= q){
            j = (i+1)%n;
            while(j != i){
                if(PCB[j].rmt != 0)
                    PCB[j].wt += q;
                j = (j+1)%n;
            }
            PCB[i].rmt -= q;
            total_burst -= q;
            i = (i+1)%n;
        }else if(PCB[i].rmt > 0 && PCB[i].rmt < q){
            j = (i+1)%n;
            while(j != i){
                if(PCB[j].rmt != 0)
                    PCB[j].wt += PCB[i].rmt;
                j = (j+1)%n;
            }
            total_burst -= PCB[i].rmt;
            PCB[i].rmt = 0;
            i = (i+1)%n;
        }else if(PCB[i].rmt == 0){
            i = (i+1)%n;
        }
    }

    for(i = 0; i < n; i++)
        PCB[i].tat = PCB[i].wt + PCB[i].bt;

    printf("\nRoundRobin:\n");
```

```
    print_processes_info(PCB);
}

int main(int argc, char const *argv[]){
    printf("number of processes: ");
    scanf("%d", &n);

    process PCB[n];
    total_burst = 0;

    printf("\nEnter Burst Time:\n");
    for(i = 0; i < n; i++){
        int burst;
        scanf("%d", &burst);
        total_burst += burst;
        PCB[i].pid = i;
        PCB[i].wt = 0;
        PCB[i].tat = 0;
        PCB[i].at = 0;
        PCB[i].priority = 0;
        PCB[i].bt = burst;
        PCB[i].rmt = burst;
    }

    RoundRobin(PCB);


    int total_wt = 0, total_tat = 0;
    for (int i = 0 ; i < n ; i++)
    {
        total_wt = total_wt + PCB[i].wt;
        total_tat = total_tat + PCB[i].tat;
    }

    printf("Average waiting time = %f", (float)total_wt / (float)n);
    printf("\nAverage turn around time = %f\n", (float)total_tat / (float)n);

    return 0;
}
```

نتایج کد بالا در زیر قابل مشاهده است:

```
cavendish@LAPTOP-J4FO4O81:/mnt/c/Users/98912/Desktop/uni/OS LAB/EXP 8$ gcc RR.c -o RR
cavendish@LAPTOP-J4FO4O81:/mnt/c/Users/98912/Desktop/uni/OS LAB/EXP 8$ ./RR
number of processes: 5

Enter Burst Time:
5 4 2 3 9

time quantum: 3

RoundRobin:

pid  arrival  priority  burst  wait turnarround
0       0         0       5     11      16

1       0         0       4     13      17

2       0         0       2      6       8

3       0         0       3      8      11

4       0         0       9     14      23

Average waiting time = 10.400000
Average turn around time = 15.000000
cavendish@LAPTOP-J4FO4O81:/mnt/c/Users/98912/Desktop/uni/OS LAB/EXP 8$
```

بخش پنجم مقایسه الگوریتم ها:

Algorithm	Time complexity	Usage
FCFS	O (n logn)	در مواقعی که cpu burst کوتاه است مفید است اما اگر طولانی باشد ممکن است باعث starvation بشود و همچنین ممکن است convey effect داشته باشد
SJF	O (n logn)	Waiting time را پایین و Response time را زیاد می‌کند که باعث می‌شود سیستم responsive نباشد
Priority	O (n logn)	اصافه کردن الویت مفید است اما ممکن است باعث starvation پردازه ها با الویت پایین تر بشود و هرگز نوبت اجرا پیدا نکنند
RoundRobin	n^2	سیستم Responsive می‌شود اما هزینه context switch بیشتری بایدپرداخت کنیم به همین خاطر باید مقدار کوانتوم زمانی دقیق باشد