دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

گزارش کار آزمایشگاه سیستم عامل – شماره ۸

شبیه سازی الگوریتم‌های زمانبندی

حسنا اویارحسینی – ۹۸۲۳۰۱۰

استاد درس: جناب آقای مهندس کیخا

نیمسال دوم سال تحصیلی ۰۱-۱۴۰۰

بخش ۱) الگوریتم (FCFS) first come first serve را پیاده‌سازی کنید.

در این الگوریتم به پردازه ای که زودتر وارد میشود cpu را تخصیص میدهیم. با فرض اینکه همه پردازه ها در زمان صفر وارد میشوند و با تاخیر اندکی دارای اولویت هستند (یعنی اولین پردازه ای که اطلاعاتش را وارد کردیم نسبت به همه زودتر وارد شده ولی همه پردازه ها در زمان صفر وارد شده اند [حالتی که زمان ورود پردازه ها متفاوت باشد در تمرین سری ۷ بخش ۲ بررسی و پیاده سازی شده]) کد زیر را پیاده سازی کرده ایم.

کد:

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

struct Process{

  int pid;
  int bt;
  int wt;
  int tt;
};
struct Process * p;

int main()
{

    int n;
    printf("Enter number of process: ");
    scanf("%d", &n);
    p = (struct Process*) malloc(n*sizeof(struct Process));
    printf("Enter Burst Time:\n");
    for(int i  = ۱; i <= n; i++){
        p[i-۱].pid = i;
        printf("P%d: ", i);
        scanf("%d",&p[i-۱].bt);
        p[i-۱].wt = ۰;
        p[i-۱].tt = ۰;
    }


    p[۰].wt = ۰;

    // calculating waiting time
    for (int i = ۱; i < n ; i++)
        p[i].wt = p[i-۱].wt + p[i-۱].bt;
```

```
    for (int i = ۰; i < n ; i++)
        p[i].tt = p[i].bt + p[i].wt;

    printf("Processes \t Burst Time \t Waiting Time \t Turn-Around Time\n");
    int total_wt = ۰, total_tat = ۰;
    for (int i = ۰ ; i < n ; i++)
    {
        total_wt = total_wt + p[i].wt;
        total_tat = total_tat + p[i].tt;
        printf("%d\t\t\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].bt, p[i].wt, p[i].tt);
    }

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

    return ۰;
}
```

توضیح کد:

در این کد هر پردازه به صورت یک struct Process می باشد که دارای متغیر های زیر است:

نام متغیر	توضیح
int pid;	شناسه پردازه
int bt;	Burst time
int wt;	Waiting time
int tt;	Turn around time

ابتدا از کاربر میخواهیم که تعداد پردازها ها و burst time مربوط به هر پردازه را وارد کند و خودمان مقدار اولیه wt, tt را صفر قرار میدهیم. سپس wt پردازه اول را صفر میگذاریم که یعنی اولین پردازه وارد شده اولین پردازه ای است که اجرا میشود. در مرحله بعد wt و tt بقیه پردازه ها را حساب میکنیم. با توجه به الگوریتم FCFS زمان انتظار هر پردازه برابر است با زمان انتظار پردازه قبلی به علاوه مقدار زمانی که برای اجرای پردازه قبلی صرف شده و زمان اجرای هر پردازه نیز برابر است با مقدار زمانی که پردازه در حال اجرا بوده و مقدار زمانی که صبر کرده. در نهایت اطلاعات هر پردازه را به ترتیب اجرا چاپ میکنیم و میانگین زمان اجرا و انتظار را نیز محاسبه و نمایش میدهیم.

خروجی کد:

*در تمامی بخش ها مثالی یکسان بررسی شده تا در نهایت الگوریتم ها را به کمک این مثال با هم مقایسه کنیم.

پردازه	Burst time	Priority (فقط برای بخش۳)
P۱	۱۰	۳
P۲	۱	۱
P۳	۲	۴
P۴	۱	۵
P۵	۵	۲

```
lab@ubuntu:~/Desktop/HW8$ gcc FCFS.c -o FCFS
lab@ubuntu:~/Desktop/HW8$ ./FCFS
Enter number of process: 5
Enter Burst Time:
P1: 10
P2: 1
P3: 2
P4: 1
P5: 5
Processes        Burst Time      Waiting Time    Turn-Around Time
1                      10              0               10
2                      1               10              11
3                      2               11              13
4                      1               13              14
5                      5               14              19
Average waiting time = 9.600000
Average turn around time = 13.400000
lab@ubuntu:~/Desktop/HW8$
```

بخش ۲) الگوریتم shortest job first(SJF) را پیاده‌سازی کنید.

در این الگوریتم به پردازه ای که کمترین burst time 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).

کد:

ابتدا پردازه ها را براساس مدت زمان سرویس دهی آنها مرتب میکنیم (از کوچک به بزرگ) و سپس همانند FCFS با ترتیب جدید که برای پرداز ها تعیین کردیم زمان انتظار و اجرای آنها را تعیین میکنیم.

```c
struct Process{

  int pid;
  int bt;
  int wt;
  int tt;
};
struct Process * p;
void swap(int i , int j);

int main()
```

```c
{

    int n;
    printf("Enter number of process: ");
    scanf("%d", &n);
    p = (struct Process*) malloc(n*sizeof(struct Process));

    printf("Enter Burst Time:\n");
    // User Input Burst Time and alloting Process Id.
    for (int i = ۰; i < n; i++) {
        printf("P%d: ", i + ۱);
        scanf("%d", &p[i].bt);
        p[i].pid = i + ۱;
    }

    // Sorting process according to their Burst Time.
    for (int i = ۰; i < n; i++) {
        for (int j = i + ۱; j < n; j++)
            if (p[j].bt < p[i].bt)
                swap(i, j);
    }

    p[۰].wt = ۰;

    // calculating waiting time
    for (int i = ۱; i < n ; i++)
        p[i].wt = p[i-۱].wt + p[i-۱].bt;


    for (int i = ۰; i < n ; i++)
        p[i].tt = p[i].bt + p[i].wt;

    printf("Processes \t Burst Time \t Waiting Time \t Turn-Around Time\n");
    int total_wt = ۰, total_tat = ۰;
    for (int i = ۰ ; i < n ; i++)
    {
        total_wt = total_wt + p[i].wt;
        total_tat = total_tat + p[i].tt;
        printf("%d\t\t\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].bt, p[i].wt, p[i].tt);
    }

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

مرتب سازی بر اساس burst time

بخشی که مشابه آن در FCFS تکرار شده بود

```c
void swap(int i, int j){

    int temp_id = p[i].pid;
    int temp_bt = p[i].bt;
    p[i].pid = p[j].pid;
    p[i].bt = p[j].bt;
    p[j].pid = temp_id;
    p[j].bt = temp_bt;
}
```

خروجی کد:

```
lab@ubuntu:~/Desktop/HW8$ gcc SJF.c -o SJF
lab@ubuntu:~/Desktop/HW8$ ./SJF
Enter number of process: 5
Enter Burst Time:
P1: 10
P2: 1
P3: 2
P4: 1
P5: 5
Processes        Burst Time      Waiting Time    Turn-Around Time
2                       1               0               1
4                       1               1               2
3                       2               2               4
5                       5               4               9
1                       10              9               19
Average waiting time = 3.200000
Average turn around time = 7.000000
lab@ubuntu:~/Desktop/HW8$
```

بخش۳) الگوریتم priority را پیاده‌سازی کنید.

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.

کد:

ابتدا یک فیلد priority به استراکت Process اضافه میکنیم سپس همانند الگوریتم SJF ابتدا پردازه ها را مرتب میکنیم (این بار بر اساس عدد اولویت از کوچک به بزرگ و اگر دو پردازه اولویت یکسانی داشتند پردازه با bt بیشتر را اولویت داده ایم) و سپس همانند FCFS با ترتیب جدید که برای پرداز ها تعیین کردیم زمان انتظار و اجرای آنها را تعیین میکنیم.

```c
struct Process{
  int pid;
  int bt;
  int wt;
  int tt;
  int priority;
};
struct Process * p;
```

```c
void swap(int i , int j);

int main() {
  int n;

  printf("Enter Total number of Processes:");
  scanf("%d",&n);
  p = (struct Process*) malloc(n*sizeof(struct Process));


  for(int i  = ۱; i <= n; i++){
    p[i-۱].pid = i;
    printf("P%d:\n", i);
    printf("Burst Time: ");
    scanf("%d",&p[i-۱].bt);
    printf("Priority: ");
    scanf("%d",&p[i-۱].priority);
    p[i-۱].wt = ۰;
    p[i-۱].tt = ۰;
  }

  for (int i = ۰; i < n; ++i)
    for (int j = i; j < n; ++j){

      if (p[i].priority > p[j].priority)
          swap(i,j);
      else if (p[i].priority == p[j].priority && p[i].bt < p[j].bt)
          swap(i,j);
    }

  p[۰].wt = ۰;
  p[۰].tt = p[۰].bt;

  // calculating waiting time
  for (int i = ۱; i < n ; i++)
    p[i].wt = p[i-۱].wt + p[i-۱].bt;

  for (int i = ۰; i < n ; i++)
    p[i].tt = p[i].bt + p[i].wt;

  printf("Processes \t Burst Time \t Waiting Time \t Turn-Around Time\n");
  int total_wt = ۰, total_tat = ۰;
  for (int i = ۰ ; i < n ; i++)
  {
    total_wt = total_wt + p[i].wt;
```

مرتب سازی بر اساس اولویت

بخشی که مشابه آن در FCFS تکرار شده بود

```c
    total_tat = total_tat + p[i].tt;
    printf("%d\t\t\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].bt, p[i].wt, p[i].tt);
  }

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

  return 0;
}

void swap(int i, int j){

    int temp_id = p[i].pid;
    int temp_bt = p[i].bt;
    int temp_priority = p[i].priority;
    p[i].pid = p[j].pid;
    p[i].bt = p[j].bt;
    p[i].priority = p[j].priority;
    p[j].pid = temp_id;
    p[j].bt = temp_bt;
    p[j].priority = temp_priority;
}
```

```
lab@ubuntu:~/Desktop/HW8$ gcc Priority.c -o Priority
lab@ubuntu:~/Desktop/HW8$ ./Priority
Enter Total number of Processes:5
P1:
Burst Time: 10
Priority: 3
P2:
Burst Time: 1
Priority: 1
P3:
Burst Time: 2
Priority: 4
P4:
Burst Time: 1
Priority: 5
P5:
Burst Time: 5
Priority: 2
Processes        Burst Time      Waiting Time    Turn-Around Time
2                     1               0               1
5                     5               1               6
1                     10              6               16
3                     2               16              18
4                     1               18              19
Average waiting time = 8.200000
Average turn around time = 12.000000
lab@ubuntu:~/Desktop/HW8$
```

بخش ۴) الگوریتم Round Robin را پیاده‌سازی کنید.

در این الگوریتم یک تایم کوانتوم در نظر میگیریم و به هر پردازه حداکثر به اندازه کوانتوم زمانی تعیین شده cpu را اختصاص میدهیم و سپس نوبت را از آن میگیریم و به پردازه بعدی میدهیم. ترتیب دادن نوبت پس از اتمام کوانتوم زمانی همانند FCFS 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.

کد:

ابتدا به استراکت Process دو متغیر at , nt را اضافه میکنیم که یکی زمان ورود را نشان میدهد(که در تمرین گفته شده لازم نیست زیرا فرض کرده ایم پردازه ها همه در زمن صفر وارد سیستم میشوند) و دیگری نشان دهنده مدت زمانی از busrt time است که هنوز پردازه دریافت نکرده است (مقدار این متغیر در ابتدا همان busrt time است).

دو متغیر جدید هم در کد داریم که به شرح زیر می باشند:

نام متغیر	توضیح
remaining_processes	تعداد پردازه هایی که هنوز کارشان تمام نشده و نیازمند CPU هستند
time	زمان فعلی که در آن هستیم

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

struct Process{

  int pid;
  int bt;
  int wt;
  int tt;
  int at; //arrival time
  int nt; //needed time
};

struct Process * p;
int main()
{
```

```c
int n = ٠; //total number of process
p = (struct Process*) malloc(n*sizeof(struct Process));
int i =٠;
int flag = ٠;
int time = ٠;
int remaining_processes = ٠;
int quantum = ٠;
int total_wt = ٠;
int total_tat = ٠;


printf("Enter number of Processes:");
scanf("%d",&n);
p = (struct Process*) malloc(n*sizeof(struct Process));

remaining_processes = n;

for(i = ٠;i < n;i++)
{
  printf("P%d: ",i + ١);
  p[i].pid = i + ١;
  p[i].at = ٠;
  scanf("%d",&p[i].bt);
  p[i].nt = p[i].bt;
}

printf("Enter the Time Quantum(q):");
scanf("%d",&quantum);
printf("\nProcess \t Turnaround Time \t Waiting Time\n\n");

time = ٠;
i = ٠;
while(remaining_processes > ٠)
{
  if(p[i].nt <= quantum && p[i].nt > ٠)
  {
    time += p[i].nt;
    p[i].nt = ٠;
    flag = ١;
  }
  else if(p[i].nt > ٠)
  {
    p[i].nt -= quantum;
    time += quantum;
    printf("P%d\n", i + ١);
```

```
    }
    if(p[i].nt == ۰ && flag == ۱)
    {
      remaining_processes--;
      p[i].tt = time - p[i].at;
      p[i].wt = time - p[i].at - p[i].bt;
      total_wt += p[i].wt;
      total_tat += p[i].tt;

      printf("P%d\t\t\t%d\t\t%d\n", i + ۱, p[i].tt, p[i].wt);
      flag=۰;
    }

    if(i == n - ۱)
      i = ۰;

    else if(p[i+۱].at <= time)
      i ++;

    else
      i = ۰;
  }
  printf("Average waiting time = %f", (float)total_wt / (float)n);
  printf("\nAverage turn around time = %f\n", (float)total_tat / (float)n);
  return ۰;
}
```

پس از دریافت اطلاعات از کاربر در یک حلقه از ابتدا شروع میکنیم و به هر پردازه حداکثر به اندازه کوانتوم زمانی CPU اختصاص میدهیم، یعنی برای پردازه ای که نوبتش هست بررسی میکنیم اگر کل زمانی که پردازه میخواهد از کوانتوم کمتر یا مساوی است (در کمتر یا دقیقا یک کوانتوم کار پردازه کاملا تمام میشود) دقیقا به اندازه زمانی که نیاز دارد به آن CPU اختصاص میدهیم و کار پردازه را تمام میکنیم، اما اگر زمانی که نیاز دارد بیشتر از یک کوانتوم است دقیقا به اندازه یک کوانتوم به آن پردازه CPU اختصاص میدهیم و از nt آن به اندازه یک کوانتوم کم میکنیم. حال چک میکنیم اگر کار پردازه ای که نوبتش بود در این نوبت به طور کلی تمام شده (nt == ۰ شده که این موضوع را با Flag بررسی میکنیم) waiting time, turn around time این پردازه را محاسبه و چاپ میکنیم و این پردازه را از پردازه هایی که باقی مانده است کم میکنیم، که زمان اجرا(کل مدت زمان حضور پردازه در سیستم) در واقع زمانی است که پردازه تمام شده (زمان فعلی) منهای زمانی که پردازه وارد شده بود(at) و زمان انتظار در واقع کل مدت زمانی است که پردازه در سیستم بوده ولی در حال اجرا نبوده.

پس از اینکه بررسی نوبت پردازه فعلی تمام شد نوبت را از آن میگیریم و به بعدی میدهیم (اگر آخرین پردازه باشد باید دوباره از سر لیست شروع کنیم و در غیر صورت کافیست به سراغ پردازه بعدی در لیست که زمان ورود آن گذشته است برویم).

خروجی کد:

```
lab@ubuntu:~/Desktop/HW8$ gcc RR.c -o RR
lab@ubuntu:~/Desktop/HW8$ ./RR
Enter number of Processes:5
P1: 10
P2: 1
P3: 2
P4: 1
P5: 5
Enter the Time Quantum(q):2

Process          Turnaround Time         Waiting Time

P1
P2                     3                2
P3                     5                3
P4                     6                5
P5
P1
P5
P1
P5                     15               10
P1
P1                     19               9
Average waiting time = 5.800000
Average turn around time = 9.600000
lab@ubuntu:~/Desktop/HW8$
```

بخش پنجم) مقایسه الگوریتم ها و بررسی کاربرد هر یک

نتایج حاصل از بخش های ۱ تا ۴:

الگوریتم	Waiting time	Turn around time
FCFS	۹,۶	۱۳,۴
SJF	۳,۲	۷
Priority	۸,۲	۱۲
RR	۵,۸	۹,۶

- مشکل الگوریتم FCFS پدیده کاروان است به این معنی که ممکن است یک پردازه های با CPU burst بالا ابتدا وارد شود و در نتیجه پردازهایی با CPU burst پایین تر بعد از آن بیایند و پشت آن گیر کنند که این موضوع باعث میشود waiting time به طور متوسط بالا رود (همان طور که در جدول نتایج میبینیم) و مشکلی دیگر این است که در هر لحظه فقط به یک پردازه میتوانیم سرویس دهیم و در تضاد با اجرای موازی و پایپ لاین است. اما مزیت این الگوریتم آن است که پیاده سازی آن ساده است. این الگوریتم در مواردی که تعداد پردازها کم است و یا پردازه های طولانی تر دیرتر وارد میشوند مناسب است.

- روش SJF کمترین average waiting time را به ما میدهد (همان طور که در جدول نتایج میبینیم) اما اشکال این الگوریتم آن است که اگر تعداد پردازه ها با CPU burst پایین خیلی زیاد شود نوبت به پردازه هایی با CPU burst بالا نمیرسد و هیچ وقت CPU به آنها تخصیص نمیابد. پس این الگوریتم زمانی که کمترین waiting time را میخواهیم و مطمنیم که تعداد پردازه ها با CPU busrt پایین خیلی زیاد نمیشود مناسب است.
- الگوریتم Priority برخلاف تمام الگوریتم های قبلی که اولویتی در نظر نمیگرفتند میتواند برای پردازه ها اولویت در نظر بگیرد و برای مثال اگر یک وقفه غیر قابل چشم پوشی داشته باشیم اولویت آن را بالاتر از تمام پردازه ها قرار دهیم و سریعا آن را اجرا کنیم. اما مشکل این الگوریتم starvation یا قحطی می باشد به این معنی که ممکن است یک پردازه با اولویت پایین پشت پردازه ها با اولویت بالا برای مدت خیلی طولانی گیر کند. پس این الگوریتم در مسئله هایی که نیازمند اولویت بندی هستند و مطمنیم که تعداد پردازه ها با اولویت بالا خیلی زیاد نمیشود مفید و قابل استفاده است.
- الگوریتم RR زمان اجرا به طور متوسط از SJF بیشتر است (همان طور که در جدول نتایج میبینیم) اما response time بهتر است زیر هر پردازه حداکثر در n*quantum ثانیه یکبار میتواند CPU را در اختیار بگیرد و در نتیجه یک روش عادلانه است. همچنین در این الگوریتم میتوانیم پردازش موازی و چندتایی نیز داشته باشیم. همچنین در این الگوریتم دچار پدیده کاروان یا starvation نمیشویم. اما مشکل این الگوریتم این است که اولویت پردازه ها را در نظر نمیگیرد و مثلا نمیتواند از Non makeable interruptها پشتیبانی کند. پس این الگوریتم برای زمانی که بهترین response time را میخواهیم و به دنبال روش عادلانه ای هستیم اما اولویت بندی برایمان مهم نیست مفید است.

الگوریتم	مزایا	معایب
FCFS	۱. پیاده سازی ساده	۱. پدیده کاروان Average waiting time بالا ۲. در تضاد با اجرای موازی و پایپ لاین ۳. اولویت را در نظر نمیگیریم
SJF	۱. کمترین average waiting time ۲. پیاده سازی ساده	۱. starvation ۲. اولویت را در نظر نمیگیریم
Priority	۱. در نظر گرفتن اولویت	۱. starvation
RR	۱. response time بهتر ۲. روش عادلانه ۳. پردازش موازی ۴. پدیده کاروان یا starvation نداریم	۱. اولویت را در نظر نمیگیریم ۲. زمان لازم برای context switch