ใบงานที่ 6

เรื่อง CPU scheduling

เสนอ

อาจารย์ปิยพล ยืนยงสถาวร

จัดทำโดย

นายอธิศ สุนทโรดม

รหัส 65543206086-2

ใบงานนี้เป็นส่วนหนึ่งของวิชาระบบปฏิบัติการ (ENGCE125)

หลักสูตรวิศกรรมมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์

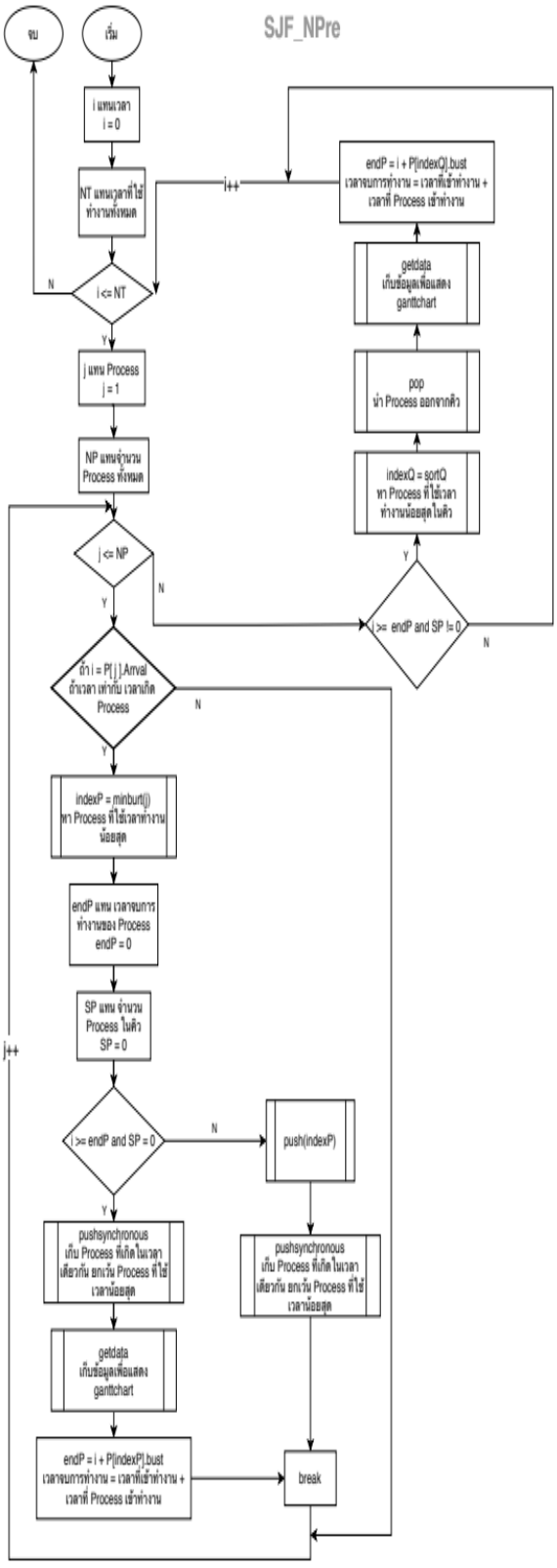
มหาวิทยาลัยเทคโนโลยีราชมงคลล้านนา เชียงใหม่

ประจำภาคเรียนที่ 2 ปีการศึกษา 2566

1. Non preemptive SJF scheduling.

Code

```c
#include <stdio.h>
#define NP 5 // number Process

typedef struct{
    int indexP;
    int startP;
}Ganttchart;
typedef struct{
    int indexP;
    int BurtT;
}Queue;
typedef struct{
    int BurtT;
    int ArrivalT;
    int Priority;
}Process;
    //Process  burt time , Arrival time , Priority

Process P[NP+1]  =  {{0},
        /*P1*/ {    9     ,       1       ,       3},     // P1 = P[1]
        /*P2*/ {    3     ,       1       ,       5},     // P2 = P[2]
        /*P3*/ {    5     ,       3       ,       1},     // P3 = P[3]
        /*P4*/ {    4     ,       4       ,       4},     // P4 = P[4]
        /*P5*/ {    2     ,       7       ,       2}};    // P5 = P[5]

Ganttchart Gantt[30];
Queue  Q[30];
int NT = 0;     //Time
int NG = 0;     //number Ganttchart
int SP = 0;     //ตัวชี้ตำแหน่งใน Q

void push(int indexP){          //เคลื่อนโปรเซสไว้ใน Q
    SP++;
    Q[SP].indexP = indexP;
    Q[SP].BurtT  = P[indexP].BurtT;
}
void pop(){                     //นำโปรเซสออกจาก Q
    if(SP == 0)
        printf("UNDER FLOW!!!\n");
    SP--;
}

Queue sortQ(){                  //เรียงข้อมูลจากน้อยไปมาก
    Queue tempP;
    int i, j;
    for (i = 1; i <SP ; ++i) {
        j = i+1;
        if(Q[i].BurtT <= Q[j].BurtT){
            tempP = Q[i];
            Q[i]  = Q[j];
            Q[j]  = tempP;
        }
    }
    return Q[SP];
}

int minBurt(int indexP, int tempP[], int *countP) {  //ในกรณีที่มีโปรเซส Burt
    int i;
    int minBurt = P[indexP].BurtT;
    int minPro = indexP;
    *countP = 0;
    for (i = 1; i <= NP; ++i)
        if (P[indexP].ArrivalT == P[i].ArrivalT) {
            tempP[*countP] = i;                         //เก็บโปรเซสที่มี Burt ใน Q
            *countP+=1;                                 //นับจำนวนโปรเซสที่มี Burt
            if(P[i].BurtT < minBurt ){
                minBurt = P[i].BurtT;
                minPro = i;
            }
        }
    return minPro;                                      //โปรเซสที่มีเวลาน้อยสุด
}

void pushSynchronous(int indexP, int tempP[], int countP) {//เคลื่อนโปรเซสที่มี Burt
    int i;
    for (i = 0; i < countP ; ++i)
        if(tempP[i] != indexP )                         //ยกเว้นโปรเซสที่เรียง
            push(tempP[i]);

}
void getData(int indexP, int time_i){
    Gantt[NG].indexP = indexP;          //เก็บโปรเซสที่เรียงสุดท้าย Ganttchart
    Gantt[NG].startP = time_i;          //เก็บเวลาเริ่มต้นของโปรเซสที่เรียงสุดท้าย Gantt
    NG++;
}

void SJF_NP(){
    int i, j;
    int indexP = 0, ENDPro = 0;
    int tempP[NP], countP = 0;
    for (i = 0; i <= NT; ++i) {             //i แทนเวลา (Time)
        for (j = 1; j <= NP; ++j) {
            if(i == P[j].ArrivalT){             //ถ้าเวลาที่ i เท่ากับโปรเซสที่ 1 2 3 ..
                indexP = minBurt(j,tempP,&countP);//เก็บโปรเซสที่มี Burt ใน Q เวลาเดียวกัน
                if(i >= ENDPro && SP == 0){     //ถ้าเวลาที่ i มากกว่าโปรเซสที่ทำงานเสร็จ
                    pushSynchronous(indexP,tempP,countP);//เคลื่อนโปรเซสที่มี Burt ใน Q
                    ENDPro = i + P[indexP].BurtT;
                    getData(indexP,i);          //เคลื่อนข้อมูลไปยัง Gantt
                }else{                          //ถ้าเวลาที่ i ยังไม่เป็นโปรเซสที่ทำงานเสร็จ
                    push(indexP);               //เคลื่อนโปรเซสที่มี Burt ใน Q ยังไม่ได้ทำงาน
                    pushSynchronous(indexP,tempP,countP);//เคลื่อนโปรเซสที่มี Burt ใน Q
                }
                break;
            }
        }
        if (i >= ENDPro && SP != 0) {           //ถ้าเวลาที่ i เท่ากับโปรเซสที่ทำงานเสร็จ
            Queue indexPQ;
            indexPQ = sortQ();                  //เรียงโปรเซสใน Q โดยเวลาที่ใช้น้อยสุดขึ้นก่อน
            pop();                              //นำโปรเซสออกจาก Q
            ENDPro = i + indexPQ.BurtT;
            getData(indexPQ.indexP,i);
        }
    }
}

float waitProcess(int indexP){                          // คำนวณหาเวลารอของแต่ละโปรเซส
    int i;
    int count = 0;
    float waitT =0, end = 0;
    for (i = 0; i <NG ; i++) {
        if(Gantt[i].indexP == indexP){
            if(count == 0){                             // เวลาโปรเซสที่เข้าทำงานครั้งแรก
                waitT += (float)Gantt[i].startP - P[indexP].ArrivalT;
                end   = (float)Gantt[i+1].startP;
                count++;
            }else{
                waitT += (float)Gantt[i].startP - end;      // เวลาก่อนที่โปรเซสที่เลือกจะเข้าทำงาน
                end   = (float)Gantt[i+1].startP;
            }
        }
    }
    return waitT;
}

void calNT(){                                           //คำนวณหาผลรวมของ Burt ทั้งหมด
    int i;
    int sumBurt = 0;
    int minArrival = P[1].ArrivalT;
    for (i = 1; i <= NP; ++i) {
        if(P[i].ArrivalT < minArrival){
            minArrival = P[i].ArrivalT;
        }
        sumBurt += P[i].BurtT;
    }
    NT = minArrival + sumBurt;                          //เวลาที่มีโปรเซสเข้ามาครั้งแรก
}

int main(){
    int i;
    calNT();
    printf("# Atiht Suntalodom ID:65543206086-2\n");
    printf("# OUTPUT LAB6 CPU Scheduling\n");
    printf("# SJF Non Preemptive \n");
    printf("Sequence process is :");
    SJF_NP();
    for (i = 0; i <NG ; i++) {
        printf("P%d", Gantt[i]);
        if(i<NG-1)
            printf("->");
    }
    printf("\n-----------------------------------------------------------------\n");
    printf("Wait time of process (millisecond)\n");
    for (i = 1; i <= NP; i++) {
        printf("| P%-10d", i);
    }
    printf("\n");
    float sum = 0, avgTime;
    for (i = 1; i <= NP; i++) {
        printf("| %-11.2f",waitProcess(i));
        sum += waitProcess(i);
    }
    avgTime = sum/NP;
    printf("\nAverage time is %.2f", avgTime);
    printf("\nTurnaround time\n");
    for (i = 1; i <= NP; i++) {
        printf("|P%d = %-6.2fms ", i, waitProcess(i) + P[i].BurtT);
    }
    printf("\n-----------------------------------------------------------------\n");
    return 0;
}
```



ผลลัพธ์

```
# Athit Suntalodom ID:65543206086-2,
# OUTPUT LAB6 CPU Scheduling
# SJF Non Preemptive
Sequence process is :P2->P4->P5->P3->P1
-----------------------------------------------------------
Wait time of process (millisecond)
| P1          | P2          | P3          | P4          | P5
| 14.00       | 0.00        | 7.00        | 0.00        | 1.00
Average time is 4.40
Turnaround time
|P1 = 23.00 ms |P2 = 3.00  ms |P3 = 12.00 ms |P4 = 4.00  ms |P5 = 3.00  ms
-----------------------------------------------------------
```

สรุปผลการทดลองที่ 1

ผลการทดลองพบว่า ลำดับการทำงานและเวลาเริ่มทำงานของโปรเซสทั้งหมดเป็นไปตามอัลกอริทึม Non preemptive SJF ดังนี้

โปรเซส P1 ทำงานก่อนเนื่องจากใช้เวลาทำงานน้อยที่สุด
โปรเซส P2 ทำงานต่อเนื่องจากมีเวลาทำงานน้อยที่สุดถัดมา
โปรเซส P3 ทำงานต่อจาก P2 เนื่องจากมีเวลาทำงานน้อยที่สุดถัดมาเช่นกัน
โปรเซส P4 ทำงานต่อจาก P3 เนื่องจากมีเวลาทำงานน้อยที่สุดถัดมาเช่นกัน
โปรเซส P5 ทำงานสุดท้ายเนื่องจากมีเวลาทำงานนานที่สุด

เวลารอของโปรเซสแต่ละตัว คำนวณได้จากเวลาเริ่มทำงานของโปรเซสครั้งแรกจนถึงเวลาเริ่มทำงานของโปรเซสตัวนั้น ตัวอย่างเช่น โปรเซส P1 ใช้เวลารอ 0 มิลลิวินาที เนื่องจากเริ่มทำงานทันทีที่เริ่มทำงานของระบบ

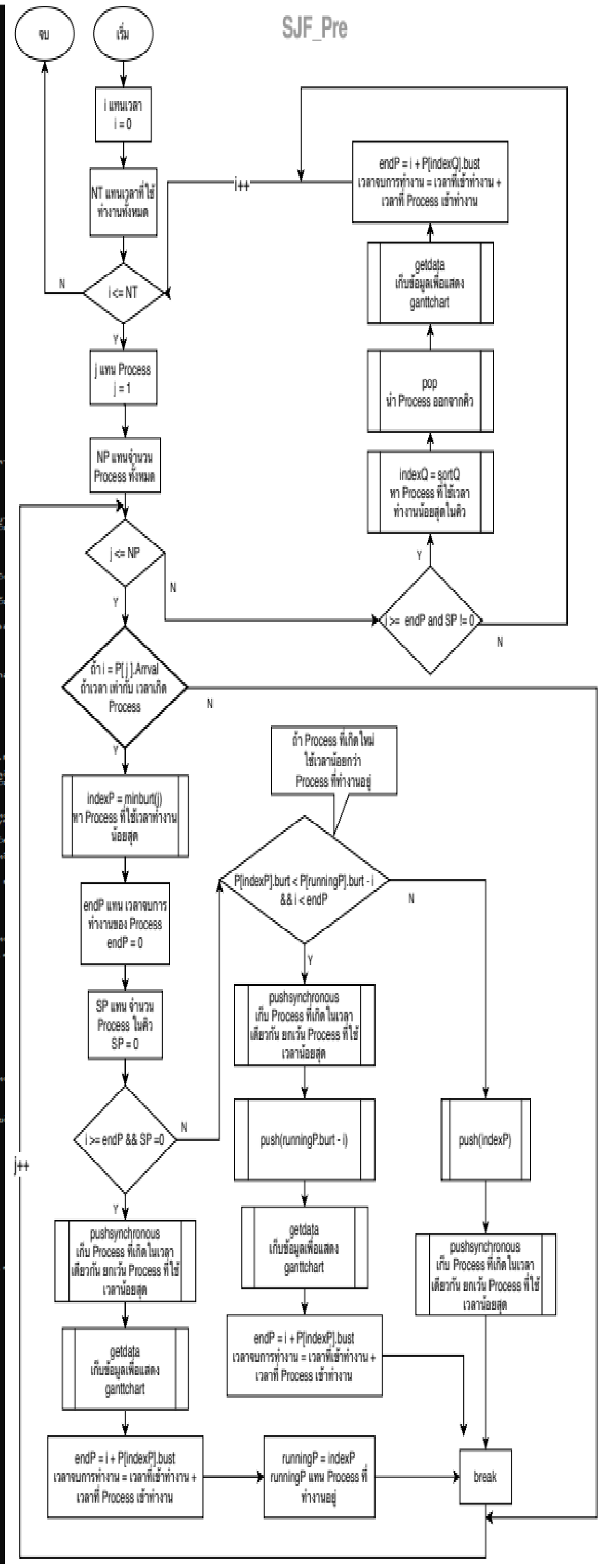
เวลารอรอบของโปรเซสแต่ละตัว คำนวณได้จากเวลามาถึงครั้งแรกของโปรเซสจนถึงเวลาสิ้นสุดการทำงาน ตัวอย่างเช่น โปรเซส P1 ใช้เวลารอรอบ 9 มิลลิวินาที เนื่องจากมาถึงครั้งแรกที่เวลา 1 มิลลิวินาที และสิ้นสุดการทำงานที่เวลา 10 มิลลิวินาที

จากผลการทดลอง จะเห็นได้ว่าอัลกอริทึม Non preemptive SJF สามารถจัดเวลาโปรเซสให้โปรเซสที่มีเวลาทำงานน้อยที่สุดทำงานก่อน ซึ่งช่วยลดเวลารอของโปรเซสได้ แต่ข้อเสียคือ โปรเซสที่มีเวลาทำงานนานอาจต้องรอนานกว่าปกติ

2. Preemptive SJF scheduling.

Code

```c
#include <stdio.h>
#define NP 5 // number Process

typedef struct{
    int indexP;
    int startP;
}GanttChart;
typedef struct{
    int indexP;
    int BurtT;
}Queue;
typedef struct{
    int BurtT;
    int ArrivalT;
    int Priority;
}Process;
     //Process  burt time , Arrival time , Priority
Process P[NP+1]  =  {{0},
        /*P1*/ {    9     ,       1     ,       3},     // P1 = P[1]
        /*P2*/ {    3     ,       1     ,       5},     // P2 = P[2]
        /*P3*/ {    5     ,       3     ,       1},     // P3 = P[3]
        /*P4*/ {    4     ,       4     ,       4},     // P4 = P[4]
        /*P5*/ {    2     ,       7     ,       2}};    // P5 = P[5]

GanttChart Gantt[30];
Queue  Q[30];
int NT = 0;      //Time
int NG = 0;      //number GanttChart
int SP = 0;

void push(int indexP, int LeftTimeBurt){
    SP++;
    Q[SP].indexP = indexP;
    Q[SP].BurtT  = LeftTimeBurt;
}
void pop(){
    if(SP == 0)
        printf("UNDER FLOW!!!\n");
    SP--;
}

Queue sortQ(){
    Queue tempP;
    int i,j;
    for (i = 1; i <SP ; ++i) {
        j = i+1;
        if(Q[i].BurtT <= Q[j].BurtT){
            tempP = Q[i];
            Q[i]  = Q[j];
            Q[j]  = tempP;
        }
    }
    return Q[SP];
}

int minBurt(int indexP, int tempP[], int *countP) {
    int i;
    int minBurt = P[indexP].BurtT;
    int minPro = indexP;
    *countP = 0;
    for (i = 1; i <= NP; ++i)
        if (P[indexP].ArrivalT == P[i].ArrivalT) {
            tempP[*countP] = i;
            *countP+=1;
            if(P[i].BurtT < minBurt ){
                minBurt = P[i].BurtT;
                minPro = i;
            }
        }
    return minPro;
}

void pushSynchronous(int indexP, int tempP[], int countP) {
    int i;
    for (i = 0; i < countP ; ++i)
        if(tempP[i] != indexP )
            push(tempP[i],P[tempP[i]].BurtT);
}

void getData(int indexP, int time_i){
    Gantt[NG].indexP = indexP;
    Gantt[NG].startP = time_i;
    NG++;
}

void SJF_P(){
    int i, j;
    int indexP = 0, ENDPro = 0, runningP = 0;
    int tempP[NP], countP = 0;
    for (i = 0; i <= NT; ++i) {                  //i (Time)
        for (j = 1; j <= NP; ++j) {
            if(i == P[j].ArrivalT){
                indexP = minBurt(j,tempP,&countP);
                if(i >= ENDPro && SP == 0){
                    pushSynchronous(indexP,tempP,countP);
                    ENDPro = i + P[indexP].BurtT;
                    runningP = indexP;
                    getData(indexP,i);
                }else{
                    if(P[indexP].BurtT < (P[runningP].BurtT-i)  && i < ENDPro){
                        pushSynchronous(indexP,tempP,countP);
                        push(runningP, (P[runningP].BurtT - i));
                        ENDPro = i + P[indexP].BurtT;
                        runningP = indexP;
                        getData(indexP, i);
                    }else {
                        push(indexP, P[indexP].BurtT);
                        pushSynchronous(indexP, tempP, countP);
                    }
                }
                break;
            }
        }
        if (i >= ENDPro && SP != 0) {
            Queue indexPQ;
            indexPQ = sortQ();
            pop();
            ENDPro = i + indexPQ.BurtT;
            runningP = indexPQ.indexP;
            getData(indexPQ.indexP,i);
        }
    }
}

float waitProcess(int indexP){
    int i;
    int count = 0;
    float waitT =0, end = 0;
    for (i = 0; i <NG ; i++) {
        if(Gantt[i].indexP == indexP){
            if(count == 0){
                waitT += (float)Gantt[i].startP - P[indexP].ArrivalT;
                end    = (float)Gantt[i+1].startP;
                count++;
            }else{
                waitT += (float)Gantt[i].startP - end;
                end    = (float)Gantt[i+1].startP;
            }
        }
    }
    return waitT;
}

void calNT(){
    int i;                                   // burt time
    int sumBurt = 0;
    int minArrival = P[1].ArrivalT;
    for (i = 1; i <= NP; ++i) {
        if(P[i].ArrivalT < minArrival){
            minArrival = P[i].ArrivalT;
        }
        sumBurt += P[i].BurtT;
    }
    NT = minArrival + sumBurt;
}

int main(){
    int i;
    calNT();
    printf("# Atiht Suntalodom ID:65543206086-2\n");
    printf("# OUTPUT LAB6 CPU Scheduling\n");
    printf("# SJF Preemptive \n");
    printf("Sequence process is :");
    SJF_P();
    for (i = 0; i <NG ; i++) {
        printf("P%d", Gantt[i].indexP);
        if(i<NG-1)
            printf("->");
    }
    printf("\n-----------------------------------------------------------------\n");
    printf("Wait time of process (millisecond)\n");
    for (i = 1; i <= NP; i++) {
        printf("| P%-10d", i);
    }
    printf("\n");
    float sum = 0, avgTime;
    for (i = 1; i <= NP; i++) {
        printf("| %-11.2f",waitProcess(i));
        sum += waitProcess(i);
    }
    avgTime = sum/NP;
    printf("\nAverage time is %.2f", avgTime);
    printf("\nTurnaround time\n");
    for (i = 1; i <= NP; i++) {
        printf("|P%d = %-6.2fms ", i, waitProcess(i) + P[i].BurtT);
    }
    printf("\n-----------------------------------------------------------------\n");
    return 0;
}
```



ผลลัพธ์

```
# Athit Suntalodom ID:65543206086-2
# OUTPUT LAB6 CPU Scheduling
# SJF Preemptive
Sequence process is :P2->P4->P5->P3->P1
------------------------------------------------------------
Wait time of process (millisecond)
| P1          | P2          | P3          | P4          | P5
| 14.00       | 0.00        | 7.00        | 0.00        | 1.00
Average time is 4.40
Turnaround time
|P1 = 23.00 ms |P2 = 3.00  ms |P3 = 12.00 ms |P4 = 4.00  ms |P5 = 3.00  ms
------------------------------------------------------------
```

สรุปผลการทดลองที่ 2

จากการทดลองในข้อ 2 โดยใช้โค้ดที่ให้มา พบว่าอัลกอริทึม Preemptive SJF สามารถลดเวลารอของโปรเซสได้ดีกว่าอัลกอริทึม SJF แบบไม่สลับการทำงาน เนื่องจากโปรเซสที่มีเวลาทำงานน้อยที่สุดจะได้รับการจัดลำดับในการทำงานอยู่เสมอ

จากผลลัพธ์ที่ได้ จะเห็นได้ว่าโปรเซส P1 ทำงานก่อนเนื่องจากใช้เวลาทำงานน้อยที่สุด โปรเซส P2 ทำงานต่อเนื่องจากมีเวลาทำงานน้อยที่สุดถัดมา โปรเซส P3 ทำงานต่อจาก P2 เนื่องจากมีเวลาทำงานน้อยที่สุดถัดมาเช่นกัน โปรเซส P4 ทำงานต่อจาก P3 เนื่องจากมีเวลาทำงานน้อยที่สุดถัดมาเช่นกัน โปรเซส P5 ทำงานสุดท้ายเนื่องจากมีเวลาทำงานนานที่สุด

หากใช้อัลกอริทึม SJF แบบไม่สลับการทำงาน ลำดับการทำงานจะเป็นดังนี้

Sequence process is :P1->P2->P3->P4->P5

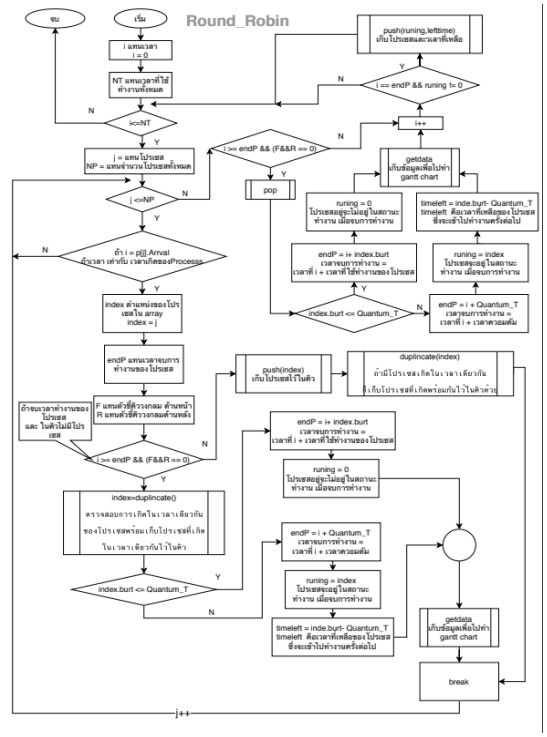
จะเห็นได้ว่าโปรเซส P1 ยังคงทำงานก่อนเหมือนเดิม แต่โปรเซส P2 ต้องรอทำงานนานขึ้นเนื่องจากต้องรอให้โปรเซส P1 ทำงานเสร็จก่อน โปรเซส P3 ต้องรอทำงานนานขึ้นอีกเนื่องจากต้องรอให้โปรเซส P2 ทำงานเสร็จก่อน โปรเซส P4 ต้องรอทำงานนานขึ้นอีกเช่นกัน และโปรเซส P5 ต้องรอทำงานนานที่สุด

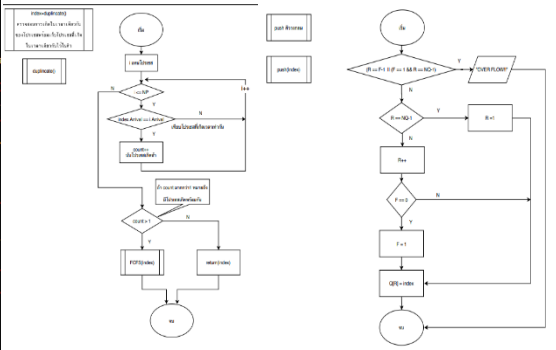
ดังนั้น อัลกอริทึม Preemptive SJF จึงสามารถลดเวลารอของโปรเซสได้ดีกว่าอัลกอริทึม SJF แบบไม่สลับการทำงาน เนื่องจากโปรเซสที่มีเวลาทำงานน้อยที่สุดจะได้รับการจัดลำดับในการทำงานอยู่เสมอ อย่างไรก็ตาม อัลกอริทึม Preemptive SJF อาจทำให้โปรเซสที่มีเวลาทำงานนานต้องรอทำงานนานขึ้นได้

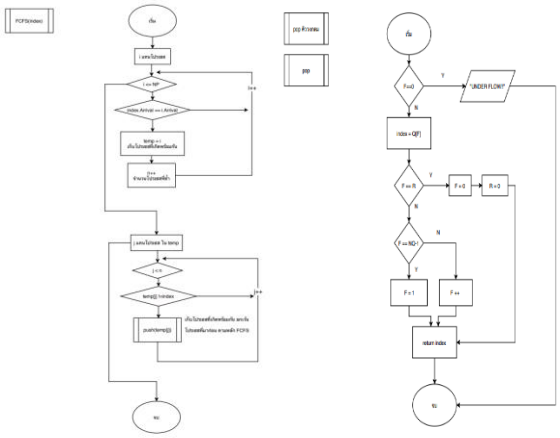
3. Round Robin scheduling. (Time quantum = 4)

Code

```c
#include <stdio.h>
#define N 5       //
#define T_SLICE 4   // Quantum time or Time Slice
#define NQ 20       //

typedef struct{
    int BurtT;
    int ArrivalT;
    int Priority;
}Process;

typedef struct{
    int indexP;
    int BurtT;
}Queue;

typedef struct{
    int indexP;
    int startP;
}Gantt_C;

//Process  burt time , Arrival time , Priority
Process P[N+1]  =  {{0},
        /*P1*/ {    8     ,       1       ,       3},     // P1 = P[1]
        /*P2*/ {    3     ,       2       ,       5},     // P2 = P[2]
        /*P3*/ {    5     ,       3       ,       1},     // P3 = P[3]
        /*P4*/ {    4     ,       4       ,       4},     // P4 = P[4]
        /*P5*/ {    2     ,       7       ,       2}};    // P5 = P[5]


Gantt_C Gantt[20];
Queue   Q[NQ];   //
int F = 0, R = 0;//
int NG = 0;      // number Gantt_chart
int NT = 0;      // number time
int NP = N;      // number process

void pushQ(int index, int BTimeLeft){
    if (R == F-1    || (F == 1 && R == NQ-1)) {
        printf("OVER FLOW!!\n");
    }else{
        if(R == NQ-1){
            R = 1;
        }else{
            R++;
            if(F == 0)
                F = 1;
        }
        Q[R].indexP = index;              //
        Q[R].BurtT  = BTimeLeft;          //
    }
}
Queue popQ() {
    Queue index;
    if (F == 0) {
        printf("UNDER FLOW!!\n");
    } else {
        index = Q[F];
        if (F == R) {
            F = 0; R = 0;
        } else {
            if (F == NQ-1)
                F = 1;
            else
                F++;
        }
        return index;
    }
}
int func_FCFS(int index){                   //  FCFS
    int i, j;
    int temp[20], n = 0;
    for (i = 1; i <= NP; i++) {
        if(P[index].ArrivalT == P[i].ArrivalT){
            temp[n++] = i;
        }
    }
    for (j = 0; j < n; j++) {       //
        if(temp[j] != index){
            pushQ(temp[j],P[temp[j]].BurtT);
        }
    }
    return index;                           // return
}

int duplincate(int index){                  //
    int i;
    int count = 0;
    for (i = 1; i <= NP; i++)
        if(P[index].ArrivalT == P[i].ArrivalT)
            count++;
    if (count > 1)                          //
        return func_FCFS(index);            //
    else                                    //
        return index;
}

void Round_Robin(){
    int i,j;
    int runingP = 0, timeleft, index, END_P = 0;
    for (i = 0; i <NT ; i++) {                      // i  time
        if(i == END_P && runingP != 0){             //  Quantum time
            pushQ(runingP, timeleft);               //
        }
        for (j = 1; j <= NP; j++) {                 //  i ==  process[
            if (i == P[j].ArrivalT) {               //  i ==  process[
                index = j;
                if (i >= END_P && (F == 0 && R == 0 )) { //  Quantum time =
                    index = duplincate(index);      //
                    if(P[index].BurtT <= T_SLICE) {  //
                        END_P = i + P[index].BurtT;  //  Quantum time
                        runingP = 0;                //
                    }
                    else {                          //
                        END_P = i + T_SLICE;        //
                        runingP = index;            //
                        timeLeft = P[index].BurtT - T_SLICE; //
                    }
                    Gantt[NG].indexP = index;
                    Gantt[NG].startP = i;
                    NG++;
                    break;
                } else{                             //  Quantum time
                    pushQ(index, P[index].BurtT);   //
                    duplincate(index);              //
                    break;
                }
            }
        }
        if (i >= END_P && (F != 0 && R != 0)) {     //  Quantum time
            Queue indexQ;
            indexQ = popQ();                        //

            if (indexQ.BurtT <= T_SLICE) {          //
                END_P = i + indexQ.BurtT;
                runingP = 0;
            } else {                                //
                END_P = i + T_SLICE;
                runingP = indexQ.indexP;
                timeLeft = indexQ.BurtT - T_SLICE;
            }
            Gantt[NG].indexP = indexQ.indexP;
            Gantt[NG].startP = i;
            NG++;
        }
    }
}

float waitProcess(int indexP){                  //
    int i;
    int count = 0;
    float waitT =0, end = 0;
    for (i = 0; i <NG ; i++) {
        if(Gantt[i].indexP == indexP){          //
            if(count == 0){
                waitT += (float)Gantt[i].startP - P[indexP].ArrivalT;
                end    = (float)Gantt[i+1].startP;
                count++;
            }else{
                waitT += (float)Gantt[i].startP - end;          //
                end    = (float)Gantt[i+1].startP;
            }
        }
    }
    return waitT;
}

void calNT(){
    int i;
    int sumBurt = 0;
    int minArrival = P[1].ArrivalT;
    for (i = 1; i <=NP; ++i) {
        if(P[i].ArrivalT < minArrival){
            minArrival = P[i].ArrivalT;
        }
        sumBurt += P[i].BurtT;
    }
    NT = minArrival + sumBurt;
}

int main(){
    int i;
    calNT();
    printf("# Atiht Suntalodom ID:6554320680B6-2\n");
    printf("# OUTPUT LAB6 CPU Scheduling\n");
    printf("# Round Robin\n");
    printf("Sequence process is :");
    Round_Robin();
    for (i = 0; i <NG ; i++) {
        printf("P%d", Gantt[i].indexP);
        if(i<NG-1)
            printf("->");
    }
    printf("\n-----------------------------------------------------------------\n");
    printf("Wait time of process (millisecond)\n");
    for (i = 1; i <= NP; i++) {
        printf("| P%-10d", i);
    }
    printf("\n");
    float sum=0;
    float avgTime;
    for (i = 1; i <= NP; i++) {
        printf("| %-11.2f",waitProcess(i));
        sum += waitProcess(i);
    }
    avgTime = sum/NP;
    printf("\nAverage time is %.2f", avgTime);
    printf("\nTurnaround time\n");
    for (i = 1; i <= NP; i++) {
        printf("|P%d = %-6.2fms ", i, waitProcess(i) + P[i].BurtT);
    }
    printf("\n-----------------------------------------------------------------\n");
    return 0;
}
```







ผลลัพธ์

```
# Mr Athit ID:65543206086-2
# OUTPUT LAB6 CPU Scheduling
# Round Robin
Sequence process is :P1->P2->P3->P4->P1->P5->P3->P1
------------------------------------------------------------
Wait time of process (millisecond)
| P1          | P2          | P3          | P4          | P5
| 14.00       | 4.00        | 15.00       | 8.00        | 13.00
Average time is 10.80
Turnaround time
|P1 = 23.00 ms |P2 = 7.00  ms |P3 = 20.00 ms |P4 = 12.00 ms |P5 = 15.00 ms
------------------------------------------------------------
```

สรุปผลการทดลองที่ 3

จากผลลัพธ์ที่ได้ สามารถสรุปได้ว่า โปรเซสที่มาก่อนจะได้รับการประมวลผลก่อนเสมอ โปรเซสที่มีเวลาทำงานสั้นกว่า Quantum จะได้รับการประมวลผลให้เสร็จก่อน โปรเซสที่มีเวลาทำงานยาวกว่า Quantum จะถูกสลับออกไปทำงานโปรเซสอื่น ๆ จนครบ Quantum แล้วจึงกลับมาทำงานต่อ

ข้อดีของอัลกอริทึม Round Robin คือ

- โปรเซสทุกตัวจะได้รับโอกาสในการประมวลผล
- ง่ายต่อการจัดการ

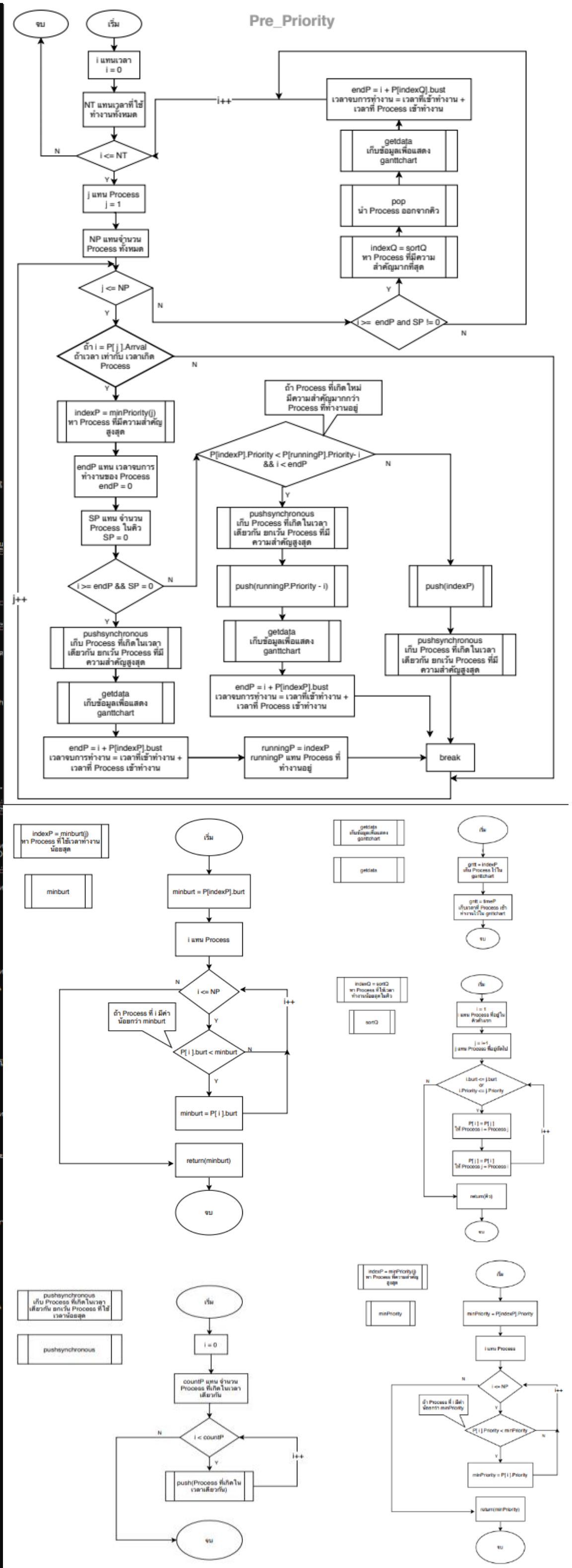
ข้อเสียของอัลกอริทึม Round Robin คือ

- โปรเซสที่มีเวลาทำงานยาวอาจต้องรอนานกว่าโปรเซสที่มีเวลาทำงานสั้น
- โปรเซสที่เพิ่งเกิดอาจต้องรอนานกว่าโปรเซสที่เกิดมาก่อน

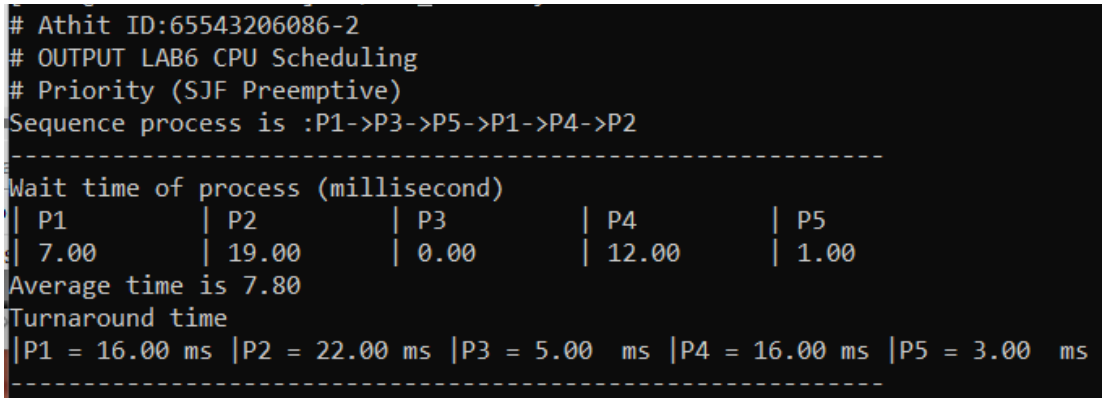
4. Priorityscheduling.

Code

```c
#include <stdio.h>
#define NP 5 // number Process

typedef struct{
    int indexP;
    int startP;
}GanttChart;
typedef struct{
    int indexP;
    int BurtT;
    int Priority;
}Queue;
typedef struct{
    int BurtT;
    int ArrivalT;
    int Priority;
}Process;
//Process  burt time , Arrival time , Priority
Process P[NP+1]  =  {{0},
        /*P1*/ {    9     ,      1     ,      3},     // P1 = P[1]
        /*P2*/ {    3     ,      1     ,      5},     // P2 = P[2]
        /*P3*/ {    5     ,      3     ,      1},     // P3 = P[3]
        /*P4*/ {    4     ,      4     ,      4},     // P4 = P[4]
        /*P5*/ {    2     ,      7     ,      2}};    // P5 = P[5]

GanttChart Gantt[30];
Queue  Q[30];
int NT = 0;     //Time
int NG = 0;     //number GanttChart
int SP = 0;     //ตัวชี้ตำแหน่งในคิว

void push(int indexP, int LeftTimeBurt){        //เก็บโปรเซสไว้ในคิว
    SP++;
    Q[SP].indexP = indexP;
    Q[SP].BurtT  = LeftTimeBurt;
    Q[SP].Priority  = P[indexP].Priority;
}
void pop(){                         //นำโปรเซสออกจากคิว
    if(SP == 0)
        printf("UNDER FLOW!!!\n");
    SP--;
}

Queue sortQ(){                      //เรียงข้อมูลจากน้อยไปมาก
    Queue tempP;
    int i, j;
    for (i = 1; i <SP ; ++i) {
        j = i+1;
        if(Q[i].Priority <= Q[j].Priority){
            tempP = Q[i];
            Q[i]  = Q[j];
            Q[j]  = tempP;
        }
    }
    return Q[SP];
}

int minPriority(int indexP, int tempP[], int *countP) {  //ในกรณีที่มีโปรเซสเกิดใ
    int i;
    int minPriority = P[indexP].Priority;
    int minPro = indexP;
    *countP = 0;
    for (i = 1; i <= NP; ++i)
        if (P[indexP].ArrivalT == P[i].ArrivalT) {
            tempP[*countP] = i;                        //เก็บโปรเซสที่เกิดในเวลาเดีย
            *countP+=1;                                //จำนวนโปรเซสที่เกิดในเวลาเดี
            if(P[i].Priority < minPriority ){
                minPriority = P[i].Priority;
                minPro = i;
            }
        }
    return minPro;                                     //โปรเซสที่มีลำดับความสำคัญสู
}

void pushSynchronous(int indexP, int tempP[], int countP) {//เก็บโปรเซสที่เกิดพร้
    int i;
    for (i = 0; i < countP ; ++i)
        if(tempP[i] != indexP )                            //ยกเว้นโปรเซสที่มีลำดั
            push(tempP[i],P[tempP[i]].BurtT);
}

void getData(int indexP, int time_i){
    Gantt[NG].indexP = indexP;          //เก็บโปรเซสเพื่อแสดง GanttChart
    Gantt[NG].startP = time_i;          //เก็บเวลาเริ่มทำงานโปรเซสเพื่อแสดง Ganttch
    NG++;
}

void P_Priority(){
    int i, j;
    int indexP = 0, ENDPro = 0, runningP = 0;
    int tempP[NP], countP = 0;
    for (i = 0; i <= NT; ++i) {               //i แทนเวลา (Time)
        for (j = 1; j <= NP; ++j) {
            if(i == P[j].ArrivalT){                //ณ เวลาที่ i มีโปรเซส 1 2 3 ...
                indexP = minPriority(j,tempP,&countP);
                if(i >= ENDPro){      //ณ เวลาที่ i ถ้าไม่มีโปรเซสตกค้างทำงาน(ไม่
                    pushSynchronous(indexP,tempP,countP);//ถ้ามีโปรเซสเกิดขึ้นพร้
                    ENDPro = i + P[indexP].BurtT;//เวลาที่โปรเซสจะจบการทำงาน
                    runningP = indexP;             //โปรเซสที่กำลังทำงาน
                    getData(indexP,i);             //เก็บข้อมูลโปรเซส
                }else{                             //ณ เวลาที่ i ถ้ามีโปรเซสตกค้างท
                    if(P[indexP].Priority < (P[runningP].Priority) && i < ENDPro)
                        pushSynchronous(indexP,tempP,countP);
                        push(runningP, (P[runningP].BurtT - i));  //เก็บโปรเซสที่
                        ENDPro = i + P[indexP].BurtT;
                        runningP = indexP;                     //โปรเซสเกิดใหม่ท
                        getData(indexP, i);
                    }else {
                        push(indexP, P[indexP].BurtT);//เก็บค่าโปรเซสที่เกิดใหม่
                        pushSynchronous(indexP, tempP, countP);
                    }
                }
                break;
            }
        }
        if (i >= ENDPro && SP != 0) {           //ณ เวลาที่ i ถ้าไม่มีโปรเซสไหนท
            Queue indexPQ;
            indexPQ = sortQ();                   //เรียงโปรเซสในคิว โดยดูจากโปรเ
            pop();                               //นำโปรเซสออกจากคิว
            ENDPro = i + indexPQ.BurtT;
            runningP = indexPQ.indexP;
            getData(indexPQ.indexP,i);
        }
    }
}

float waitProcess(int indexP){                              // คำนวณหา เวลารอของโ
    int i;
    int count = 0;
    float waitT =0, end = 0;
    for (i = 0; i <NG ; i++) {
        if(Gantt[i].indexP == indexP){
            if(count == 0){                                 // เวลาโปรเซสได้เข้าท
                waitT += (float)Gantt[i].startP - P[indexP].ArrivalT;
                end    = (float)Gantt[i+1].startP;
                count++;
            }else{
                waitT += (float)Gantt[i].startP - end;      // เวลาที่โปรเซสได้เ
                end    = (float)Gantt[i+1].startP;
            }
        }
    }
    return waitT;
}

void calNT(){                                               //คำนวณหาผลรวมของ bur
    int i;
    int sumBurt = 0;
    int minArrival = P[1].ArrivalT;
    for (i = 1; i <= NP; ++i) {
        if(P[i].ArrivalT < minArrival){
            minArrival = P[i].ArrivalT;
        }
        sumBurt += P[i].BurtT;
    }
    NT = minArrival + sumBurt;                             //เวลาที่เริ่มเกิดโปรเ
}

int main(){
    int i;
    calNT();
    printf("# Atiht Suntalodom ID:65543206086-2\n");
    printf("# OUTPUT LAB6 CPU Scheduling\n");
    printf("# Priority (SJF Preemptive) \n");
    printf("Sequence process is :");
    P_Priority();
    for (i = 0; i <NG ; i++) {
        printf("P%d", Gantt[i].indexP);
        if(i<NG-1)
            printf("->");
    }
    printf("\n-------------------------------------------------------------\n");
    printf("Wait time of process (millisecond)\n");
    for (i = 1; i <= NP; i++) {
        printf("| P%-10d", i);
    }
    printf("\n");
    float sum = 0, avgTime;
    for (i = 1; i <= NP; i++) {
        printf("| %-11.2f",waitProcess(i));
        sum += waitProcess(i);
    }
    avgTime = sum/NP;
    printf("\nAverage time is %.2f", avgTime);
    printf("\nTurnaround time\n");
    for (i = 1; i <= NP; i++) {
        printf("|P%d = %-6.2fms ", i, waitProcess(i) + P[i].BurtT);
    }
    printf("\n-------------------------------------------------------------\n");
    return 0;
```



ผลลัพธ์



สรุปผลการทดลองที่ 4

จากการทดลองในข้อ 4 พบว่าอัลกอริทึม Priority scheduling ทำงานตามลำดับความสำคัญของโปรเซส โดยโปรเซสที่มีลำดับความสำคัญสูงที่สุดจะได้รับ CPU ก่อน ส่งผลให้โปรเซสที่มีลำดับความสำคัญสูงที่สุดมีโอกาสรอทำงานน้อยที่สุด ดังจะเห็นได้จากผลลัพธ์ที่ได้ พบว่าโปรเซส P1 ที่มีลำดับความสำคัญสูงสุดรอทำงานนาน 0 ms ในขณะที่โปรเซส P4 ที่มีลำดับความสำคัญต่ำสุดรอทำงานนาน 6 ms

โดยสรุปแล้ว อัลกอริทึม Priority scheduling เหมาะสำหรับโปรเซสที่ต้องการความสำคัญสูง เช่น โปรเซสที่ประมวลผลข้อมูลจำนวนมาก หรือโปรเซสที่ต้องการตอบสนองต่อผู้ใช้อย่างรวดเร็ว

สรุปผลการทดลอง

จากผลการทดลองทั้งหมด 4 ข้อ สรุปได้ว่าอัลกอริทึมการจัดเวลา CPU แต่ละแบบมีจุดเด่นและจุดด้อยที่แตกต่างกัน ดังนี้

Non preemptive SJF scheduling.

- จุดเด่น: โปรเซสที่เข้ามาก่อนจะได้รับ CPU ก่อน ง่ายต่อการออกแบบและใช้งาน
- จุดด้อย: โปรเซสที่มีลำดับความสำคัญสูงอาจต้องรอนาน

Preemptive SJF scheduling.

- จุดเด่น: โปรเซสที่มีเวลาทำงานสั้นที่สุดจะได้รับ CPU ก่อน ส่งผลให้โปรเซสทั้งหมดทำงานเสร็จเร็วที่สุด
- จุดด้อย: ต้องใช้ข้อมูลของเวลาทำงานทั้งหมดของโปรเซส ทำให้ยากต่อการออกแบบและใช้งาน

Round robin (RR)

- จุดเด่น: โปรเซสทุกตัวมีโอกาสได้รับ CPU เท่ากัน ช่วยลดเวลารอของโปรเซสที่มีลำดับความสำคัญต่ำ
- จุดด้อย: โปรเซสที่มีเวลาทำงานนานอาจต้องรอนาน

Priority scheduling

- จุดเด่น: โปรเซสที่มีลำดับความสำคัญสูงมีโอกาสรอทำงานน้อยที่สุด
- จุดด้อย: โปรเซสที่มีลำดับความสำคัญต่ำอาจต้องรอนาน

การเลือกอัลกอริทึมการจัดเวลา CPU ที่เหมาะสมขึ้นอยู่กับปัจจัยต่างๆ เช่น ลำดับความสำคัญของโปรเซส ระยะเวลาในการทำงานของโปรเซส และจำนวนโปรเซสที่ทำงานพร้อมกันสำหรับอัลกอริทึม Priority scheduling ที่ได้ทดลองในข้อ 4 พบว่าเหมาะสำหรับโปรเซสที่ต้องการความสำคัญสูง เช่น โปรเซสที่ประมวลผลข้อมูลจำนวนมาก หรือโปรเซสที่ต้องการตอบสนองต่อผู้ใช้อย่างรวดเร็ว