ใบงานที่ 6

เรื่อง CPU scheduling

จัดทำโดย

นางสาวรัชนีกร เชื้อดี 65543206077-1

เสนอ

อาจารย์ปิยพล ยืนยงสถาวร

ใบงานนี้เป็นส่วนหนึ่งของรายวิชา ระบบปฏิบัติการ

หลักสูตรวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์

มหาวิทยาลัยเทคโนโลยีราชมงคลล้านนา

ประจำภาคที่ 2 ปีการศึกษา 2566

ใบงานที่ 6

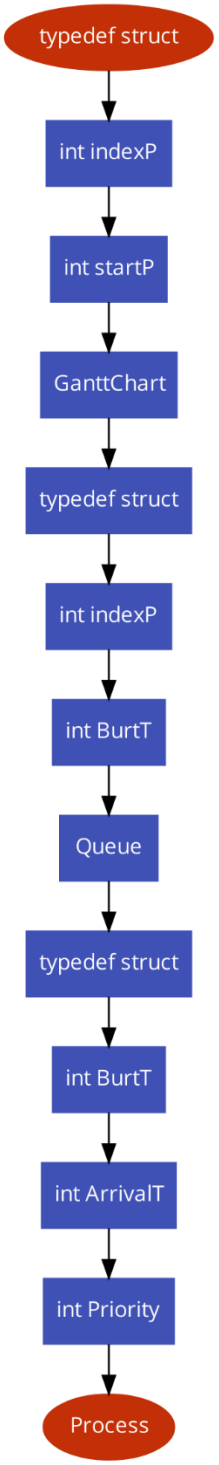
CPU scheduling

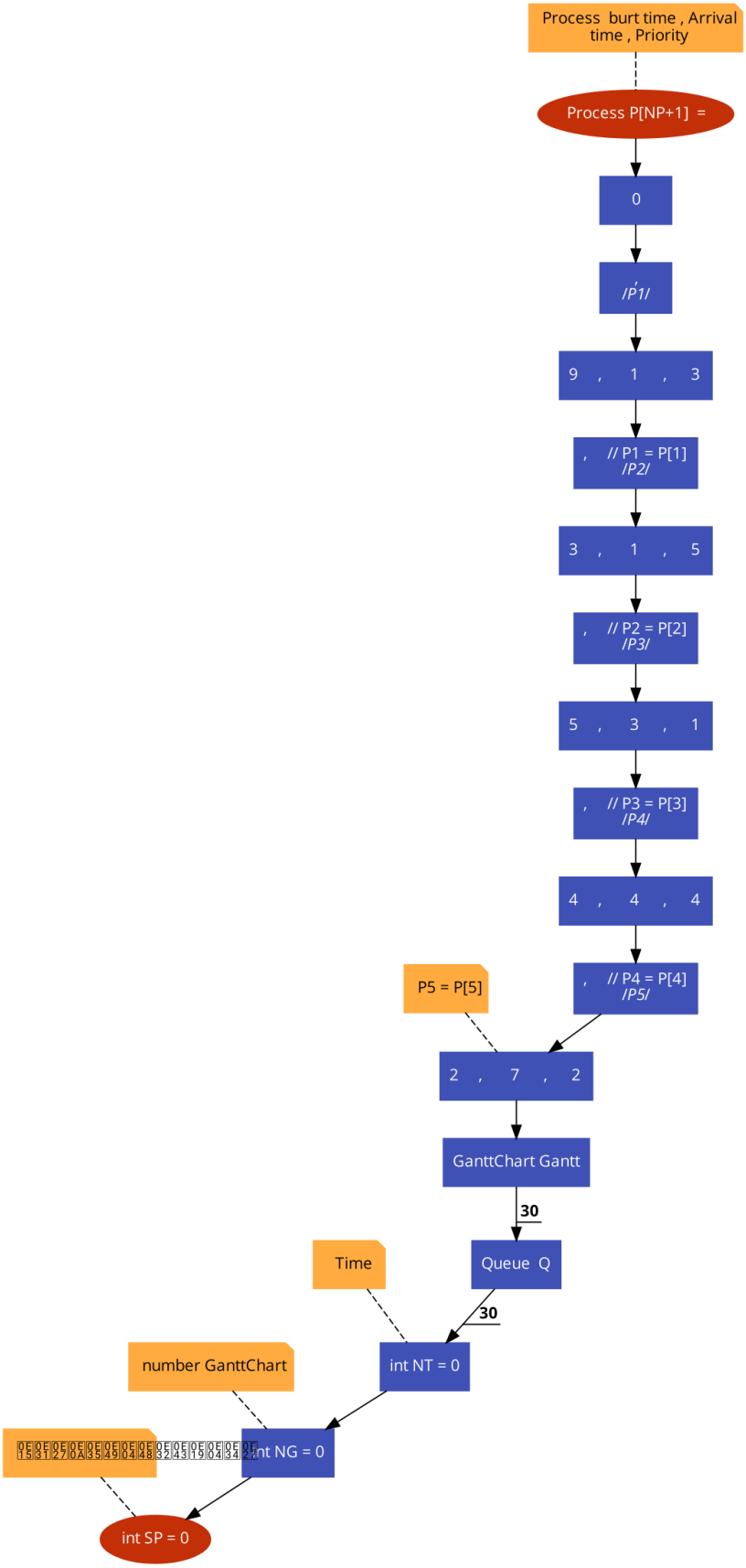
ลำดับขั้นการทดลอง

1. **ออกแบบโปรแกรมด้วยผังงาน(Flowchart)**
2. **เขียนโปรแกรมตามที่ออกแบบไว้ด้วยภาษาซีบนระบบปฏิบัติการ CentOS**
3. **เขียนอธิบายโค้ดโปรแกรมอย่างละเอียด**
4. **บันทึกผลการทดลอง และสรุปผล**
5. **ส่งไฟล์รูปเล่มใบงานพร้อมอัดคลิปแสดงผลการรันโปรแกรมที่เขียนขึ้นมาใน MS Team**

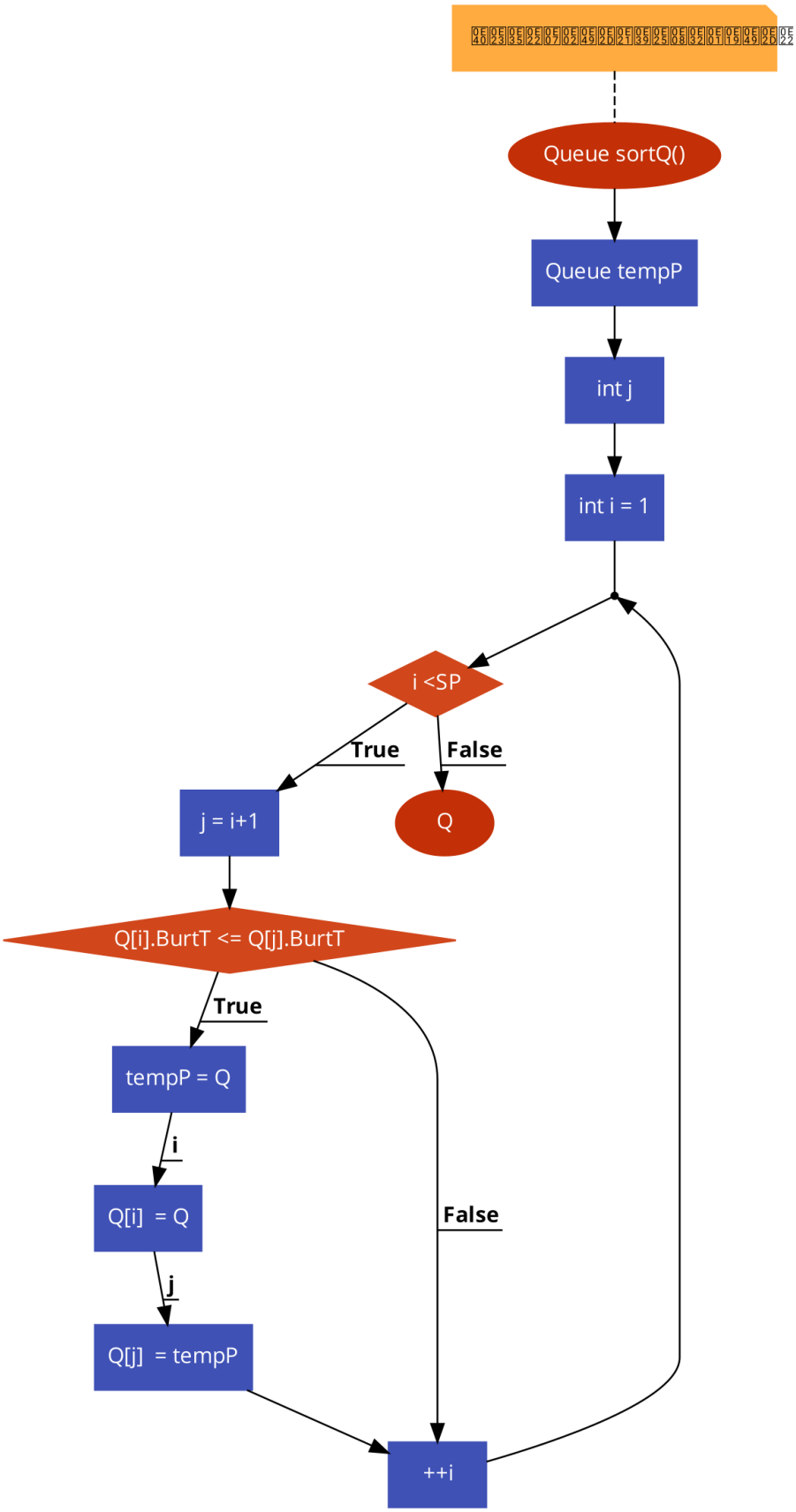
Flowchart

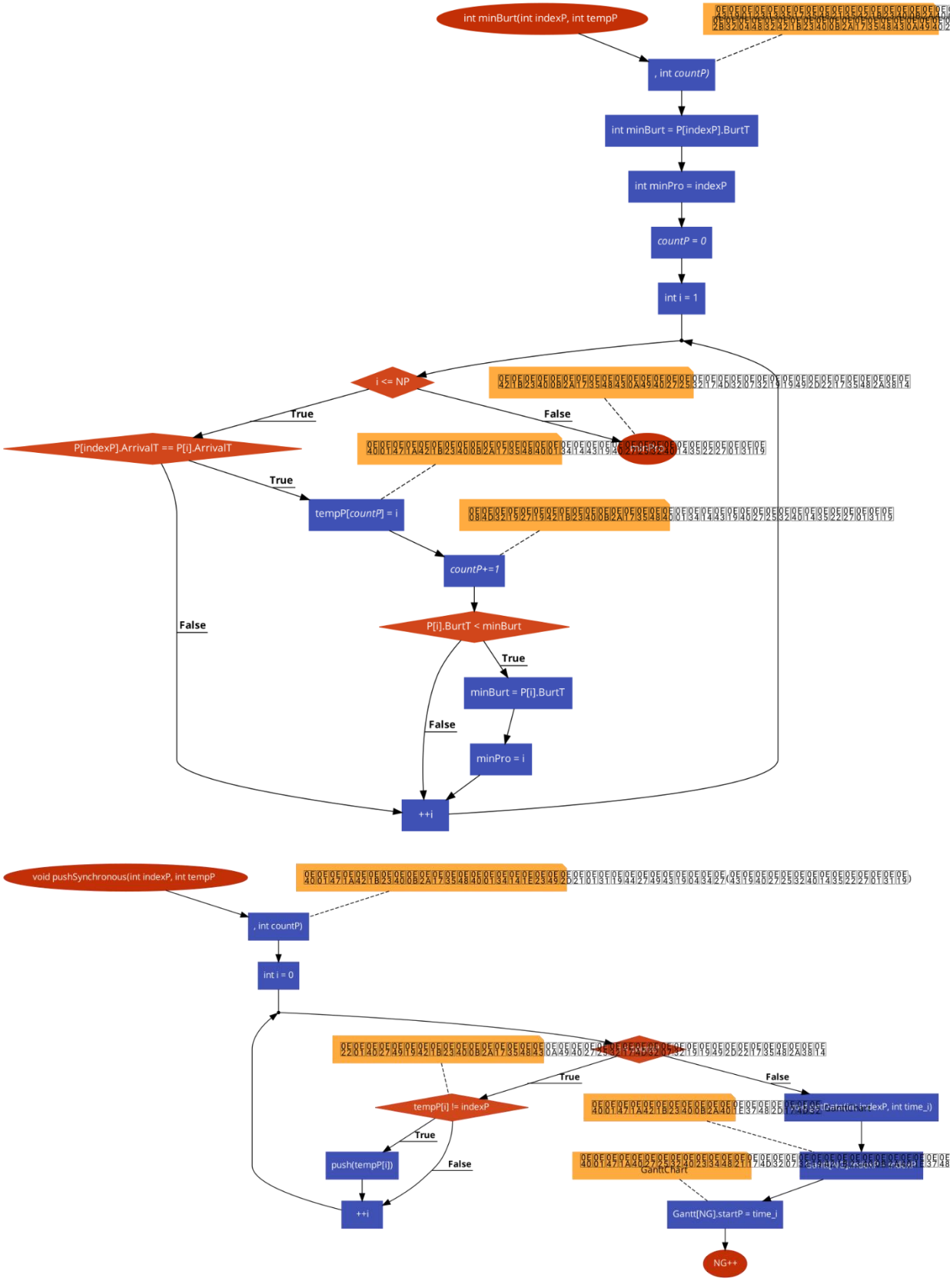
Non preemptive SJF scheduling

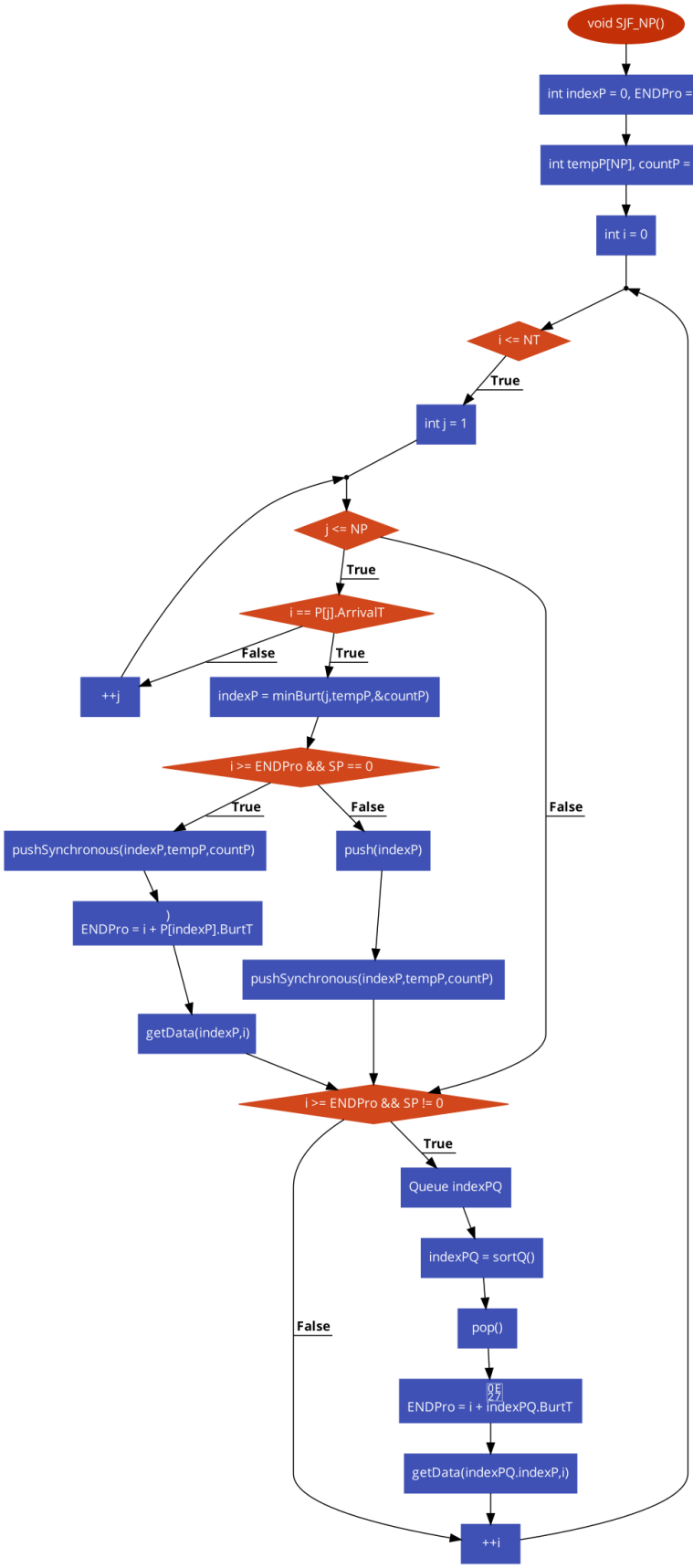


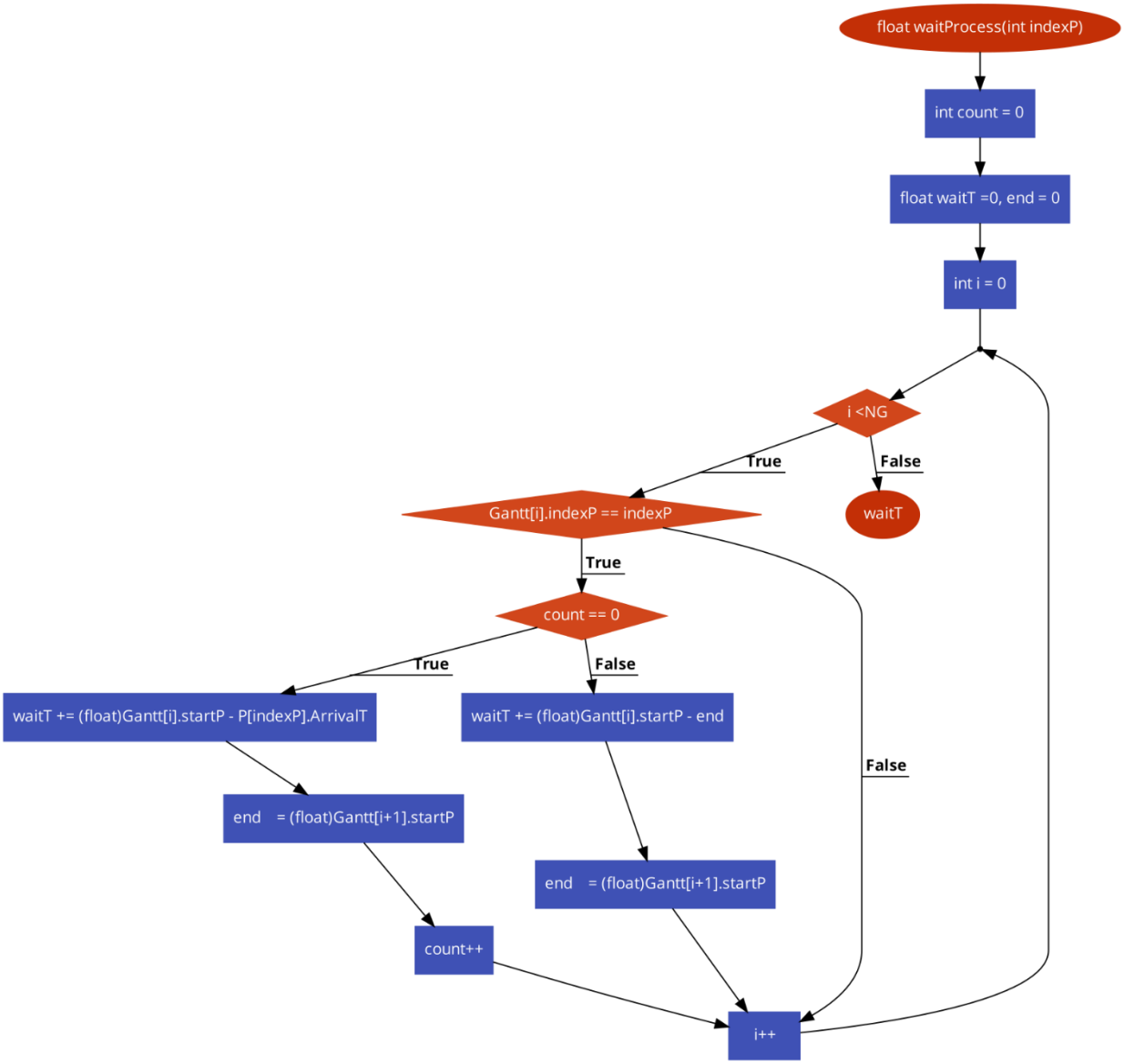

Process burt time , Arrival time , Priority
Process P[NP+1] =
0
/P1/
9 , 1 , 3
, // P1 = P[1] /P2/
3 , 1 , 5
, // P2 = P[2] /P3/
5 , 3 , 1
, // P3 = P[3] /P4/
4 , 4 , 4
, // P4 = P[4] /P5/
P5 = P[5]
2 , 7 , 2
GanttChart Gantt
30
Queue Q
Time
30
int NT = 0
number GanttChart
nt NG = 0
int SP = 0

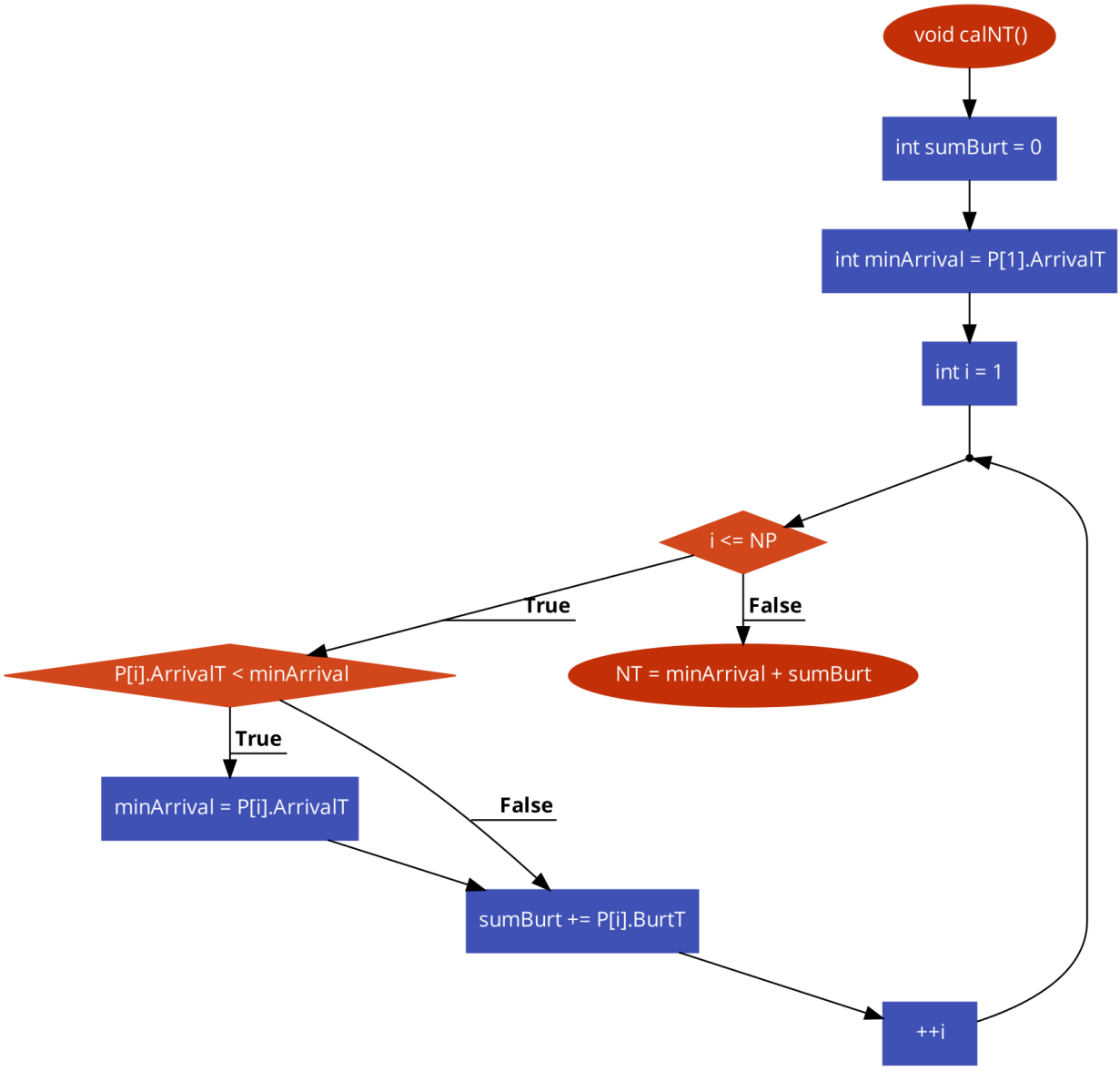

void push(int indexP)
SP++
Q[SP].indexP = indexP
Q[SP].BurtT = P[indexP].BurtT
void pop()
SP == 0
True
False
printf("UNDER FLOW!!!\n")
SP--


Queue sortQ()
Queue tempP
int j
int i = 1
i <SP
True
False
j = i+1
Q
Q[i].BurtT <= Q[j].BurtT
True
tempP = Q
i
Q[i] = Q
False
j
Q[j] = tempP
++i

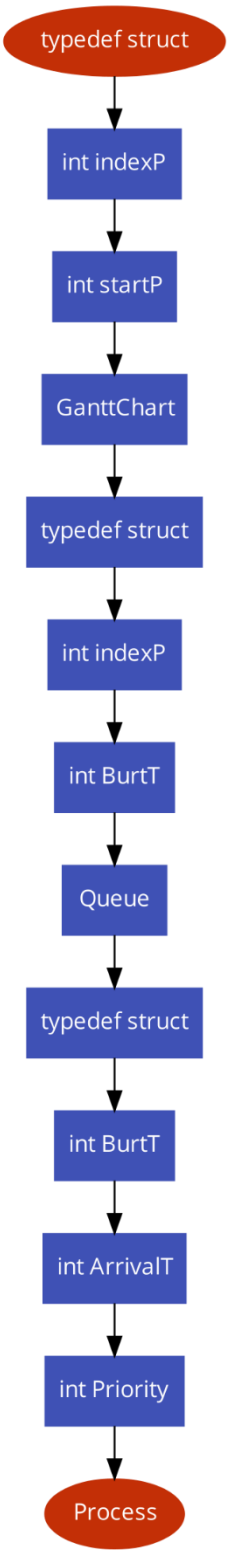

int minBurt(int indexP, int tempP
, int countP)
int minBurt = P[indexP].BurtT
int minPro = indexP
countP = 0
int i = 1
i <= NP
True
False
P[indexP].ArrivalT == P[i].ArrivalT
True
tempP[countP] = i
countP+=1
P[i].BurtT < minBurt
True
False
minBurt = P[i].BurtT
False
minPro = i
++i
void pushSynchronous(int indexP, int tempP
, int countP)
int i = 0
tempP[i] != indexP
True
False
push(tempP[i])
True
False
++i
Gantt[NG].startP = time_i
NG++

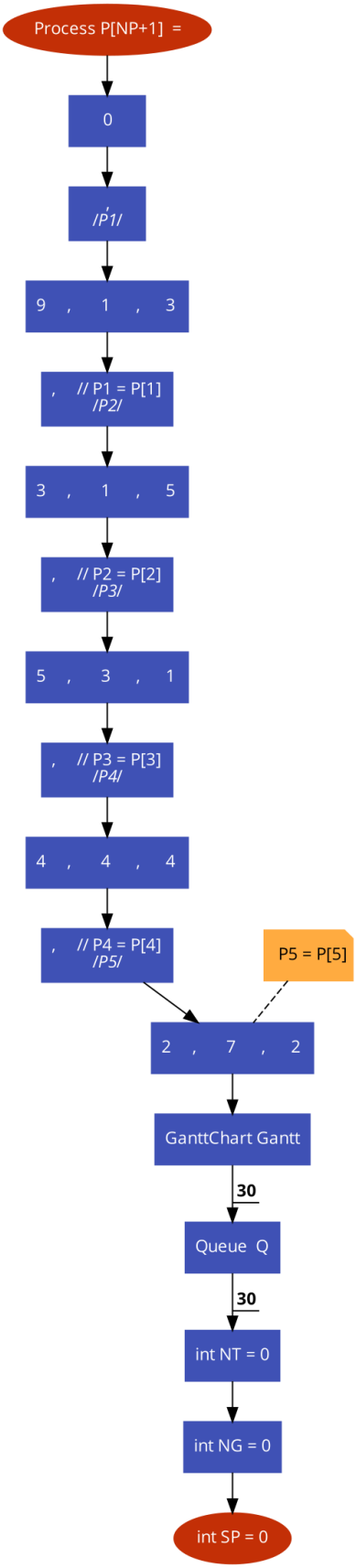

void SJF_NP()
int indexP = 0, ENDPro = 0
int tempP[NP], countP = 0
int i = 0
i <= NT
True
int j = 1
j <= NP
True
False
i == P[j].ArrivalT
False
True
++j
indexP = minBurt(j,tempP,&countP)
i >= ENDPro && SP == 0
True
False
False
pushSynchronous(indexP,tempP,countP)
push(indexP)
)
ENDPro = i + P[indexP].BurtT
pushSynchronous(indexP,tempP,countP)
getData(indexP,i)
i >= ENDPro && SP != 0
True
Queue indexPQ
indexPQ = sortQ()
False
pop()
ENDPro = i + indexPQ.BurtT
getData(indexPQ.indexP,i)
++i

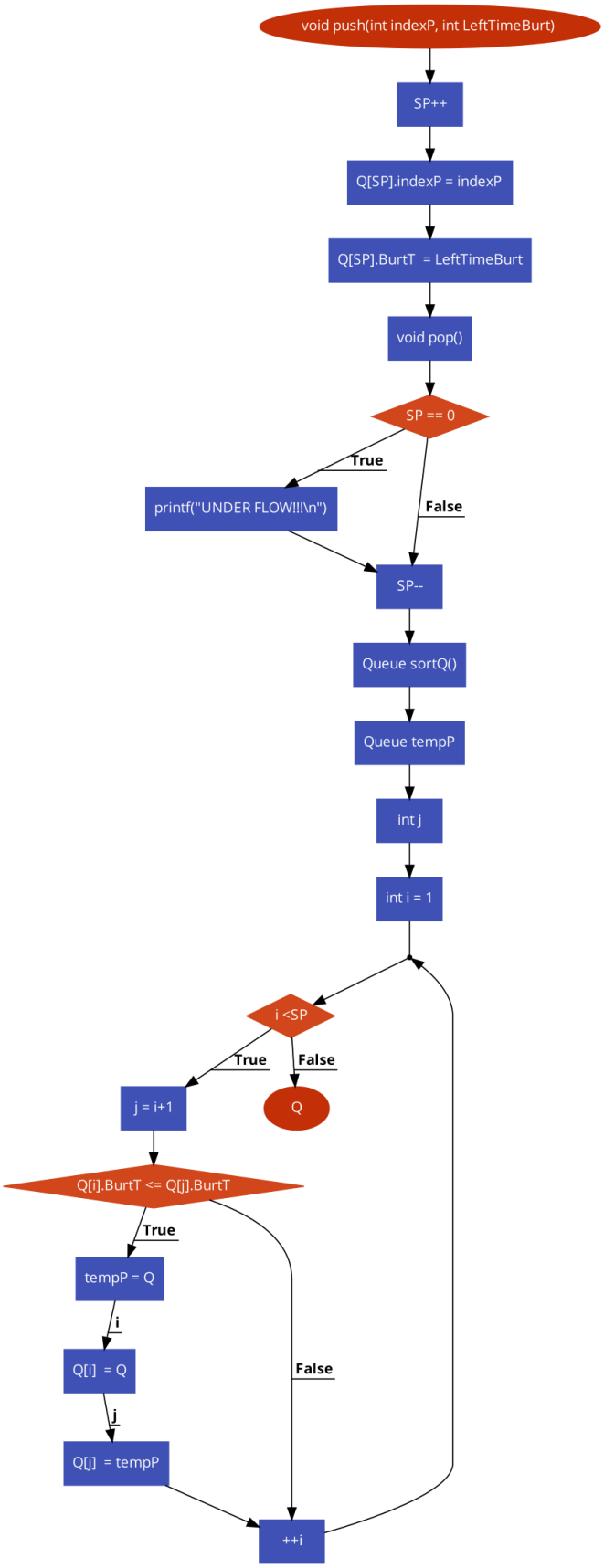

float waitProcess(int indexP)
int count = 0
float waitT =0, end = 0
int i = 0
i <NG
True
False
waitT
Gantt[i].indexP == indexP
True
False
count == 0
True
False
waitT += (float)Gantt[i].startP - P[indexP].ArrivalT
waitT += (float)Gantt[i].startP - end
end = (float)Gantt[i+1].startP
end = (float)Gantt[i+1].startP
count++
i++

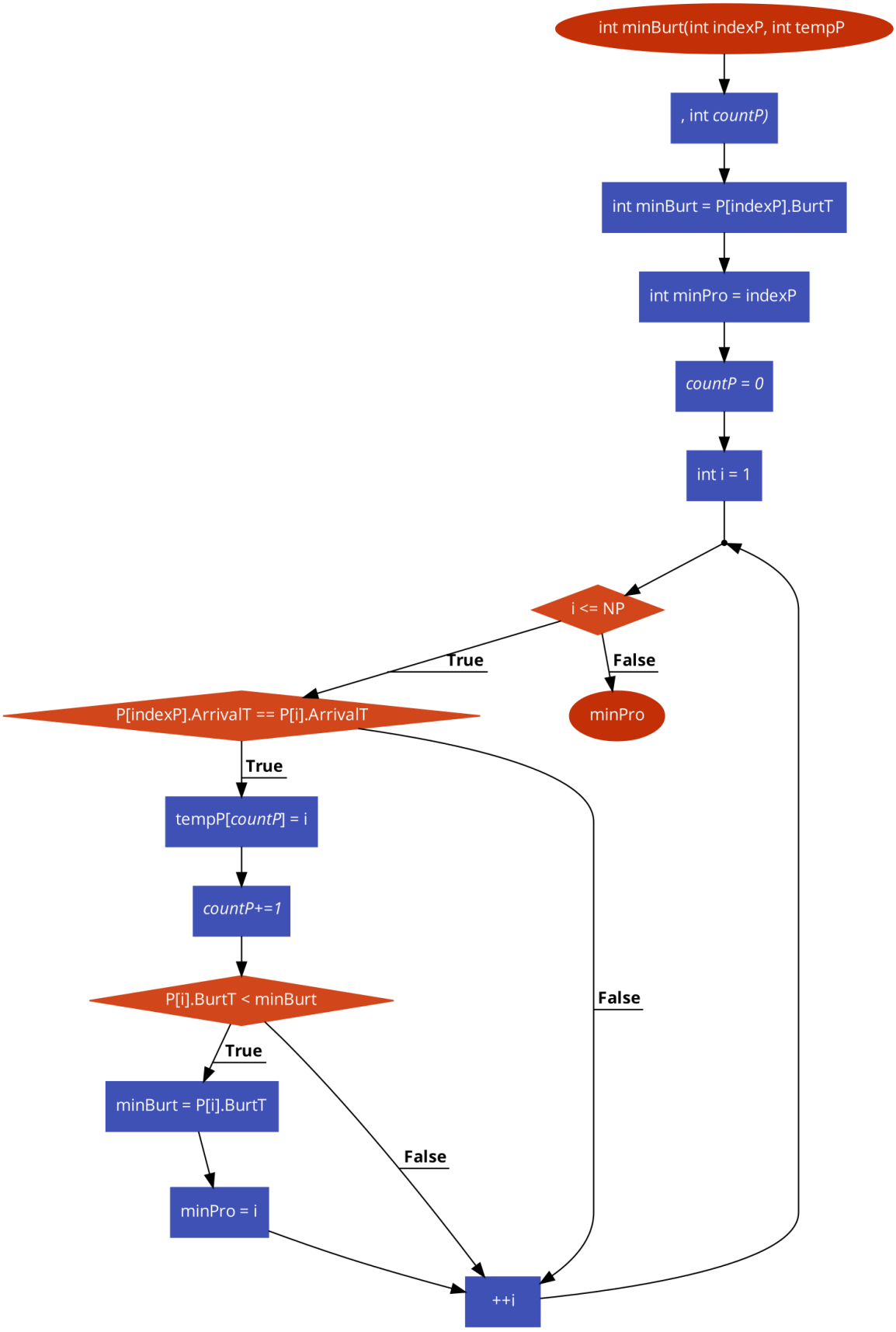

void calNT()
int sumBurt = 0
int minArrival = P[1].ArrivalT
int i = 1
i <= NP
True
False
P[i].ArrivalT < minArrival
NT = minArrival + sumBurt
True
minArrival = P[i].ArrivalT
False
sumBurt += P[i].BurtT
++i

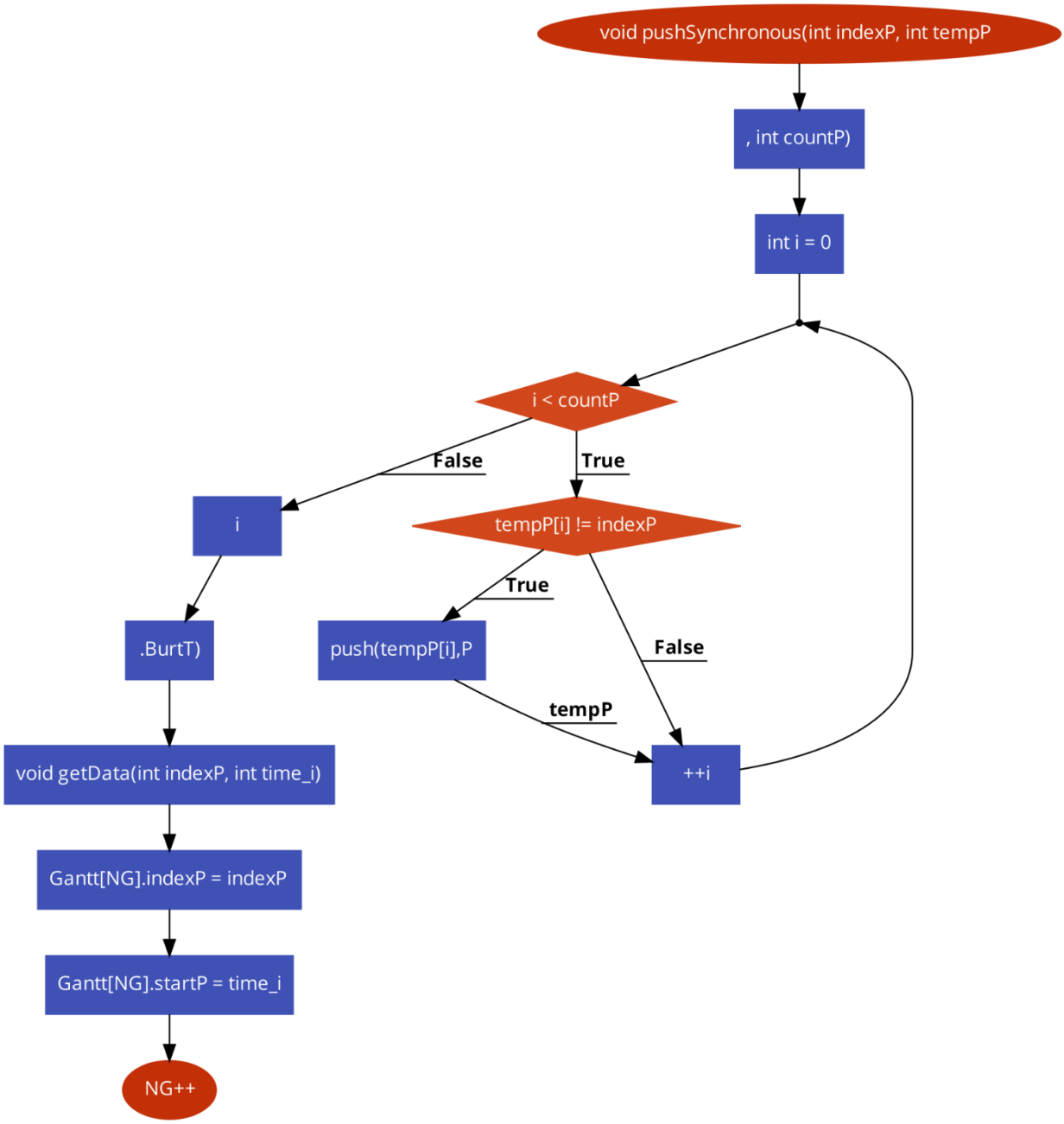
Preemptive SJF scheduling

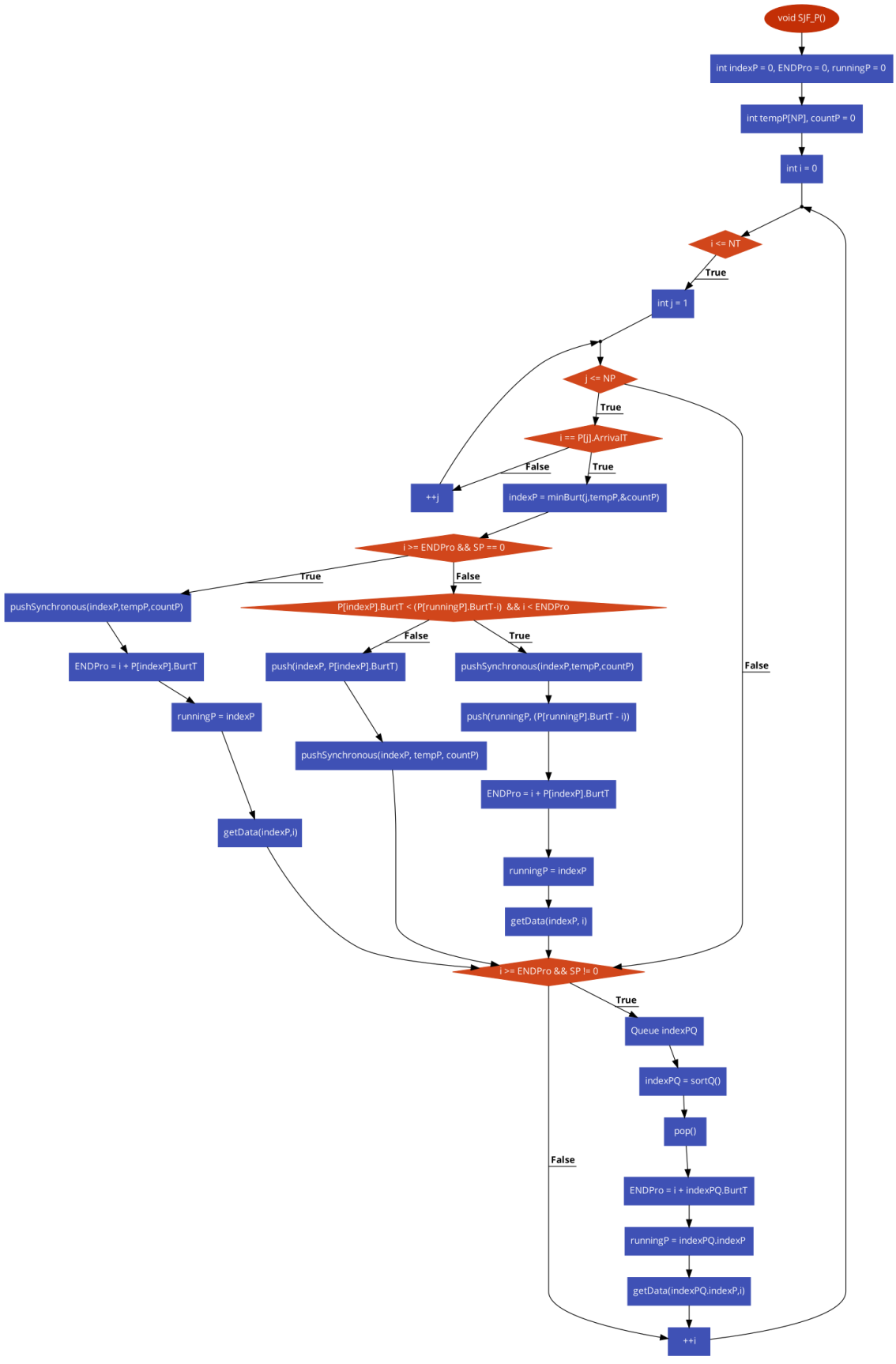


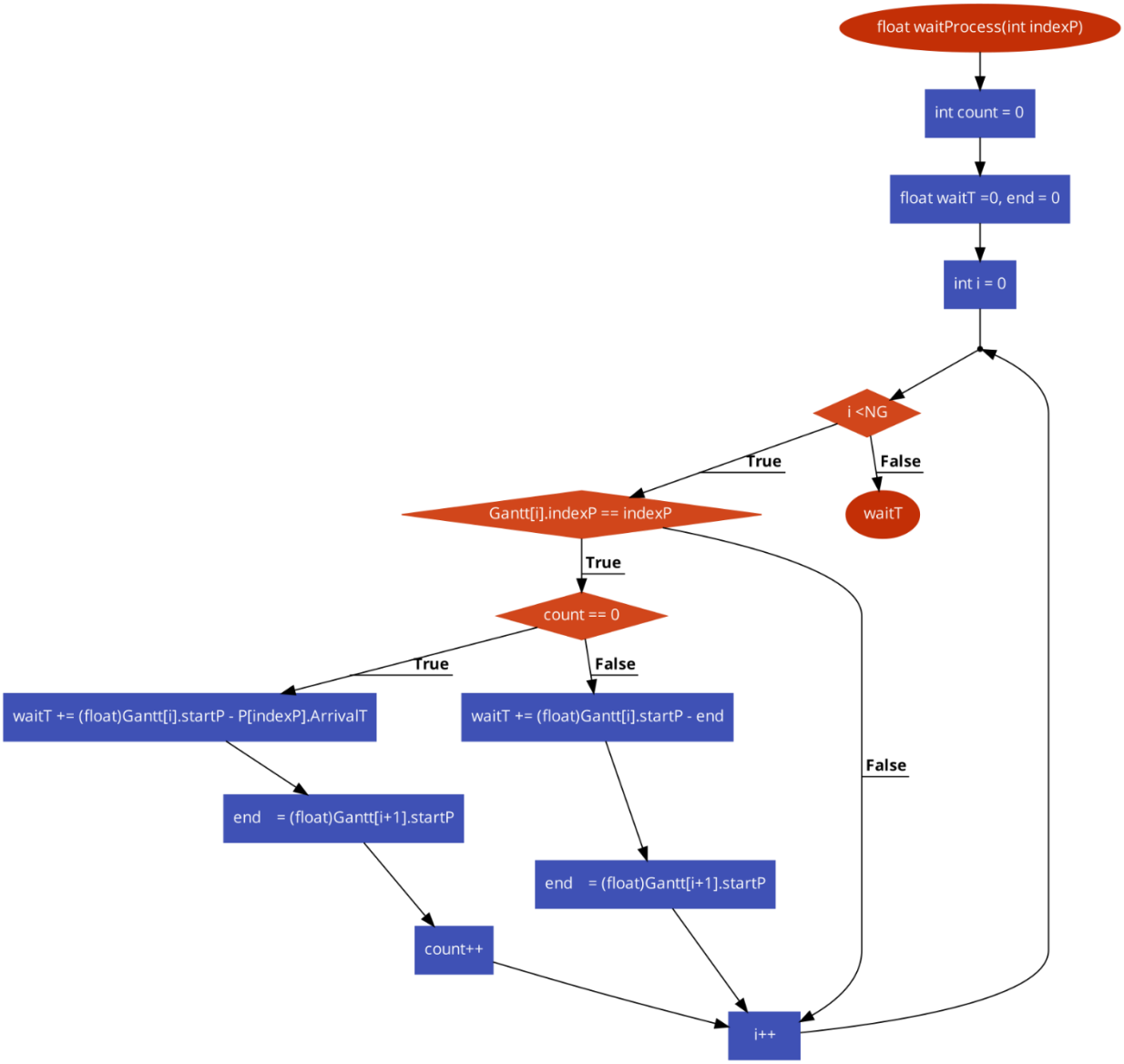

Process P[NP+1] =
0
/P1/
9 , 1 , 3
, // P1 = P[1]
/P2/
3 , 1 , 5
, // P2 = P[2]
/P3/
5 , 3 , 1
, // P3 = P[3]
/P4/
4 , 4 , 4
, // P4 = P[4]
/P5/
P5 = P[5]
2 , 7 , 2
GanttChart Gantt
30
Queue Q
30
int NT = 0
int NG = 0
int SP = 0

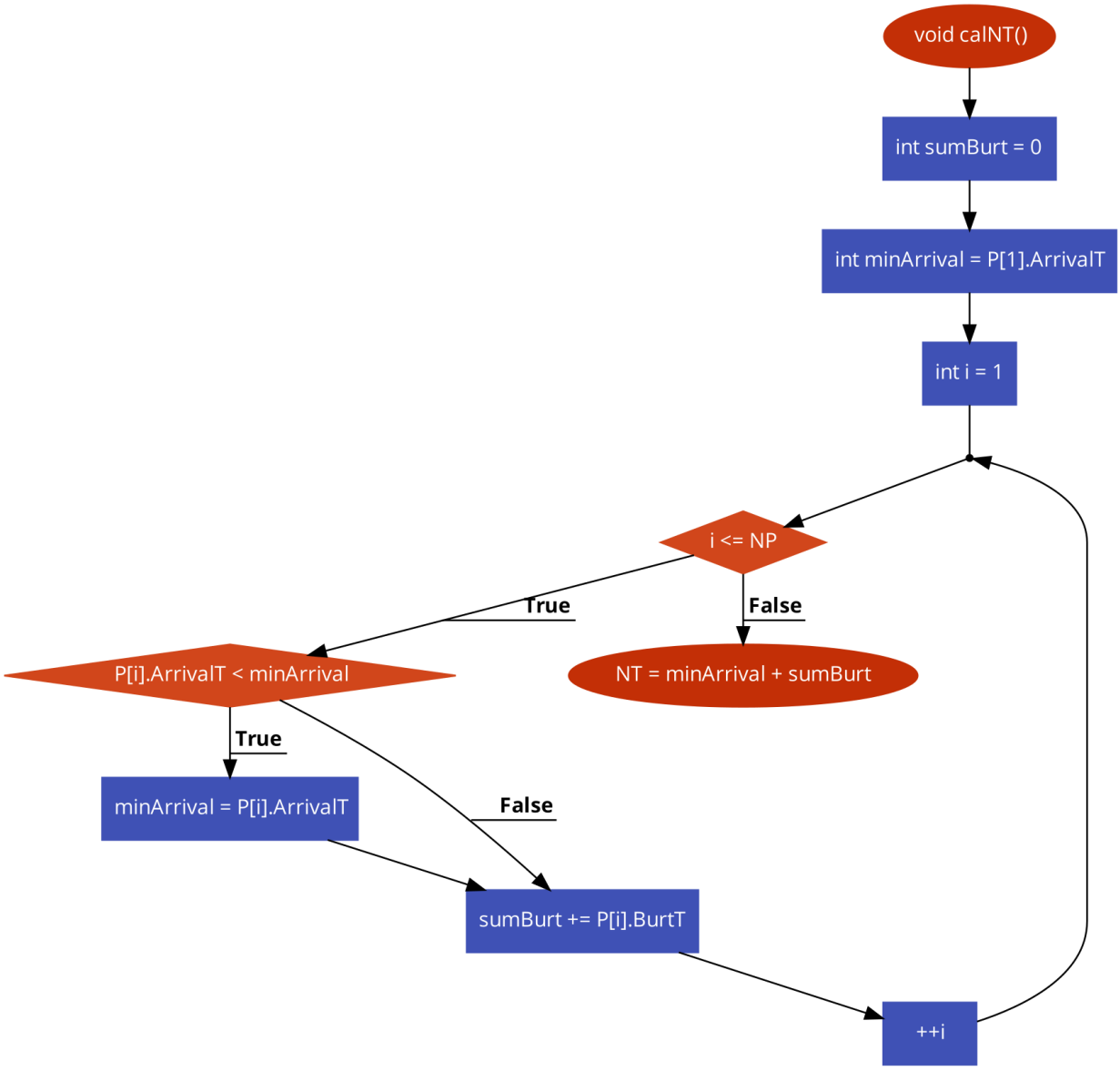

void push(int indexP, int LeftTimeBurt)
SP++
Q[SP].indexP = indexP
Q[SP].BurtT = LeftTimeBurt
void pop()
SP == 0
True
False
printf("UNDER FLOW!!!\n")
SP--
Queue sortQ()
Queue tempP
int j
int i = 1
i <SP
True
False
j = i+1
Q
Q[i].BurtT <= Q[j].BurtT
True
tempP = Q
i
Q[i] = Q
j
Q[j] = tempP
False
++i

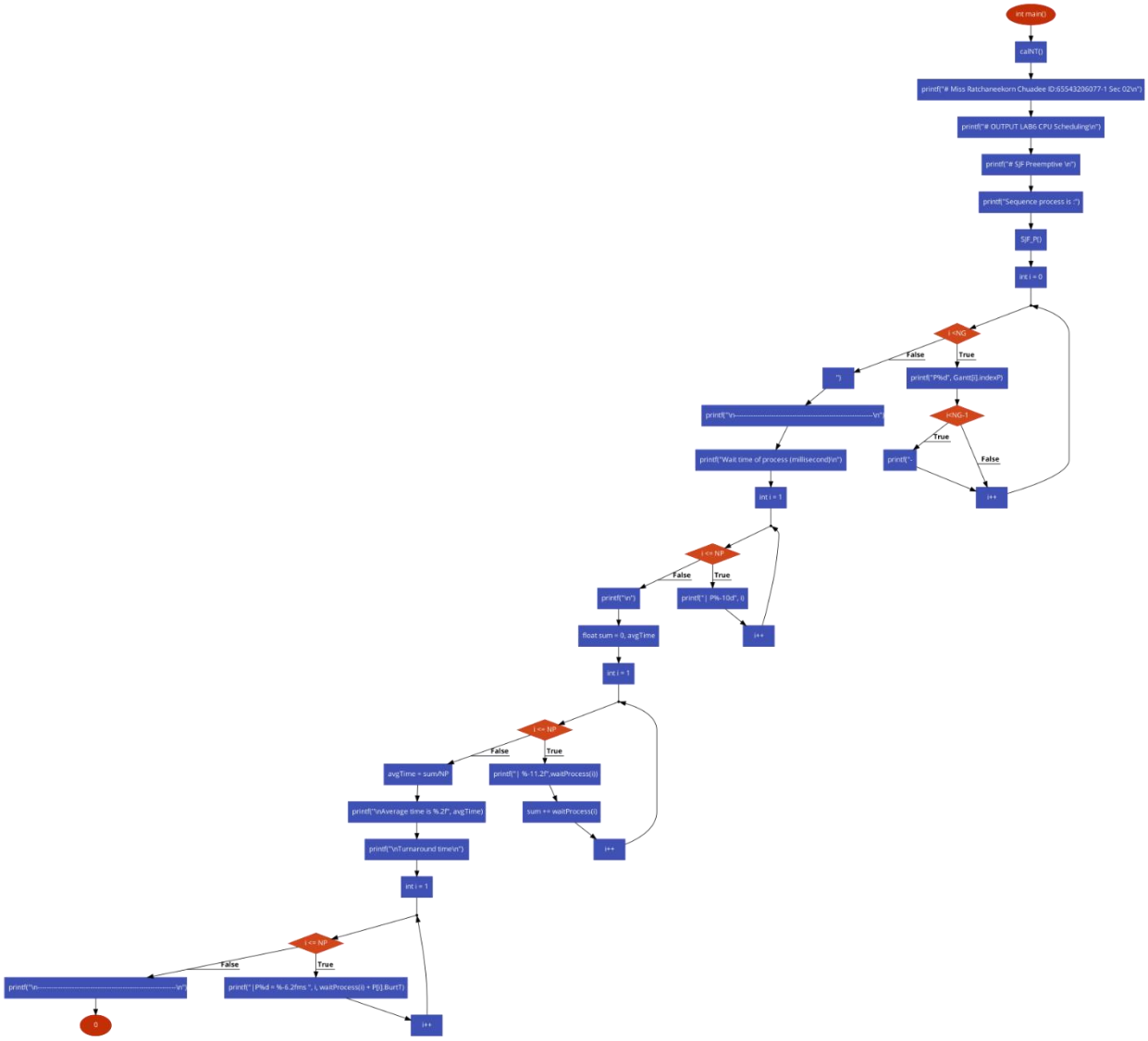

int minBurt(int indexP, int tempP
, int countP)
int minBurt = P[indexP].BurtT
int minPro = indexP
countP = 0
int i = 1
i <= NP
True
False
minPro
P[indexP].ArrivalT == P[i].ArrivalT
True
False
tempP[countP] = i
countP+=1
P[i].BurtT < minBurt
True
False
minBurt = P[i].BurtT
minPro = i
++i


void pushSynchronous(int indexP, int tempP
, int countP)
int i = 0
i < countP
False
True
i
tempP[i] != indexP
True
.BurtT)
push(tempP[i],P
False
tempP
void getData(int indexP, int time_i)
++i
Gantt[NG].indexP = indexP
Gantt[NG].startP = time_i
NG++

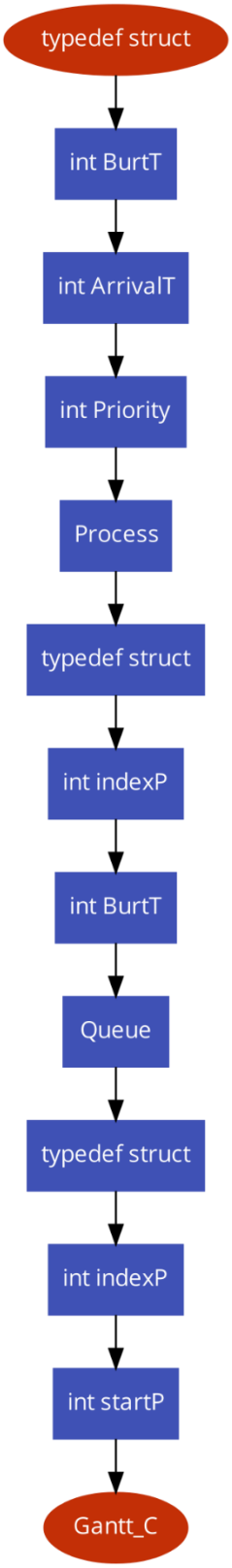

void SJF_P()
int indexP = 0, ENDPro = 0, runningP = 0
int tempP[NP], countP = 0
int i = 0
i <= NT
True
int j = 1
j <= NP
True
i == P[j].ArrivalT
False
True
++j
indexP = minBurt(j,tempP,&countP)
i >= ENDPro && SP == 0
True
False
pushSynchronous(indexP,tempP,countP)
P[indexP].BurtT < (P[runningP].BurtT-i) && i < ENDPro
False
True
ENDPro = i + P[indexP].BurtT
push(indexP, P[indexP].BurtT)
pushSynchronous(indexP,tempP,countP)
runningP = indexP
push(runningP, (P[runningP].BurtT - i))
pushSynchronous(indexP, tempP, countP)
ENDPro = i + P[indexP].BurtT
getData(indexP,i)
runningP = indexP
getData(indexP, i)
i >= ENDPro && SP != 0
True
False
Queue indexPQ
indexPQ = sortQ()
pop()
ENDPro = i + indexPQ.BurtT
runningP = indexPQ.indexP
getData(indexPQ.indexP,i)
++i
False

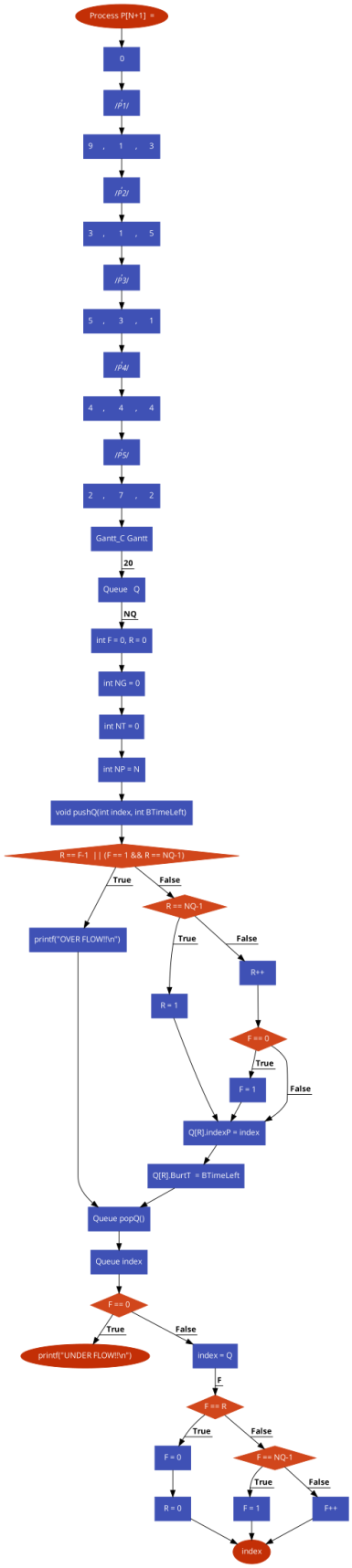

float waitProcess(int indexP)
int count = 0
float waitT =0, end = 0
int i = 0
i <NG
True
False
waitT
Gantt[i].indexP == indexP
True
False
count == 0
True
False
waitT += (float)Gantt[i].startP - P[indexP].ArrivalT
waitT += (float)Gantt[i].startP - end
end = (float)Gantt[i+1].startP
end = (float)Gantt[i+1].startP
count++
i++

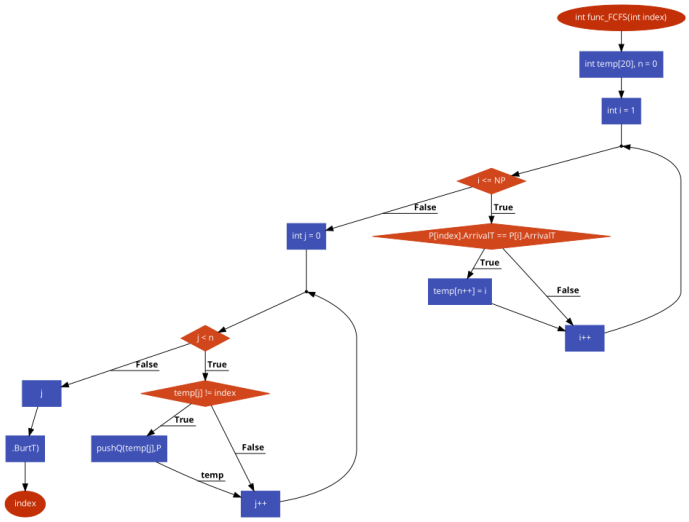

void calNT()
int sumBurt = 0
int minArrival = P[1].ArrivalT
int i = 1
i <= NP
True
False
P[i].ArrivalT < minArrival
NT = minArrival + sumBurt
True
minArrival = P[i].ArrivalT
False
sumBurt += P[i].BurtT
++i

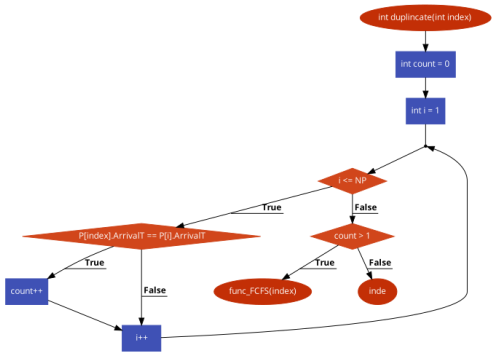

int main()
callNT()
printf("# Miss Ratchaneekorn Chuadee ID:65543206077-1 Sec 02\n")
printf("# OUTPUT LAB6 CPU Scheduling\n")
printf("# SJF Preemptive \n")
printf("Sequence process is :")
SJF_P()
int i = 0
i <NG
False
True
"]
printf("P%d", Gantt[i].indexP)
printf("\n----------\n")
i=NG-1
True
False
printf("-
i++
printf("Wait time of process (millisecond)\n")
int i = 1
i <= NP
False
True
printf("\n")
print(" | P%-10d", i)
i++
float sum = 0, avgTime
int i = 1
i <= NP
False
True
avgTime = sum/NP
printf(" | %-11.2f",waitProcess(i))
printf("\nAverage time is %.2f", avgTime)
sum += waitProcess(i)
printf("\nTurnaround time\n")
i++
int i = 1
i <= NP
False
True
printf("\n----------\n")
printf(" | P%d = %-6.2fms ", i, waitProcess(i) + P[i].Burt)
0
i++

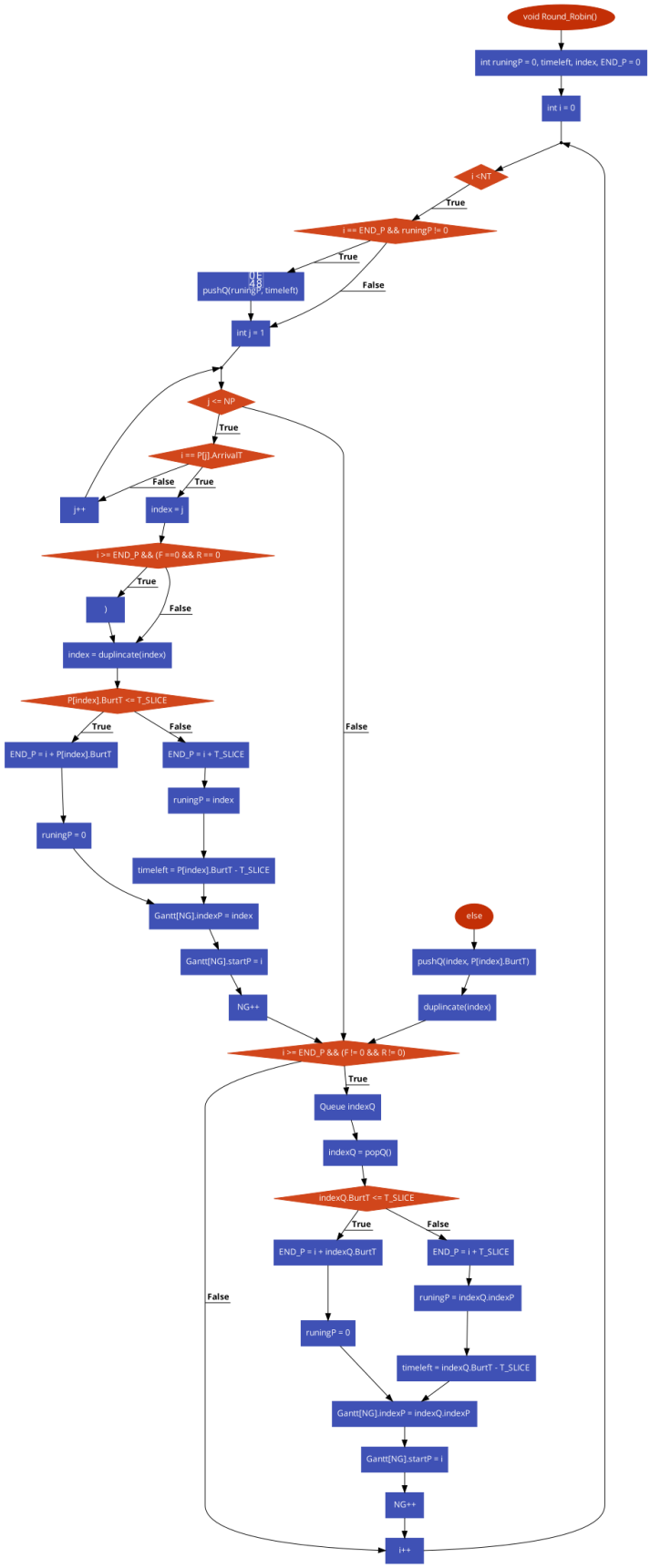
Round Robin scheduling (Time quantum = 4)

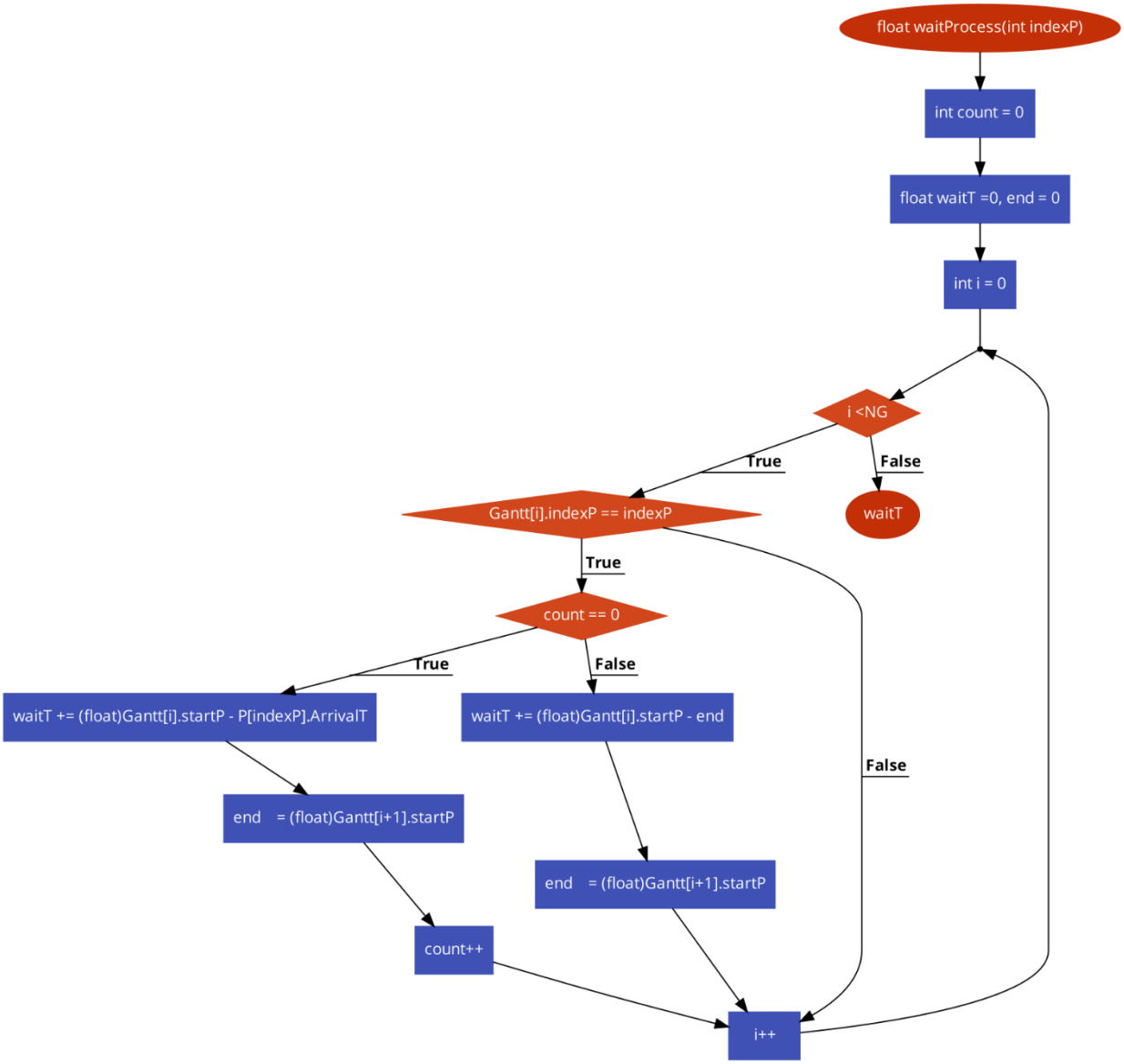


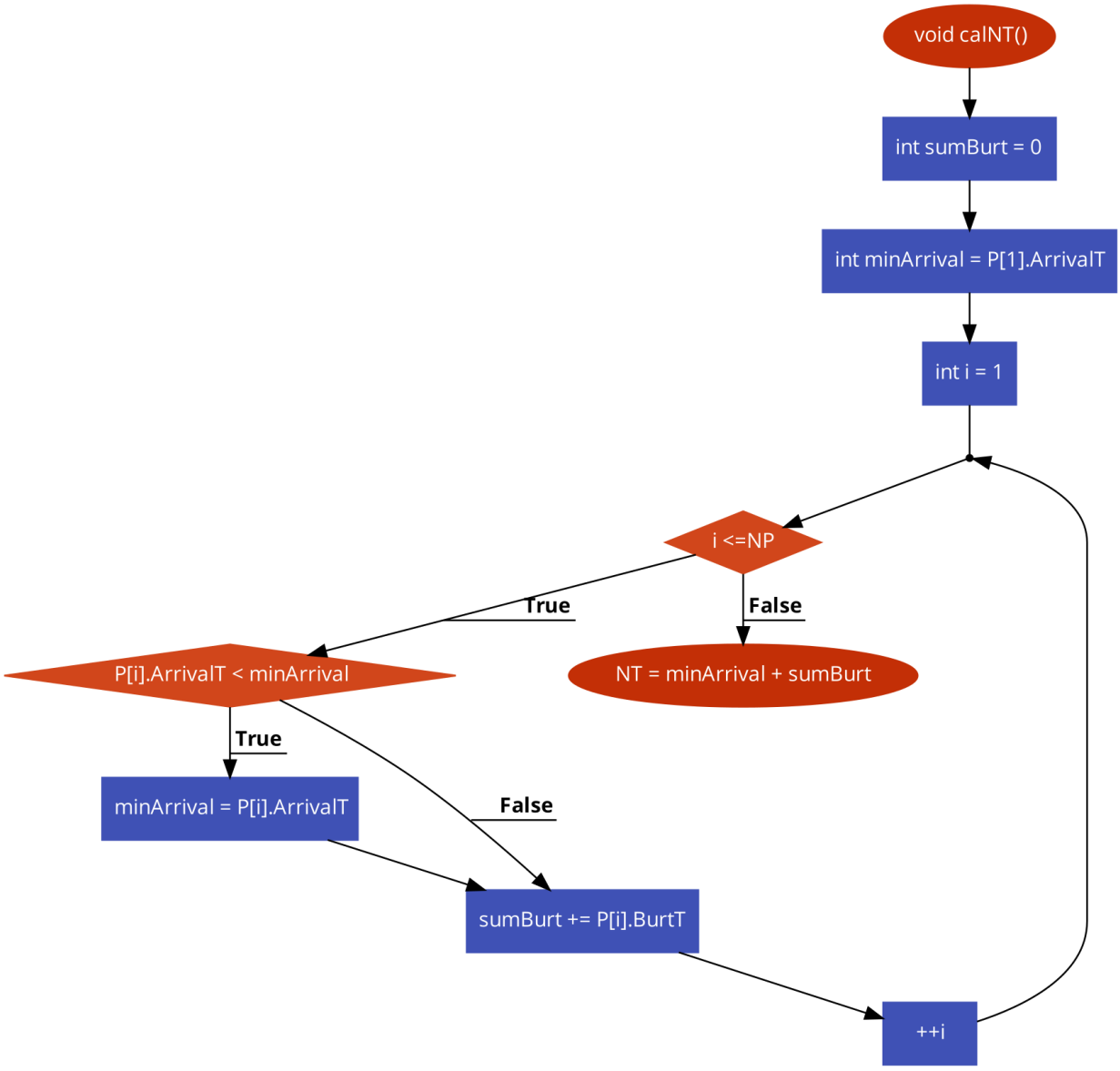

Process P[N+1] =
0
/P1/
9 , 1 , 3
/P2/
3 , 1 , 5
/P3/
5 , 3 , 1
/P4/
4 , 4 , 4
/P5/
2 , 7 , 2
Gantt_C Gantt
20
Queue Q
NQ
int F = 0, R = 0
int NG = 0
int NT = 0
int NP = N
void pushQ(int index, int BTimeLeft)
R == F-1 || (F == 1 && R == NQ-1)
True
False
printf("OVER FLOW!!\n")
R == NQ-1
True
False
R = 1
R++
F == 0
True
False
F = 1
Q[R].indexP = index
Q[R].BurtT = BTimeLeft
Queue popQ()
Queue index
F == 0
True
False
printf("UNDER FLOW!!\n")
index = Q
F
F == R
True
False
F = 0
F == NQ-1
True
False
R = 0
F = 1
F++
index

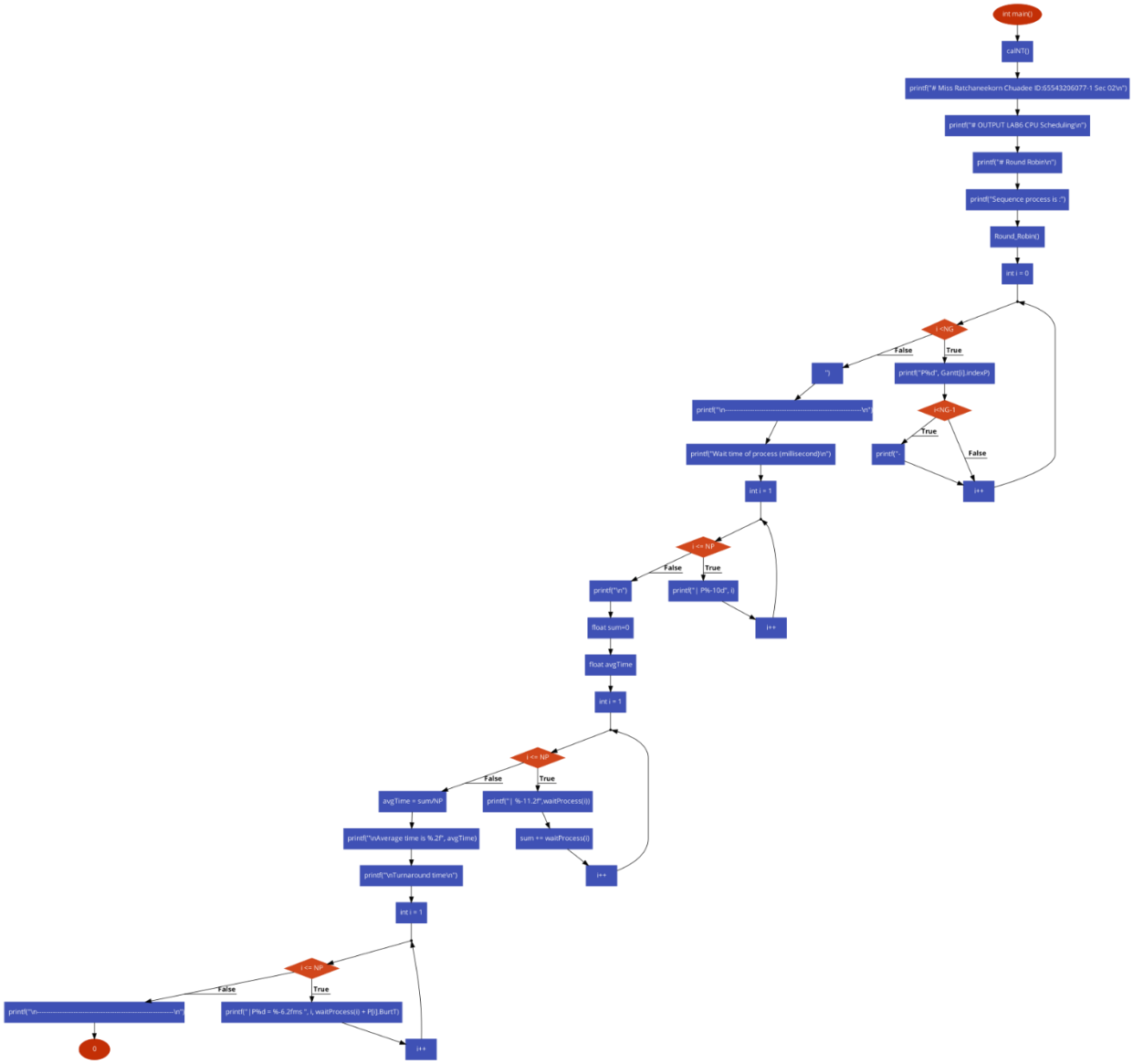

int func_FCFS(int index)
int temp[20], n = 0
int i = 1
i <= NP
False
True
int j = 0
P[index].ArrivalT == P[i].ArrivalT
True
temp[n++] = i
False
i++
j < n
False
True
j
temp[j] != index
True
.BurtT)
pushQ(temp[j],P
False
temp
index
j++


int duplincate(int index)
int count = 0
int i = 1
i <= NP
True
False
P[index].ArrivalT == P[i].ArrivalT
count > 1
True
count++
False
True
False
func_FCFS(index)
inde
i++


void Round_Robin()
int runingP = 0, timeleft, index, END_P = 0
int i = 0
i <NT
True
i == END_P && runingP != 0
True
False
pushQ(runingP, timeleft)
int j = 1
j <= NP
True
i == P[j].ArrivalT
False
True
j++
index = j
i >= END_P && (F ==0 && R == 0
True
)
False
index = duplincate(index)
P[index].BurtT <= T_SLICE
True
False
False
END_P = i + P[index].BurtT
END_P = i + T_SLICE
runingP = index
runingP = 0
timeleft = P[index].BurtT - T_SLICE
Gantt[NG].indexP = index
Gantt[NG].startP = i
NG++
else
pushQ(index, P[index].BurtT)
duplincate(index)
i >= END_P && (F != 0 && R != 0)
True
Queue indexQ
indexQ = popQ()
indexQ.BurtT <= T_SLICE
True
False
END_P = i + indexQ.BurtT
END_P = i + T_SLICE
runingP = indexQ.indexP
False
runingP = 0
timeleft = indexQ.BurtT - T_SLICE
Gantt[NG].indexP = indexQ.indexP
Gantt[NG].startP = i
NG++
i++

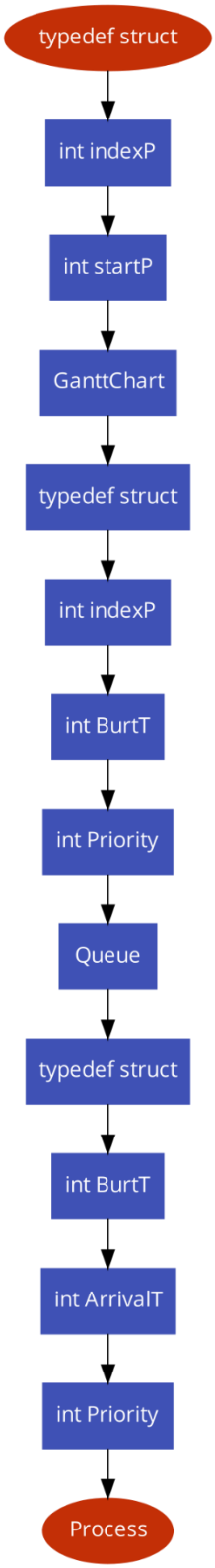

float waitProcess(int indexP)
int count = 0
float waitT =0, end = 0
int i = 0
i <NG
True
False
waitT
Gantt[i].indexP == indexP
True
False
count == 0
True
False
waitT += (float)Gantt[i].startP - P[indexP].ArrivalT
waitT += (float)Gantt[i].startP - end
end = (float)Gantt[i+1].startP
end = (float)Gantt[i+1].startP
count++
i++

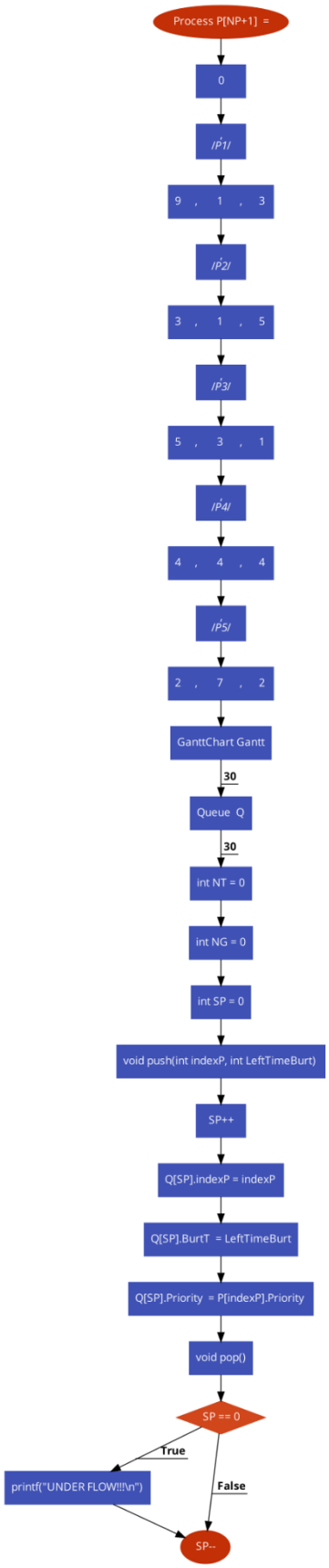

void calNT()
int sumBurt = 0
int minArrival = P[1].ArrivalT
int i = 1
i <=NP
True
False
P[i].ArrivalT < minArrival
NT = minArrival + sumBurt
True
minArrival = P[i].ArrivalT
False
sumBurt += P[i].BurtT
++i

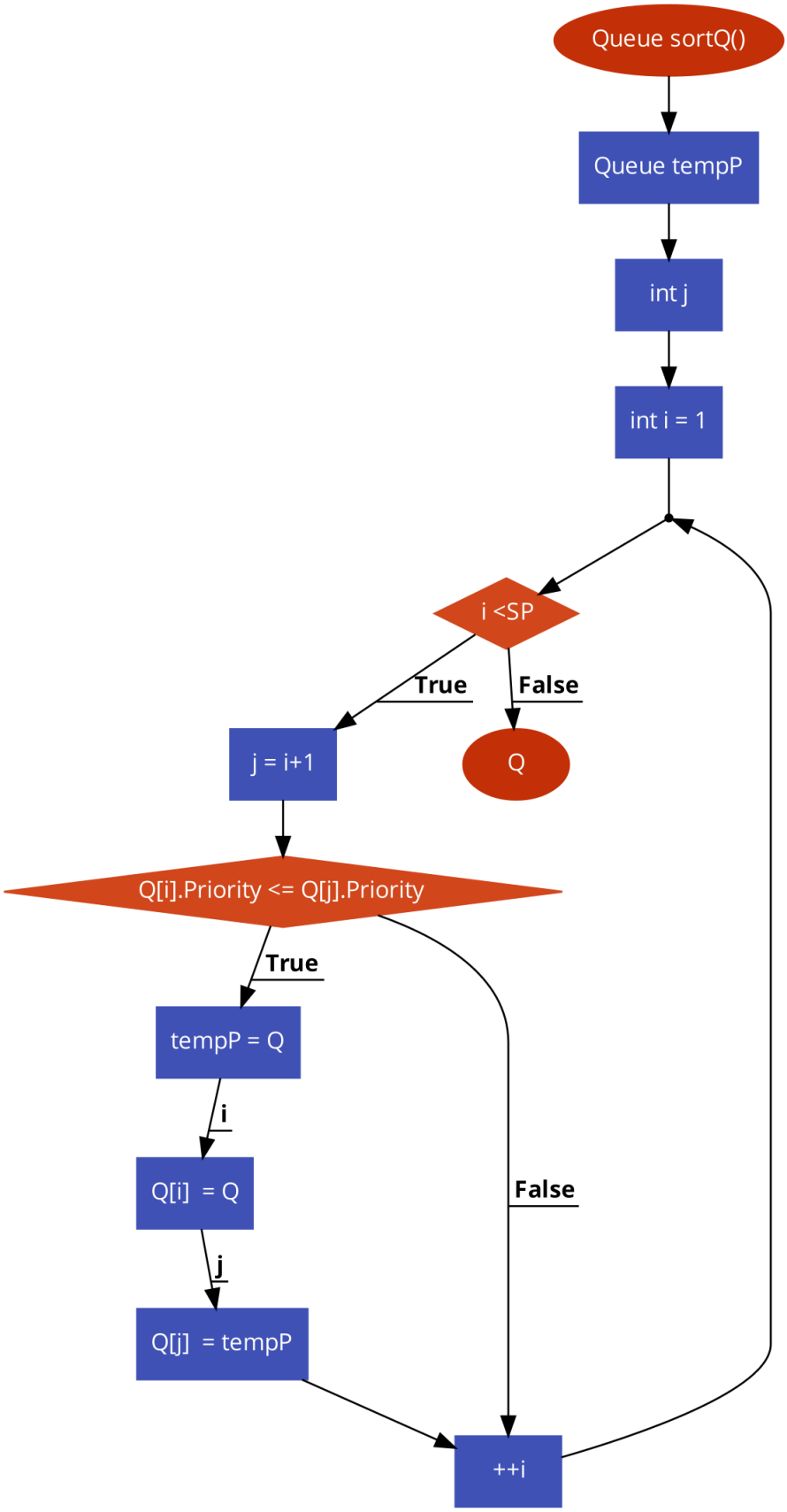

int main()
callINT()
printf("# Miss Ratchaneekorn Chuadee ID:65543206077-1 Sec 02\n")
printf("# OUTPUT LAB6 CPU Scheduling\n")
printf("# Round Robin\n")
printf("Sequence process is :")
Round_Robin()
int i = 0
i <NG
False
True
"}
printf("P%d", Gantt[i].indexP)
printf("\n----------\n")
i<NG-1
True
False
printf("Wait time of process (millisecond)\n")
int i = 1
i++
i <= NP
False
True
printf("\n")
printf("| P%-10d", i)
i++
float sum=0
float avgTime
int i = 1
i <= NP
False
True
avgTime = sum/NP
printf("| %-11.2f",waitProcess(i))
printf("\nAverage time is %.2f", avgTime)
sum += waitProcess(i)
printf("\nTurnaround time\n")
i++
int i = 1
i <= NP
False
True
printf("\n----------\n")
printf("| P%d = %-6.2fms ", i, waitProcess(i) + P[i].BurtT)
0
i++

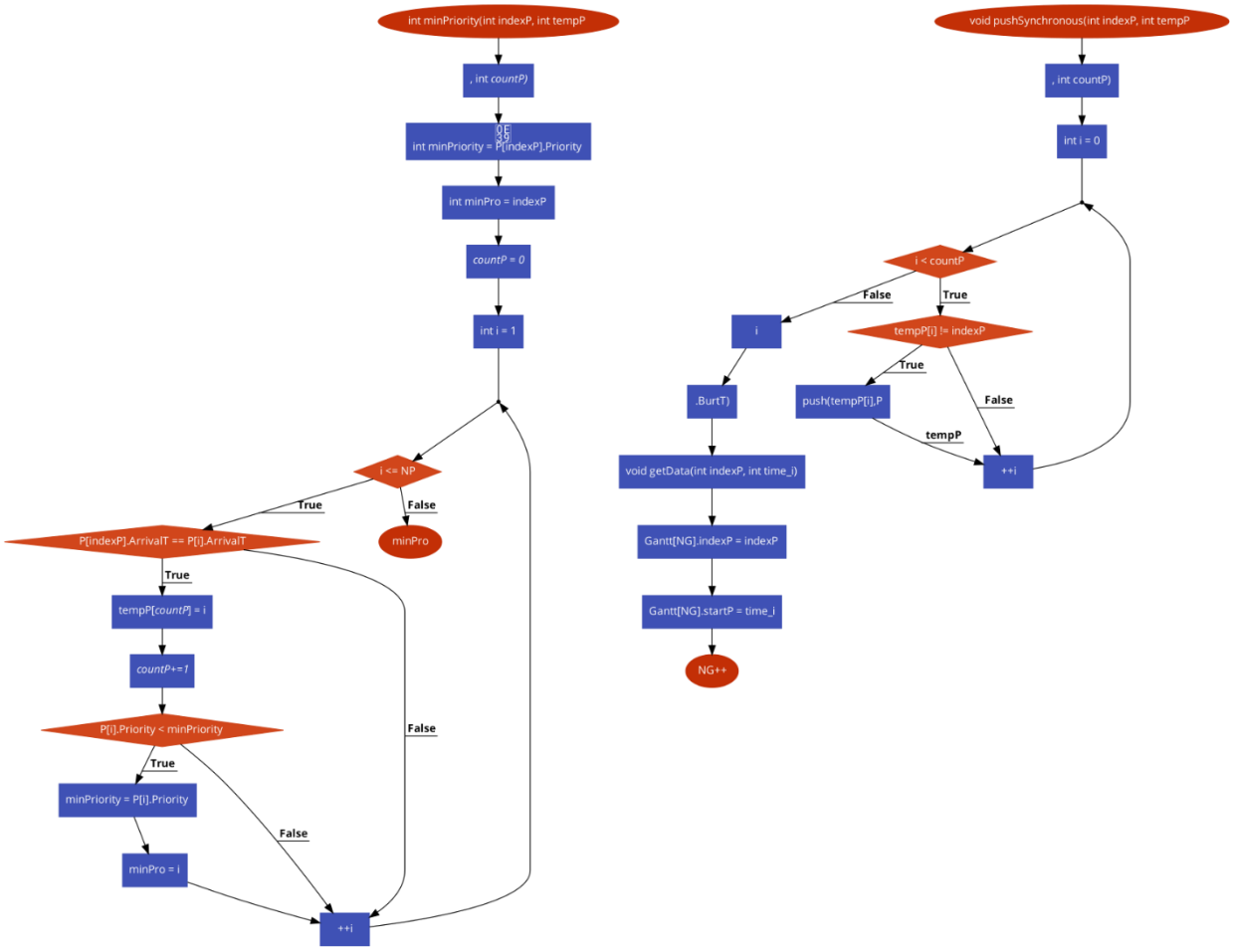
Priority scheduling

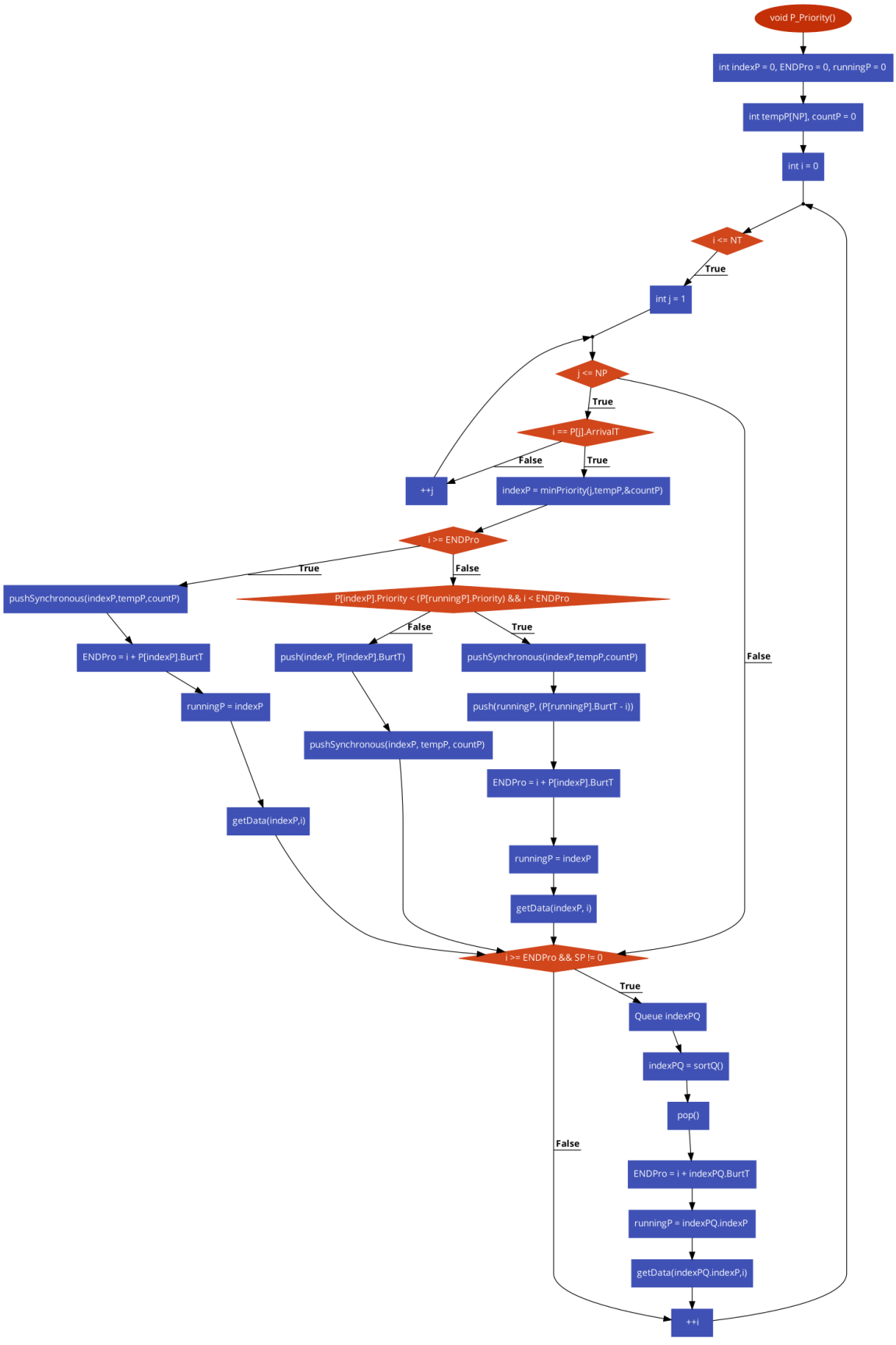
- Priority (SJF Preemptive)

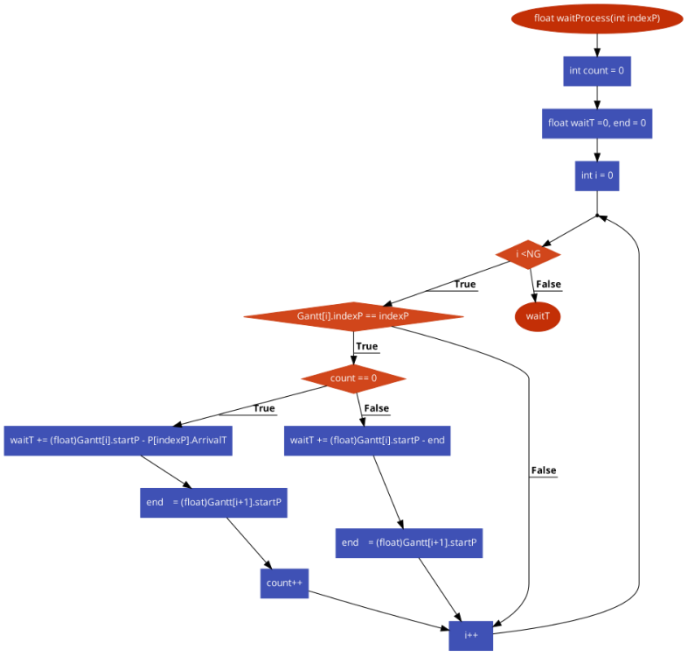


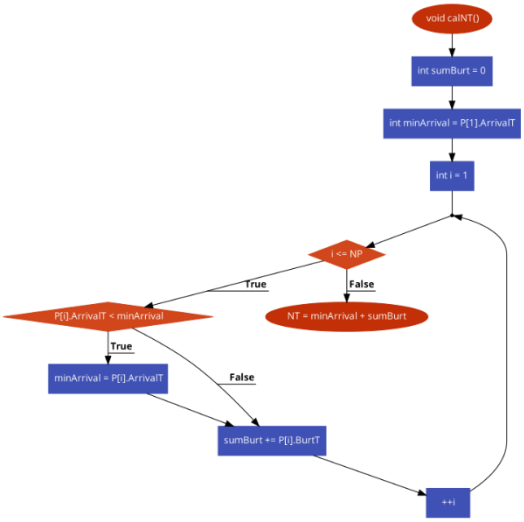

Process P[NP+1] =
0
/P1/
9 , 1 , 3
/P2/
3 , 1 , 5
/P3/
5 , 3 , 1
/P4/
4 , 4 , 4
/P5/
2 , 7 , 2
GanttChart Gantt
30
Queue Q
30
int NT = 0
int NG = 0
int SP = 0
void push(int indexP, int LeftTimeBurt)
SP++
Q[SP].indexP = indexP
Q[SP].BurtT = LeftTimeBurt
Q[SP].Priority = P[indexP].Priority
void pop()
SP == 0
True
False
printf("UNDER FLOW!!!\n")
SP--

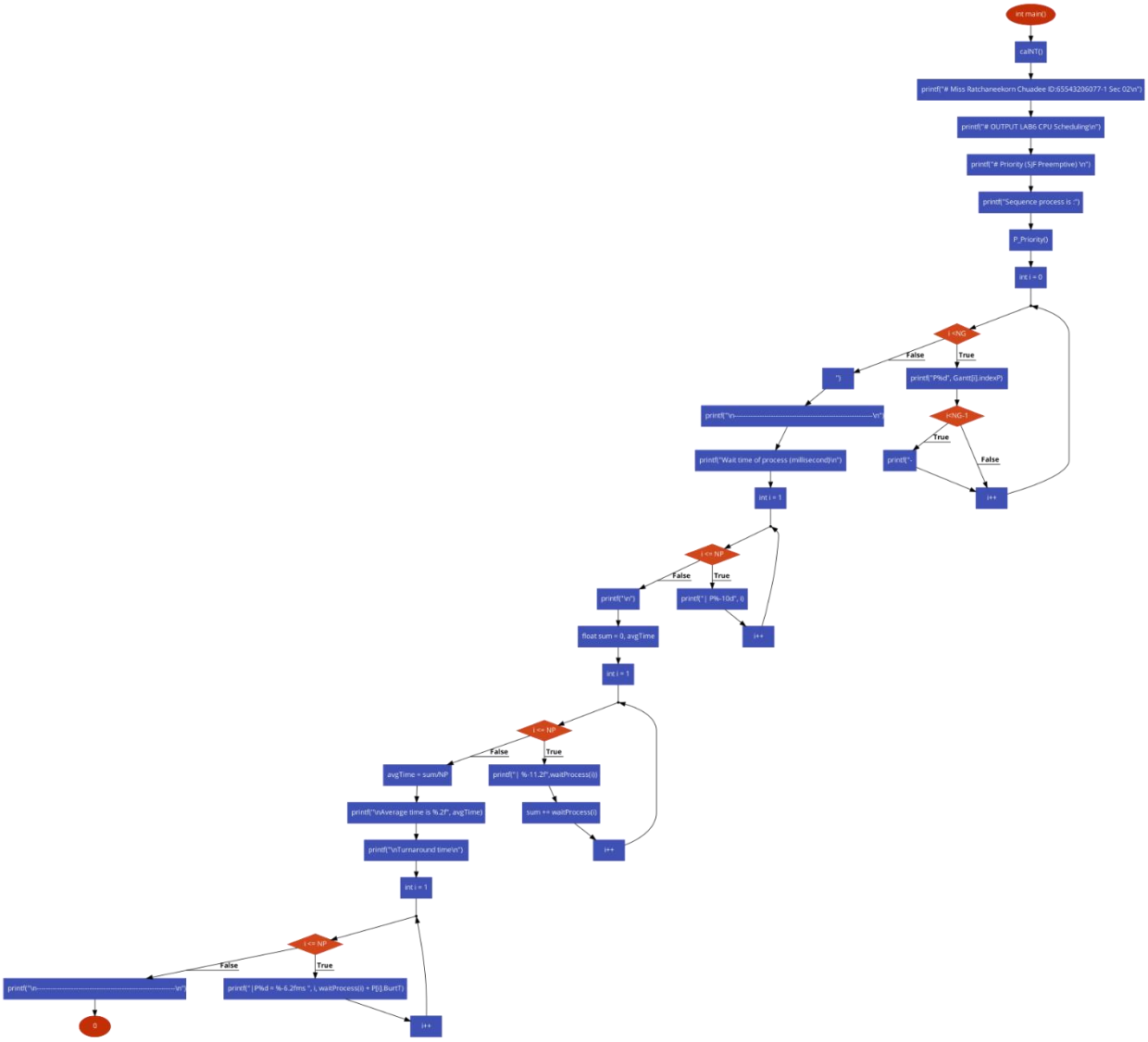

Queue sortQ()
Queue tempP
int j
int i = 1
i <SP
True
False
j = i+1
Q
Q[i].Priority <= Q[j].Priority
True
tempP = Q
i
Q[i] = Q
False
j
Q[j] = tempP
++i


int minPriority(int indexP, int tempP
, int countP)
int minPriority = P[indexP].Priority
int minPro = indexP
countP = 0
int i = 1
i <= NP
True
False
minPro
P[indexP].ArrivalT == P[i].ArrivalT
True
False
tempP[countP] = i
countP+=1
P[i].Priority < minPriority
True
False
minPriority = P[i].Priority
minPro = i
++i
void pushSynchronous(int indexP, int tempP
, int countP)
int i = 0
i < countP
False
True
i
tempP[i] != indexP
True
False
.BurtT)
push(tempP[i],P
tempP
++i
void getData(int indexP, int time_i)
Gantt[NG].indexP = indexP
Gantt[NG].startP = time_i
NG++


void P_Priority()
int indexP = 0, ENDPro = 0, runningP = 0
int tempP[NP], countP = 0
int i = 0
i <= NT
True
int j = 1
j <= NP
True
i == P[j].ArrivalT
False
True
++j
indexP = minPriority(j,tempP,&countP)
i >= ENDPro
True
False
pushSynchronous(indexP,tempP,countP)
P[indexP].Priority < (P[runningP].Priority) && i < ENDPro
False
True
ENDPro = i + P[indexP].BurtT
push(indexP, P[indexP].BurtT)
pushSynchronous(indexP,tempP,countP)
runningP = indexP
push(runningP, (P[runningP].BurtT - i))
pushSynchronous(indexP, tempP, countP)
ENDPro = i + P[indexP].BurtT
getData(indexP,i)
runningP = indexP
getData(indexP, i)
False
i >= ENDPro && SP != 0
True
Queue indexPQ
indexPQ = sortQ()
pop()
False
ENDPro = i + indexPQ.BurtT
runningP = indexPQ.indexP
getData(indexPQ.indexP,i)
++i


float waitProcess(int indexP)
int count = 0
float waitT =0, end = 0
int i = 0
i <NG
True
False
waitT
Gantt[i].indexP == indexP
True
False
count == 0
True
False
waitT += (float)Gantt[i].startP - P[indexP].ArrivalT
waitT += (float)Gantt[i].startP - end
end = (float)Gantt[i+1].startP
end = (float)Gantt[i+1].startP
count++
i++


void calNT()
int sumBurt = 0
int minArrival = P[1].ArrivalT
int i = 1
i <= NP
True
False
NT = minArrival + sumBurt
P[i].ArrivalT < minArrival
True
False
minArrival = P[i].ArrivalT
sumBurt += P[i].BurtT
++i


int main()
callNT()
printf("# Miss Ratchaneekorn Chuadee ID:65543206077-1 Sec 02\n")
printf("# OUTPUT LAB6 CPU Scheduling\n")
printf("# Priority (SJF Preemptive) \n")
printf("Sequence process is :")
P_Priority()
int i = 0
i <NG
False
True
"]
printf("P%d", Gantt[i].indexP)
printf("\n----------\n")
i<NG-1
True
False
printf("Wait time of process (millisecond)\n")
printf("-
i++
int i = 1
i <= NP
False
True
printf("\n")
printf("| P%-10d", i)
i++
float sum = 0, avgTime
int i = 1
i <= NP
False
True
avgTime = sum/NP
printf("| %-11.2f",waitProcess[i])
printf("\nAverage time is %.2f", avgTime)
sum += waitProcess[i]
printf("\nTurnaround time\n")
i++
int i = 1
i <= NP
False
True
printf("\n----------\n")
printf("| P%d = %-6.2fms ", i, waitProcess[i] + P[i].BurstT)
0
i++

อธิบาย Code

Non preemptive SJF scheduling

```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]

```

โค้ดเริ่มต้นด้วยการประกาศตัวแปรดังนี้

- NP เป็นจำนวนกระบวนการ
- GanttChart เป็นโครงสร้างข้อมูลสำหรับเก็บลำดับการทำงานของกระบวนการ
- Queue เป็นโครงสร้างข้อมูลสำหรับเก็บกระบวนการที่มีเวลารอนานที่สุด
- Process เป็นโครงสร้างข้อมูลสำหรับเก็บข้อมูลเกี่ยวกับกระบวนการแต่ละกระบวนการ

จากนั้นโค้ดจะกำหนดค่าเริ่มต้นให้กับตัวแปร P[NP+1] ซึ่งเป็นอาร์เรย์ของโครงสร้างข้อมูล Process โดยกำหนดให้ทุกกระบวนการมีค่า BurtT เป็น 0 และ ArrivalT เป็น 1

หลังจากนั้นโค้ดจะเข้าสู่ลูปหลัก โดยลูปนี้จะวนซ้ำจนกว่าทุกกระบวนการจะเสร็จสิ้น

ภายในลูปหลัก โค้ดจะดำเนินการดังนี้

1. ตรวจสอบว่ามีกระบวนการใดเข้ามาใหม่หรือไม่ โดยดูจาก ArrivalT ของกระบวนการใน P[0] หาก ArrivalT ของกระบวนการใน P[0] เท่ากับ 0 แสดงว่ามีกระบวนการใหม่เข้ามา
2. หากมีกระบวนการใหม่เข้ามา โค้ดจะเพิ่มกระบวนการนั้นลงในคิว Queue
3. เลือกกระบวนการที่มีเวลารอนานที่สุดจากคิว Queue
4. เริ่มต้นกระบวนการที่เลือก
5. ลดเวลารอของกระบวนการอื่น ๆ ในคิว Queue ลง 1

```
26  GanttChart Gantt[30];
27  Queue  Q[30];
28  int NT = 0;     //Time
29  int NG = 0;     //number GanttChart
30  int SP = 0;     //ตัวชี้ค่าในคิว
31
32  void push(int indexP){        //เก็บโปรเซสไว้ในคิว
33      SP++;
34      Q[SP].indexP = indexP;
35      Q[SP].BurtT  = P[indexP].BurtT;
36  }
37  void pop(){                   //นำโปรเซสออกจากคิว
38      if(SP == 0)
39          printf("UNDER FLOW!!!\n");
40      SP--;
41  }
```

ตัวแปร:

- GanttChart Gantt[30]: ตาราง Gantt สำหรับเก็บลำดับการทำงานของกระบวนการ
- Queue Q[30]: คิวสำหรับเก็บกระบวนการที่รอทำงาน
- int NT = 0;: เวลาปัจจุบัน
- int NG = 0;: จำนวนจุดในตาราง Gantt
- int SP = 0;: ตัวชี้ค่าในคิว

ฟังก์ชัน push():

- เพิ่มกระบวนการที่มีหมายเลข indexP ลงในคิว
- เพิ่มค่า SP ขึ้น 1
- เก็บหมายเลขของกระบวนการใน Q[SP].indexP
- เก็บเวลาทำงานของกระบวนการใน Q[SP].BurtT

ฟังก์ชัน pop():

- ลดค่า SP ลง 1
- หาก SP == 0 แสดงว่าคิวว่างเปล่า แสดงข้อความแจ้งเตือน

```
43  Queue sortQ(){                  //เรียงข้อมูลจากน้อยไปมาก
44      Queue tempP;
45      int j;
46      for (int i = 1; i <SP ; ++i) {
47          j = i+1;
48          if(Q[i].BurtT <= Q[j].BurtT){
49              tempP = Q[i];
50              Q[i]  = Q[j];
51              Q[j]  = tempP;
52          }
53      }
54      return Q[SP];
55  }
```

ฟังก์ชัน sortQ():

- เรียงกระบวนการในคิว Q จากน้อยไปมากตามเวลาทำงาน
- ใช้การวนลูปแบบบับเบิ้ลโซร์ต

การทำงานของฟังก์ชัน:

- วนลูปจาก i = 1 ถึง SP - 1
- กำหนด j = i + 1
- เปรียบเทียบเวลาทำงานของกระบวนการที่ Q[i] กับ Q[j]
- หาก Q[i].BurtT <= Q[j].BurtT แสดงว่ากระบวนการที่ Q[i] มีเวลาทำงานน้อยกว่ากระบวนการที่ Q[j]
- ทำการสลับตำแหน่งของกระบวนการที่ Q[i] กับ Q[j]

```
57  int minBurt(int indexP, int tempP[], int *countP) {  //ในกรณีที่มีโปรเซสเกิดในเวลาเดียวกัน หาค่าโปรเซสที่ใช้เวลาทำงานน้อยที่สุ
58      int minBurt = P[indexP].BurtT;
59      int minPro = indexP;
60      *countP = 0;
61      for (int i = 1; i <= NP; ++i)
62          if (P[indexP].ArrivalT == P[i].ArrivalT) {
63              tempP[*countP] = i;                    //เก็บโปรเซสที่เกิดในเวลาเดียวกัน
64              *countP+=1;                            //จำนวนโปรเซสที่เกิดในเวลาเดียวกัน
65              if(P[i].BurtT < minBurt ){
66                  minBurt = P[i].BurtT;
67                  minPro = i;
68              }
69          }
70      return minPro;                                 //โปรเซสที่ใช้เวลาทำงานน้อยที่สุด
71  }
```

ฟังก์ชัน minBurt():

- หากระบวนการที่ใช้เวลาทำงานน้อยที่สุดจากกลุ่มกระบวนการที่เกิดในเวลาเดียวกัน

การทำงานของฟังก์ชัน:

1. กำหนด minBurt = P[indexP].BurtT และ minPro = indexP
2. วนลูปจาก i = 1 ถึง NP

3. หาก P[i].ArrivalT == P[indexP].ArrivalT แสดงว่ากระบวนการ i เกิดในเวลาเดียวกันกับกระบวนการ indexP
4. เก็บหมายเลขของกระบวนการ i ลงใน tempP[*countP]
5. เพิ่มค่า *countP ขึ้น 1
6. หาก P[i].BurtT < minBurt แสดงว่ากระบวนการ i มีเวลาทำงานน้อยกว่ากระบวนการ minPro
7. กำหนด minBurt = P[i].BurtT และ minPro = i
8. คืนค่า minPro ไปยังฟังก์ชันที่เรียกใช้

```
73  void pushSynchronous(int indexP, int tempP[], int countP) {//เก็บโปรเซสที่เกิดพร้อมกันไว้ในคิว(ในเวลาเดียวกัน)
74      for (int i = 0; i < countP ; ++i)
75          if(tempP[i] != indexP )                          //ยกเว้นโปรเซสที่ใช้เวลาทำงานน้อยที่สุด
76              push(tempP[i]);
77
78  }
79  void getData(int indexP, int time_i){
80      Gantt[NG].indexP = indexP;          //เก็บโปรเซสเพื่อทำ GanttChart
81      Gantt[NG].startP = time_i;          //เก็บเวลาเริ่มทำงานโปรเซสเพื่อทำ GanttChart
82      NG++;
83  }
```

ฟังก์ชัน pushSynchronous():

- เก็บกระบวนการที่เกิดในเวลาเดียวกันไว้ในคิว

การทำงานของฟังก์ชัน:

1. วนลูปจาก i = 0 ถึง countP - 1
2. หาก tempP[i] != indexP แสดงว่ากระบวนการ tempP[i] ไม่ใช่กระบวนการที่ใช้เวลาทำงานน้อยที่สุด
3. เพิ่มกระบวนการ tempP[i] ลงในคิว

ฟังก์ชัน getData():

- เก็บข้อมูลกระบวนการเพื่อทำตาราง Gantt

การทำงานของฟังก์ชัน:

1. เก็บหมายเลขของกระบวนการลงใน Gantt[NG].indexP
2. เก็บเวลาเริ่มทำงานของกระบวนการลงใน Gantt[NG].startP
3. เพิ่มค่า NG ขึ้น 1

```
85  void SJF_NP(){
86      int indexP = 0, ENDPro = 0;
87      int tempP[NP], countP = 0;
88      for (int i = 0; i <= NT; ++i) {              //i แทนเวลา (Time)
89          for (int j = 1; j <= NP; ++j) {
90              if(i == P[j].ArrivalT){              //ณ เวลาที่ i มีโปรเซส 1 2 3 ...N เกิดขึ้นใหม่
91                  indexP = minBurt(j,tempP,&countP);//ถ้ามีโปรเซสเกิดในเวลาเดียวกันให้หาโปรเซสที่ใช้เวลาทำงานน้อยที่สุด
92                  if(i >= ENDPro && SP == 0){      //ณ เวลาที่ i ถ้าไม่มีโปรเซสลำลังทำงาน(ไม่มีการใช้ทรัพยากร) และ ไม่มีโปรเ
93                      pushSynchronous(indexP,tempP,countP);//เก็บโปรเซสที่เหลือไว้ในคิวถ้ามีโปรเซสมีกิดขึ้นพร้อกัน(ในเวลาเดียวกัน)
94                      ENDPro = i + P[indexP].BurtT;
95                      getData(indexP,i);           //เก็บข้อมูลโปรเซส
96                  }else{                           //ณ เวลาที่ i ถ้ามีโปรเซสลำลังทำงาน(มีการใช้ทรัพยากร) หรือ มีโปรเซสออยู่ใ
97                      push(indexP);                //เก็บค่าโปรเซสที่เกิดใหม่ ณ เวลาที่ i ไว้ในคิว
98                      pushSynchronous(indexP,tempP,countP);//เก็บโปรเซสที่เหลือไว้ในคิวถ้ามีโปรเซสมีกิดขึ้นพร้อกัน(ในเวลาเดียวกัน)
99                  }
100                 break;
101             }
102         }
103         if (i >= ENDPro && SP != 0) {            //ณ เวลาที่ i ถ้าไม่มีโปรเซสไหนทำงาน(ทรัพยากรว่าง) แต่ ยังมีโปรเซสเหลืออยู่
104             Queue indexPQ;
105             indexPQ = sortQ();                    //เรียงโปรเซสในคิว โดยดูจากโปรเซสที่ใช้เวลาการทำงานน้อยที่สุด
106             pop();                               //นำโปรเซสออกจากคิว
107             ENDPro = i + indexPQ.BurtT;
108             getData(indexPQ.indexP,i);
109         }
110     }
111 }
```

ฟังก์ชัน SJF_NP():

- กำหนดลำดับการทำงานของกระบวนการโดยใช้อัลกอริทึม Priority (SJF Non Preemptive) สำหรับกระบวนการแบบหลาย (NP)

การทำงานของฟังก์ชัน:

1. วนลูปจาก i = 0 ถึง NT
2. วนลูปจาก j = 1 ถึง NP
3. หาก i == P[j].ArrivalT แสดงว่ากระบวนการ j เกิดขึ้นใหม่
4. เรียกใช้ฟังก์ชัน minBurt() เพื่อหากระบวนการที่ใช้เวลาทำงานน้อยที่สุดจากกลุ่มกระบวนการที่เกิดในเวลาเดียวกัน
5. หาก i >= ENDPro && SP == 0 แสดงว่าไม่มีกระบวนการใดทำงานอยู่และไม่มีกระบวนการใดรออยู่ในคิว
6. เรียกใช้ฟังก์ชัน pushSynchronous() เพื่อเก็บกระบวนการที่เกิดในเวลาเดียวกันไว้ในคิว
7. ตั้งค่า ENDPro เป็น i + P[indexP].BurtT
8. เรียกใช้ฟังก์ชัน getData() เพื่อเก็บข้อมูลกระบวนการ
9. หาก i >= ENDPro && SP != 0 แสดงว่าไม่มีกระบวนการใดทำงานอยู่แต่มีกระบวนการรออยู่ในคิว
10. เรียกใช้ฟังก์ชัน sortQ() เพื่อเรียงกระบวนการในคิว โดยดูจากโปรเซสที่ใช้เวลาการทำงานน้อยที่สุด
11. เรียกใช้ฟังก์ชัน pop() เพื่อนำกระบวนการออกจากคิว

12. ตั้งค่า ENDPro เป็น i + Q[0].BurtT
13. เรียกใช้ฟังก์ชัน getData() เพื่อเก็บข้อมูลกระบวนการ

```
113  float waitProcess(int indexP){                              // คำนวณหา เวลารอของโปรเซสที่ i
114      int count = 0;
115      float waitT =0, end = 0;
116      for (int i = 0; i <NG ; i++) {
117          if(Gantt[i].indexP == indexP){
118              if(count == 0){                                 // เวลาโปรเซสได้เข้าทำงาน - เวลาเกิดของโปรเซส
119                  waitT += (float)Gantt[i].startP - P[indexP].ArrivalT;
120                  end    = (float)Gantt[i+1].startP;
121                  count++;
122              }else{
123                  waitT += (float)Gantt[i].startP - end;       // เวล่ที่โปรเซสได้เข้าทำงานอีกครั้ง - เวลาที่โปรเซสจบการทำงานค
124                  end    = (float)Gantt[i+1].startP;
125              }
126          }
127      }
128      return waitT;
129  }
```

ฟังก์ชัน waitProcess():

- คำนวณหาเวลารอของกระบวนการที่ indexP

การทำงานของฟังก์ชัน:

1. วนลูปจาก i = 0 ถึง NG
2. หาก Gantt[i].indexP == indexP แสดงว่าจุด i บนตาราง Gantt เป็นของกระบวนการ indexP
3. หาก count == 0 แสดงว่ากระบวนการ indexP กำลังทำงานเป็นครั้งแรก
4. คำนวณเวลารอของกระบวนการ indexP โดยเอาเวลาเริ่มทำงานของกระบวนการ indexP ลบด้วยเวลาเกิดของกระบวนการ indexP
5. กำหนดค่า end เป็นเวลาเริ่มทำงานของจุดถัดไปบนตาราง Gantt
6. เพิ่มค่า count ขึ้น 1
7. หาก count > 0 แสดงว่ากระบวนการ indexP กำลังทำงานอีกครั้ง
8. คำนวณเวลารอของกระบวนการ indexP โดยเอาเวลาเริ่มทำงานของกระบวนการ indexP ลบด้วยเวลาจบการทำงานครั้งก่อน
9. กำหนดค่า end เป็นเวลาเริ่มทำงานของจุดถัดไปบนตาราง Gantt
10. คืนค่าเวลารอของกระบวนการ indexP

```
131  void calNT(){                                          //คำนวณหาผลรวมของ burt time
132      int sumBurt = 0;
133      int minArrival = P[1].ArrivalT;
134      for (int i = 1; i <= NP; ++i) {
135          if(P[i].ArrivalT < minArrival){
136              minArrival = P[i].ArrivalT;
137          }
138          sumBurt += P[i].BurtT;
139      }
140      NT = minArrival + sumBurt;                         //เวลาที่เริ่มเกิดโปรเซสตัวแรก + ผลรวมเวลาที่ใช้ในการทำงานของโปร
141  }
```

ฟังก์ชัน calNT():

- คำนวณหาค่า NT ซึ่งเป็นเวลาที่กระบวนการทั้งหมดทำงานเสร็จ

การทำงานของฟังก์ชัน:

1. กำหนดค่า sumBurt เป็น 0
2. กำหนดค่า minArrival เป็นเวลาเกิดของกระบวนการแรก
3. วนลูปจาก i = 1 ถึง NP
4. หาก P[i].ArrivalT < minArrival แสดงว่ากระบวนการ i เกิดก่อนกระบวนการแรก
5. กำหนดค่า minArrival เป็นเวลาเกิดของกระบวนการ i
6. เพิ่มค่า sumBurt ด้วยเวลาทำงานของกระบวนการ i
7. กำหนดค่า NT เป็นเวลาเกิดของกระบวนการแรกบวกกับเวลาทำงานของกระบวนการทั้งหมด

```c
int main(){
    calNT();
    printf("# Miss Ratchaneekorn Chuadee ID:65543206077-1 Sec 02\n");
    printf("# OUTPUT LAB6 CPU Scheduling\n");
    printf("# SJF Non Preemptive \n");
    printf("Sequence process is :");
    SJF_NP();
    for (int 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 (int i = 1; i <= NP; i++) {
        printf("| P%-10d", i);
    }
    printf("\n");
    float sum = 0, avgTime;
    for (int 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 (int i = 1; i <= NP; i++) {
        printf("|P%d = %-6.2fms ", i, waitProcess(i) + P[i].BurtT);
    }
    printf("\n-----------------------------------------------------------------\n");
    return 0;
}
```

ฟังก์ชัน main():

- ฟังก์ชันหลักของโปรแกรม

การทำงานของฟังก์ชัน:

1. เรียกใช้ฟังก์ชัน calNT() เพื่อคำนวณหาค่า NT
2. พิมพ์ข้อมูลผู้เขียนและชื่อโปรแกรม
3. เรียกใช้ฟังก์ชัน SJF_NP() เพื่อกำหนดลำดับการทำงานของกระบวนการ
4. พิมพ์ลำดับการทำงานของกระบวนการ
5. พิมพ์ตาราง Gantt
6. พิมพ์เวลารอของกระบวนการแต่ละกระบวนการ
7. คำนวณหาเวลารอเฉลี่ย
8. พิมพ์เวลาตอบกลับของกระบวนการแต่ละกระบวนการ

ผลลัพธ์

```
# Miss Ratchaneekorn Chuadee ID:65543206077-1 Sec 02
# 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
-----------------------------------------------------------
```

ลำดับการทำงานของกระบวนการ:

- P2 -> P4 -> P5 -> P3 -> P1

เวลารอของกระบวนการ:

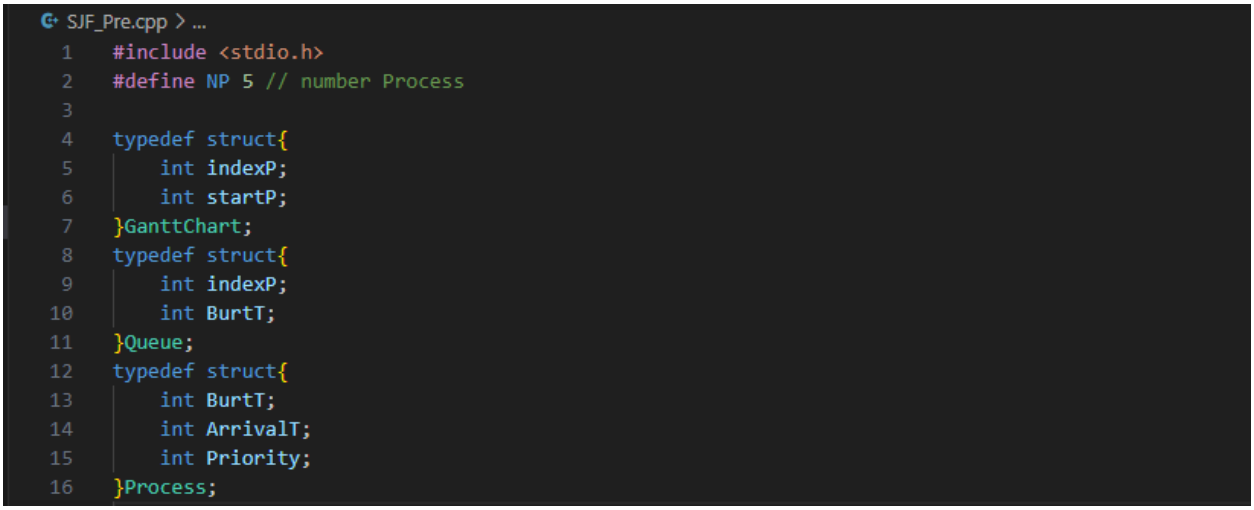
- P1: 14.00 ms
- P2: 0.00 ms
- P3: 7.00 ms
- P4: 0.00 ms
- P5: 1.00 ms

เวลารอเฉลี่ย: 4.40 ms

เวลาตอบกลับของกระบวนการ:

- P1: 23.00 ms
- P2: 3.00 ms
- P3: 12.00 ms
- P4: 4.00 ms
- P5: 3.00 ms

Preemptive SJF scheduling



```
SJF_Pre.cpp > ...
1   #include <stdio.h>
2   #define NP 5 // number Process
3
4   typedef struct{
5       int indexP;
6       int startP;
7   }GanttChart;
8   typedef struct{
9       int indexP;
10      int BurtT;
11  }Queue;
12  typedef struct{
13      int BurtT;
14      int ArrivalT;
15      int Priority;
16  }Process;
```

กำหนดโครงสร้างข้อมูลสำหรับใช้ในการทำงานของระบบปฏิบัติการแบบ FCFS (First Come First Serve) โดยโครงสร้างข้อมูลมีดังนี้

GanttChart เป็นโครงสร้างข้อมูลสำหรับแสดงแผนภูมิ Gantt ของกระบวนการทำงาน โดยแต่ละองค์ประกอบของโครงสร้างข้อมูลจะเก็บข้อมูลดังนี้

- indexP: ลำดับของกระบวนการ
- startP: เวลาเริ่มต้นของกระบวนการ

Queue เป็นโครงสร้างข้อมูลสำหรับใช้จัดลำดับกระบวนการทำงาน โดยแต่ละองค์ประกอบของโครงสร้างข้อมูลจะเก็บข้อมูลดังนี้

- indexP: ลำดับของกระบวนการ
- BurtT: เวลาทำงานของกระบวนการ

Process เป็นโครงสร้างข้อมูลสำหรับเก็บข้อมูลเกี่ยวกับกระบวนการทำงาน โดยแต่ละองค์ประกอบของโครงสร้างข้อมูลจะเก็บข้อมูลดังนี้

- BurtT: เวลาทำงานของกระบวนการ
- ArrivalT: เวลามาถึงของกระบวนการ
- Priority: ลำดับความสำคัญ

โค้ดส่วนนี้กำหนดค่าคงที่ NP เป็นจำนวนกระบวนการที่ต้องการทำงาน ซึ่งในที่นี้คือ 5 กระบวนการ จากนั้นโค้ดจะกำหนดโครงสร้างข้อมูล GanttChart และ Queue สำหรับใช้งาน โดย GanttChart จะมีขนาดเท่ากับ NP และ Queue จะมีขนาดเท่ากับ NP

สุดท้ายโค้ดจะกำหนดโครงสร้างข้อมูล Process สำหรับใช้งาน โดย Process จะมีขนาดเท่ากับ NP

การทำงานของโค้ดส่วนนี้สามารถอธิบายได้ดังนี้

โครงสร้างข้อมูล GanttChart จะใช้แสดงแผนภูมิ Gantt ของกระบวนการทำงาน โดยแต่ละองค์ประกอบของโครงสร้างข้อมูลจะเก็บข้อมูลดังนี้

- indexP: ลำดับของกระบวนการ
- startP: เวลาเริ่มต้นของกระบวนการ

โครงสร้างข้อมูล Queue จะใช้จัดลำดับกระบวนการทำงาน โดยแต่ละองค์ประกอบของโครงสร้างข้อมูลจะเก็บข้อมูลดังนี้

- indexP: ลำดับของกระบวนการ
- BurtT: เวลาทำงานของกระบวนการ

โครงสร้างข้อมูล Process จะใช้เก็บข้อมูลเกี่ยวกับกระบวนการทำงาน โดยแต่ละองค์ประกอบของโครงสร้างข้อมูลจะเก็บข้อมูลดังนี้

- BurtT: เวลาทำงานของกระบวนการ
- ArrivalT: เวลามาถึงของกระบวนการ
- Priority: ลำดับความสำคัญ

โดยโครงสร้างข้อมูล Process จะถูกใช้สำหรับคำนวณเวลาเริ่มต้นของกระบวนการทำงานตามลำดับความสำคัญที่กำหนดไว้

```
    //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;     //ตัวชี้ค่าในคิว

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

กำหนดค่าเริ่มต้นสำหรับตัวแปรและฟังก์ชันต่างๆ ที่ใช้ในการทำงานของระบบปฏิบัติการแบบ FCFS (First Come First Serve)

ตัวแปร NT เก็บค่าเวลาปัจจุบัน

ตัวแปร NG เก็บจำนวนองค์ประกอบในแผนภูมิ Gantt

ตัวแปร SP เก็บตัวชี้ค่าในคิว

ฟังก์ชัน push() ใช้ในการเพิ่มกระบวนการลงในคิว โดยกำหนดค่า indexP เป็นลำดับของกระบวนการ และ LeftTimeBurt เป็นเวลาทำงานที่เหลือของกระบวนการ

ฟังก์ชัน pop() ใช้ในการลบกระบวนการออกจากคิว

การทำงานของโค้ดส่วนนี้สามารถอธิบายได้ดังนี้

- ตัวแปร NT จะถูกใช้สำหรับเก็บค่าเวลาปัจจุบัน โดยค่าเริ่มต้นจะเท่ากับ 0
- ตัวแปร NG จะถูกใช้สำหรับเก็บจำนวนองค์ประกอบในแผนภูมิ Gantt โดยค่าเริ่มต้นจะเท่ากับ 0
- ตัวแปร SP จะถูกใช้สำหรับเก็บตัวชี้ค่าในคิว โดยค่าเริ่มต้นจะเท่ากับ 0
- ฟังก์ชัน push() จะถูกใช้สำหรับเพิ่มกระบวนการลงในคิว โดยกำหนดค่า indexP เป็นลำดับของกระบวนการ และ LeftTimeBurt เป็นเวลาทำงานที่เหลือของกระบวนการ
- ฟังก์ชัน pop() จะถูกใช้สำหรับลบกระบวนการออกจากคิว

โดยฟังก์ชัน push() และ pop() จะทำงานร่วมกันเพื่อจัดลำดับกระบวนการในคิว โดยกระบวนการที่มีเวลาทำงานเหลือน้อยที่สุดจะถูกจัดลำดับไว้ข้างหน้า

สำหรับตัวแปร P[NP+1] เป็นตัวแปรสำหรับเก็บข้อมูลเกี่ยวกับกระบวนการทำงาน โดยแต่ละองค์ประกอบของโครงสร้างข้อมูลจะเก็บข้อมูลดังนี้

- indexP: ลำดับของกระบวนการ
- BurtT: เวลาทำงานของกระบวนการ
- ArrivalT: เวลามาถึงของกระบวนการ
- Priority: ลำดับความสำคัญ

ในโค้ดตัวอย่างนี้ ตัวแปร P[NP+1] จะถูกกำหนดค่าเริ่มต้นดังนี้

- P1: BurtT = 9, ArrivalT = 1, Priority = 3
- P2: BurtT = 3, ArrivalT = 1, Priority = 5
- P3: BurtT = 5, ArrivalT = 3, Priority = 1
- P4: BurtT = 4, ArrivalT = 4, Priority = 4
- P5: BurtT = 2, ArrivalT = 7, Priority = 2

โดยกระบวนการ P1 มีเวลาทำงานนานที่สุด กระบวนการ P2 มีเวลาทำงานน้อยที่สุด และกระบวนการ P5 มีลำดับความสำคัญสูงสุด

```
43  Queue sortQ(){                  //เรียงข้อมูลจากน้อยไปมาก
44      Queue tempP;
45      int j;
46      for (int i = 1; i <SP ; ++i) {
47          j = i+1;
48          if(Q[i].BurtT <= Q[j].BurtT){
49              tempP = Q[i];
50              Q[i]  = Q[j];
51              Q[j]  = tempP;
52          }
53      }
54      return Q[SP];
55  }
```

ฟังก์ชัน sortQ() ใช้สำหรับจัดเรียงกระบวนการในคิว โดยเรียงจากน้อยไปมากตามเวลาทำงานที่เหลือของกระบวนการ

ฟังก์ชันนี้ใช้หลักการของ Bubble Sort โดยเปรียบเทียบกระบวนการที่อยู่ติดกันทีละคู่ โดยกระบวนการที่มีเวลาทำงานเหลือน้อยกว่าจะอยู่ข้างหน้า

การทำงานของฟังก์ชันนี้สามารถอธิบายได้ดังนี้

- ฟังก์ชันนี้จะวนลูปจากกระบวนการที่ 2 ไปจนถึงกระบวนการตัวสุดท้ายในคิว
- สำหรับแต่ละกระบวนการที่วนลูป ฟังก์ชันจะเปรียบเทียบเวลาทำงานที่เหลือของกระบวนการนั้นกับเวลาทำงานที่เหลือของกระบวนการที่อยู่ถัดไป
- หากพบว่าเวลาทำงานที่เหลือของกระบวนการที่ 1 น้อยกว่าเวลาทำงานที่เหลือของกระบวนการที่ 2 ฟังก์ชันจะทำการสลับที่ตำแหน่งของทั้งสองกระบวนการ

ฟังก์ชัน sortQ() จะวนลูปจนกว่าจะสิ้นสุดกระบวนการในคิว หรือจนกว่าเวลาทำงานที่เหลือของกระบวนการทั้งหมดจะเท่ากัน

```
57  int minBurt(int indexP, int tempP[], int *countP) {  //ในกรณีที่มีโปรเซสเกิดในเวลาเดียวกัน หาค่าโปรเซสที่ใช้เวลาทำงานน้อยที่สุ
58      int minBurt = P[indexP].BurtT;
59      int minPro = indexP;
60      *countP = 0;
61      for (int i = 1; i <= NP; ++i)
62          if (P[indexP].ArrivalT == P[i].ArrivalT) {
63              tempP[*countP] = i;                    //เก็บโปรเซสที่เกิดในเวลาเดียวกัน
64              *countP+=1;                            //จำนวนโปรเซสที่เกิดในเวลาเดียวกัน
65              if(P[i].BurtT < minBurt ){
66                  minBurt = P[i].BurtT;
67                  minPro = i;
68              }
69          }
70      return minPro;                                 //โปรเซสที่ใช้เวลาทำงานน้อยที่สุด
71  }
72
73  void pushSynchronous(int indexP, int tempP[], int countP) {//เก็บโปรเซสที่เกิดพร้อมกันไว้ในคิว(ในเวลาเดียวกัน)
74      for (int i = 0; i < countP ; ++i)
75          if(tempP[i] != indexP )                          //ยกเว้นโปรเซสที่ใช้เวลาทำงานน้อยที่สุด
76              push(tempP[i],P[tempP[i]].BurtT);
77  }
```

ฟังก์ชัน minBurt() ใช้สำหรับหากระบวนการที่ใช้เวลาทำงานน้อยที่สุดในกรณีที่มีกระบวนการเกิดในเวลาเดียวกัน

ฟังก์ชันทำงานดังนี้

- วนลูปจากกระบวนการที่ 1 ไปจนถึงกระบวนการตัวสุดท้าย
- หากพบว่าเวลามาถึงของกระบวนการนั้นเท่ากับเวลามาถึงของกระบวนการที่ส่งเข้าฟังก์ชัน ฟังก์ชันจะเก็บข้อมูลของกระบวนการนั้นลงในตัวแปร tempP
- บันทึกจำนวนกระบวนการที่เกิดในเวลาเดียวกันลงในตัวแปร countP
- วนลูปจาก 0 ไปจนถึงจำนวนกระบวนการที่เกิดในเวลาเดียวกัน
- หากพบว่ากระบวนการนั้นไม่ใช่กระบวนการที่ใช้เวลาทำงานน้อยที่สุด ฟังก์ชันจะเพิ่มกระบวนการนั้นลงในคิว

ฟังก์ชัน pushSynchronous() ใช้สำหรับเพิ่มกระบวนการที่เกิดพร้อมกันไว้ในคิว โดยยกเว้นกระบวนการที่ใช้เวลาทำงานน้อยที่สุด

ฟังก์ชันทำงานดังนี้

- วนลูปจาก 0 ไปจนถึงจำนวนกระบวนการที่เกิดในเวลาเดียวกัน
- หากพบว่ากระบวนการนั้นไม่ใช่กระบวนการที่ใช้เวลาทำงานน้อยที่สุด ฟังก์ชันจะเพิ่มกระบวนการนั้นลงในคิว

```
79  void getData(int indexP, int time_i){
80      Gantt[NG].indexP = indexP;          //เก็บโปรเซสเพื่อทำ GanttChart
81      Gantt[NG].startP = time_i;          //เก็บเวลาเริ่มทำงานโปรเซสเพื่อทำ GanttChart
82      NG++;
83  }
```

ฟังก์ชัน getData() ใช้สำหรับเก็บข้อมูลของกระบวนการลงในแผนภูมิ Gantt

ฟังก์ชันทำงานดังนี้

- บันทึกลำดับของกระบวนการลงในตัวแปร Gantt[NG].indexP
- บันทึกเวลาเริ่มทำงานของกระบวนการลงในตัวแปร Gantt[NG].startP
- เพิ่มจำนวนองค์ประกอบในแผนภูมิ Gantt ขึ้น 1

ฟังก์ชัน getData() จะถูกเรียกใช้งานเมื่อกระบวนการที่มีเวลาทำงานน้อยที่สุดถูกเพิ่มลงในคิว ฟังก์ชันจะบันทึกลำดับของกระบวนการนั้นลงในตัวแปร Gantt[NG].indexP และบันทึกเวลาเริ่มทำงานของกระบวนการนั้นลงในตัวแปร Gantt[NG].startP โดยค่าเวลาเริ่มทำงานของกระบวนการนั้นเท่ากับค่าเวลาปัจจุบัน (NT)

```
85  void SJF_P(){
86      int indexP = 0, ENDPro = 0, runningP = 0;
87      int tempP[NP], countP = 0;
88      for (int i = 0; i <= NT; ++i) {             //i แทนเวลา (Time)
89          for (int j = 1; j <= NP; ++j) {
90              if(i == P[j].ArrivalT){             //ณ เวลาที่ i มีโปรเซส 1 2 3 ...N เกิดขึ้นใหม่
91                  indexP = minBurt(j,tempP,&countP);
92                  if(i >= ENDPro && SP == 0){     //ณ เวลาที่ i ถ้าไม่มีโปรเซสลกำลังทำงาน(ไม่มีการใช้ทรัพยากร) และ ไม่มีโปรเซ
93                      pushSynchronous(indexP,tempP,countP);//ถ้ามีโปรเซสเกิดขึ้นพร้อกัน(ในเวลาเดียวกัน)
94                      ENDPro = i + P[indexP].BurtT;//เวลาที่โปรเซสจะจบการทำงาน
95                      runningP = indexP;          //โปรเซสที่กำลังทำงานข
96                      getData(indexP,i);          //เก็บข้อมูลโปรเซส
97                  }else{                          //ณ เวลาที่ i ถ้ามีโปรเซสลกำลังทำงาน(มีการใช้ทรัพยากร) หรือ มีโปรเซสออยู่ใ
98                      if(P[indexP].BurtT < (P[runningP].BurtT-i)  && i < ENDPro){//ให้เช็คว่าโปรเซสที่เกิดใหม่ใช้เวลาทำงานน
99                          pushSynchronous(indexP,tempP,countP);
100                         push(runningP, (P[runningP].BurtT - i));  //เก็บโปรเซสที่กำลังทำงานไว้ในคิว และเวลาทำงานที่เหลือ
101                         ENDPro = i + P[indexP].BurtT;
102                         runningP = indexP;                      //โปรเซสเกิดใหม่ที่ใช้เวลาทำงานน้อยกว่า เริ่มทำงาน
103                         getData(indexP, i);
104                     }else {
105                         push(indexP, P[indexP].BurtT);//เก็บค่าโปรเซสที่เกิดใหม่ ณ เวลาที่ i ไว้ในคิว
106                         pushSynchronous(indexP, tempP, countP);
107                     }
108                 }
109                 break;
110             }
111         }
```

ฟังก์ชัน SJF_P() ใช้สำหรับจำลองการทำงานของระบบปฏิบัติการแบบ SJF (Shortest Job First)

ฟังก์ชันทำงานดังนี้

- วนลูปจากเวลา 0 ไปจนถึงเวลาสิ้นสุด
- สำหรับแต่ละช่วงเวลา
    - ตรวจสอบว่ามีกระบวนการใหม่เกิดขึ้นหรือไม่

- หากมีกระบวนการใหม่เกิดขึ้น
    - หากระบวนการที่ใช้เวลาทำงานน้อยที่สุด
    - หากไม่มีกระบวนการใดทำงานอยู่
        - เริ่มทำงานกระบวนการที่ใช้เวลาทำงานน้อยที่สุด
        - เก็บข้อมูลของกระบวนการที่เริ่มทำงานลงในแผนภูมิ Gantt
    - หากมีกระบวนการทำงานอยู่
        - หากกระบวนการที่ใช้เวลาทำงานน้อยที่สุดใช้เวลาทำงานน้อยกว่ากระบวนการที่กำลังทำงานอยู่
            - หยุดกระบวนการที่กำลังทำงานอยู่
            - เริ่มทำงานกระบวนการที่ใช้เวลาทำงานน้อยที่สุด
            - เก็บข้อมูลของกระบวนการที่เริ่มทำงานลงในแผนภูมิ Gantt
        - หากกระบวนการที่ใช้เวลาทำงานน้อยที่สุดใช้เวลาทำงานเท่ากันหรือมากกว่ากระบวนการที่กำลังทำงานอยู่
            - เพิ่มกระบวนการที่ใช้เวลาทำงานน้อยที่สุดลงในคิว
- ตรวจสอบว่ากระบวนการใดทำงานเสร็จหรือไม่
    - หากมีกระบวนการทำงานเสร็จ
        - ลบกระบวนการนั้นออกจากคิว
- อัปเดตเวลาสิ้นสุดของกระบวนการที่ทำงานอยู่

```
112         if (i >= ENDPro && SP != 0) {            //ณ เวลาที่ i ถ้าไม่มีโปรเซสไหนทำงาน(ทรัพยากรว่าง) แต่ ยังมีโปรเซสเหลืออยู่
113             Queue indexPQ;
114             indexPQ = sortQ();                    //เรียงโปรเซสในคิว โดยดูจากโปรเซสที่ใช้เวลาการทำงานน้อยที่สุด
115             pop();                               //นำโปรเซสออกจากคิว
116             ENDPro = i + indexPQ.BurtT;
117             runningP = indexPQ.indexP;
118             getData(indexPQ.indexP,i);
119         }
120     }
121 }
```

ใช้สำหรับจัดการกรณีที่กระบวนการที่กำลังทำงานอยู่เสร็จสิ้นการทำงาน และยังมีกระบวนการเหลืออยู่ในคิว หากกระบวนการที่กำลังทำงานอยู่เสร็จสิ้นการทำงาน (ENDPro <= i) และยังมีกระบวนการเหลืออยู่ในคิว (SP != 0) ฟังก์ชันจะทำดังนี้

- เรียงลำดับกระบวนการในคิว โดยดูจากโปรเซสที่ใช้เวลาการทำงานน้อยที่สุด
- ลบกระบวนการออกจากคิว
- อัปเดตเวลาสิ้นสุดของกระบวนการที่ทำงานอยู่
- เก็บข้อมูลของกระบวนการที่เริ่มทำงานลงในแผนภูมิ Gantt

```
123 float waitProcess(int indexP){                          // คำนวณหา เวลารอของโปรเซสที่ i
124     int count = 0;
125     float waitT =0, end = 0;
126     for (int i = 0; i <NG ; i++) {
127         if(Gantt[i].indexP == indexP){
128             if(count == 0){                             // เวลาโปรเซสได้เข้าทำงาน - เวลาแกิดของโปรเซส
129                 waitT += (float)Gantt[i].startP - P[indexP].ArrivalT;
130                 end    = (float)Gantt[i+1].startP;
131                 count++;
132             }else{
133                 waitT += (float)Gantt[i].startP - end;      // เวล่ที่โปรเซสได้เข้าทำงานอีกครั้ง - เวลาที่โปรเซสจบการทำงานค
134                 end    = (float)Gantt[i+1].startP;
135             }
136         }
137     }
138     return waitT;
139 }
```

ฟังก์ชัน waitProcess() ใช้สำหรับคำนวณหาเวลารอของกระบวนการที่ระบุโดย indexP

ฟังก์ชันทำงานดังนี้

- วนลูปจาก 0 ไปจนถึงจำนวนองค์ประกอบในแผนภูมิ Gantt
- หากพบองค์ประกอบที่ตรงกับ indexP
    - หากเป็นองค์ประกอบแรก
        - คำนวณเวลารอเท่ากับเวลาเริ่มทำงานของกระบวนการ - เวลามาถึงของกระบวนการ
    - หากไม่ใช่องค์ประกอบแรก
        - คำนวณเวลารอเท่ากับเวลาเริ่มทำงานของกระบวนการ – เวลาที่กระบวนการจบการทำงานครั้งก่อน

```
141  void calNT(){                                          //คำนวณหาผลรวมของ burt time
142      int sumBurt = 0;
143      int minArrival = P[1].ArrivalT;
144      for (int i = 1; i <= NP; ++i) {
145          if(P[i].ArrivalT < minArrival){
146              minArrival = P[i].ArrivalT;
147          }
148          sumBurt += P[i].BurtT;
149      }
150      NT = minArrival + sumBurt;                         //เวลาที่เริ่มเกิดโปรเซสตัวแรก + ผลรวมเวลาที่ใช้ในการทำงานของโปร
151  }
```

ฟังก์ชัน calNT() ใช้สำหรับคำนวณหาเวลาสิ้นสุดของระบบ

ฟังก์ชันทำงานดังนี้

- วนลูปจาก 1 ไปจนถึงจำนวนกระบวนการทั้งหมด
- หาเวลามาถึงของกระบวนการที่น้อยที่สุด
- หาผลรวมของเวลาทำงานของทุกกระบวนการ
- คำนวณเวลาสิ้นสุดของระบบเท่ากับเวลามาถึงของกระบวนการที่น้อยที่สุด + ผลรวมของเวลาทำงานของทุกกระบวนการ

```
153  int main(){
154      calNT();
155     printf("# Miss Ratchaneekorn Chuadee ID:65543206077-1 Sec 02\n");
156      printf("# OUTPUT LAB6 CPU Scheduling\n");
157      printf("# SJF Preemptive \n");
158      printf("Sequence process is :");
159      SJF_P();
160      for (int i = 0; i <NG ; i++) {
161          printf("P%d", Gantt[i].indexP);
162          if(i<NG-1)
163              printf("->");
164      }
165      printf("\n------------------------------------------------------------\n");
166      printf("Wait time of process (millisecond)\n");
167      for (int i = 1; i <= NP; i++) {
168          printf("| P%-10d", i);
169      }
170      printf("\n");
171      float sum = 0, avgTime;
172      for (int i = 1; i <= NP; i++) {
173          printf("| %-11.2f",waitProcess(i));
174          sum += waitProcess(i);
175      }
176      avgTime = sum/NP;
177      printf("\nAverage time is %.2f", avgTime);
178      printf("\nTurnaround time\n");
179      for (int i = 1; i <= NP; i++) {
180          printf("|P%d = %-6.2fms ", i, waitProcess(i) + P[i].BurtT);
181      }
182      printf("\n------------------------------------------------------------\n");
183      return 0;
184  }
```

นี่คือคำอธิบายโค้ดหลัก (main function) ของโปรแกรมจำลองการ scheduling แบบ SJF preemptive:

เริ่มต้น

- เรียกใช้ฟังก์ชัน calNT() เพื่อคำนวณหาเวลาสิ้นสุดของระบบ
- พิมพ์ข้อมูลเบื้องต้นของโปรแกรมและชื่อผู้เขียน

จำลองการ scheduling

- เรียกใช้ฟังก์ชัน SJF_P() เพื่อจำลองการ scheduling แบบ SJF preemptive
- พิมพ์ลำดับการทำงานของกระบวนการต่างๆ ตามแผนภูมิ Gantt

แสดงผลลัพธ์

- พิมพ์ข้อมูลเวลารอของกระบวนการแต่ละกระบวนการ
- คำนวณหาเวลารอเฉลี่ย
- พิมพ์ข้อมูล turnaround time ของกระบวนการแต่ละกระบวนการ

จบการทำงาน

- จบการทำงานของโปรแกรม

ผลลัพธ์

```
# Miss Ratchaneekorn Chuadee ID:65543206077-1 Sec 02
# 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
------------------------------------------------------------
PS D:\เทอม 2 66\OS_Lab\lab6>
```

ลำดับการทำงานของกระบวนการ:

- P2 -> P4 -> P5 -> P3 -> P1 (ตามแผนภูมิ Gantt)

เวลารอของกระบวนการ:

- P1: 14.00 มิลลิวินาที
- P2: 0.00 มิลลิวินาที
- P3: 7.00 มิลลิวินาที
- P4: 0.00 มิลลิวินาที
- P5: 1.00 มิลลิวินาที

เวลารอเฉลี่ย:

- 4.40 มิลลิวินาที

เวลาตอบสนอง (Turnaround Time):

- P1: 23.00 มิลลิวินาที
- P2: 3.00 มิลลิวินาที
- P3: 12.00 มิลลิวินาที
- P4: 4.00 มิลลิวินาที
- P5: 3.00 มิลลิวินาที

ข้อสังเกต:

- กระบวนการ P2, P4 และ P5 ใช้เวลารอน้อยที่สุด เนื่องจากเป็นกระบวนการที่ใช้เวลาทำงานน้อยและเกิดขึ้นก่อนกระบวนการอื่นๆ
- กระบวนการ P1 ใช้เวลารอนานที่สุด เนื่องจากเป็นกระบวนการที่ใช้เวลาทำงานนานที่สุด และถูกขัดจังหวะโดยกระบวนการที่มีเวลาทำงานน้อยกว่าหลายครั้ง
- เวลารอเฉลี่ยของกระบวนการทั้งหมดค่อนข้างต่ำ เนื่องจากการใช้ SJF Preemptive ทำให้กระบวนการที่ใช้เวลาทำงานน้อยได้ทำงานก่อน ส่งผลให้ลดเวลารอโดยรวมลง

Round Robin scheduling (Time quantum = 4)

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

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

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

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

การรวมไลบรารี:

- #include <stdio.h>: เป็นการรวมไลบรารีมาตรฐานสำหรับอินพุตและเอาต์พุต ซึ่งช่วยให้โค้ดสามารถแสดงผลข้อความและรับข้อมูลเข้าได้

การกำหนดค่าคงที่:

- #define N 5: เป็นการประกาศค่าคงที่ชื่อ N และกำหนดค่าเท่ากับ 5 ซึ่งในโค้ดนี้จะใช้สำหรับกำหนดจำนวนกระบวนการ (process) ที่ต้องการจัดการ
- #define T_SLICE 4: เป็นการประกาศค่าคงที่ชื่อ T_SLICE และกำหนดค่าเท่ากับ 4 ซึ่งในโค้ดนี้จะใช้สำหรับกำหนดเวลาการทำงานแบบแบ่งชิ้น (quantum time)
- #define NQ 20: เป็นการประกาศค่าคงที่ชื่อ NQ และกำหนดค่าเท่ากับ 20 ซึ่งในโค้ดนี้จะใช้สำหรับกำหนดจำนวนช่องเก็บคิว

การประกาศโครงสร้างข้อมูล:

- Process: เป็นโครงสร้างข้อมูลที่ใช้สำหรับเก็บข้อมูลเกี่ยวกับกระบวนการแต่ละกระบวนการ ประกอบด้วยสมาชิกดังนี้
    - BurtT: ใช้เก็บเวลาที่ใช้ในการทำงาน (burst time) ของกระบวนการ
    - ArrivalT: ใช้เก็บเวลาที่กระบวนการมาถึง (arrival time)
    - Priority: ใช้เก็บลำดับความสำคัญ (priority) ของกระบวนการ
- Queue: เป็นโครงสร้างข้อมูลที่ใช้สำหรับเก็บกระบวนการที่กำลังรอคิวเพื่อทำงาน ประกอบด้วยสมาชิกดังนี้

- indexP: ใช้เก็บหมายเลขของกระบวนการ
- BurtT: ใช้เก็บเวลาที่ใช้ในการทำงาน (burst time) ของกระบวนการ

- Gantt_C: เป็นโครงสร้างข้อมูลที่ใช้สำหรับเก็บข้อมูลเกี่ยวกับลำดับการทำงานของกระบวนการ ประกอบด้วยสมาชิกดังนี้
  - indexP: ใช้เก็บหมายเลขของกระบวนการ
  - startP: ใช้เก็บเวลาที่กระบวนการเริ่มต้นทำงาน

โค้ดส่วนนี้ทำหน้าที่กำหนดโครงสร้างพื้นฐานสำหรับการจัดเก็บข้อมูลกระบวนการและข้อมูลเกี่ยวกับการทำงาน ซึ่งจะถูกนำไปใช้ในการคำนวณลำดับการทำงานของกระบวนการในขั้นตอนต่อไป

ความแตกต่างระหว่าง code สองส่วน:

ความแตกต่างหลักระหว่าง code ทั้งสองส่วนคือจำนวนกระบวนการที่ต้องการจัดการ โดย code ส่วนแรกกำหนดจำนวนกระบวนการไว้ที่ 5 ส่วน code ส่วนที่สองกำหนดจำนวนกระบวนการไว้ที่ 5

นอกจากนี้ code ส่วนที่สองยังมีการกำหนดค่าคงที่เพิ่มเติมอีกสองค่า คือ T_SLICE และ NQ โดย T_SLICE ใช้สำหรับกำหนดเวลาการทำงานแบบแบ่งชิ้น และ NQ ใช้สำหรับกำหนดจำนวนช่องเก็บคิว

โดยสรุปแล้ว code ทั้งสองส่วนมีโครงสร้างพื้นฐานที่เหมือนกัน แต่ code ส่วนที่สองมีรายละเอียดเพิ่มเติมสำหรับกำหนดจำนวนกระบวนการและเวลาการทำงานแบบแบ่งชิ้น

```
22  //Process  burt time , Arrival time , Priority
23  Process P[N+1]  =  {{0},
24          /*P1*/ {    9     ,       1      ,      3},     // P1 = P[1]
25          /*P2*/ {    3     ,       1      ,      5},     // P2 = P[2]
26          /*P3*/ {    5     ,       3      ,      1},     // P3 = P[3]
27          /*P4*/ {    4     ,       4      ,      4},     // P4 = P[4]
28          /*P5*/ {    2     ,       7      ,      2}};    // P5 = P[5]
29
30
31  Gantt_C Gantt[20];
32  Queue   Q[NQ];   // คิว
33  int F = 0, R = 0;// ตัวชี้คิววงกลม Fชี้หน้า Rชี้หลัง
34  int NG = 0;      // number Gantt_chart
35  int NT = 0;      // number time
36  int NP = N;      // number process
```

การกำหนดค่าเริ่มต้น:

- Process P[N+1] = {{0}}: เป็นการประกาศอาร์เรย์ P ที่มีขนาดเท่ากับจำนวนกระบวนการที่ต้องการจัดการ (N) + 1 โดยกำหนดค่าเริ่มต้นให้กับทุกกระบวนการในอาร์เรย์เป็นค่า 0

การกำหนดค่าให้กับกระบวนการแต่ละกระบวนการ:

/*P1*/ { 9 , 1 , 3}, // P1 = P[1]

- โค้ดส่วนนี้เป็นการระบุค่าให้กับกระบวนการแต่ละกระบวนการในอาร์เรย์ P โดยระบุค่า burst time, arrival time และ priority ตามลำดับ

การประกาศโครงสร้างข้อมูล:

- Gantt_C Gantt[20]: เป็นการประกาศอาร์เรย์ Gantt ที่มีขนาดเท่ากับ 20 โดยแต่ละองค์ประกอบในอาร์เรย์จะเป็นโครงสร้างข้อมูล Gantt_C ซึ่งใช้สำหรับเก็บข้อมูลเกี่ยวกับลำดับการทำงานของกระบวนการ
- Queue Q[NQ]: เป็นการประกาศอาร์เรย์ Q ที่มีขนาดเท่ากับ NQ โดยแต่ละองค์ประกอบในอาร์เรย์จะเป็นโครงสร้างข้อมูล Queue ซึ่งใช้สำหรับเก็บกระบวนการที่กำลังรอคิวเพื่อทำงาน
- int F = 0, R = 0;: เป็นการประกาศตัวแปร F และ R เพื่อใช้เป็นตัวชี้คิววงกลม โดย F ชี้หน้า R ชี้หลัง
- int NG = 0;: เป็นการประกาศตัวแปร NG เพื่อใช้เก็บจำนวนลำดับการทำงานของกระบวนการ
- int NT = 0;: เป็นการประกาศตัวแปร NT เพื่อใช้เก็บจำนวนเวลา
- int NP = N;: เป็นการประกาศตัวแปร NP เพื่อใช้เก็บจำนวนกระบวนการ

โค้ดส่วนนี้ทำหน้าที่กำหนดโครงสร้างพื้นฐานสำหรับการจัดเก็บข้อมูลกระบวนการและข้อมูลเกี่ยวกับการทำงาน ซึ่งจะถูกนำไปใช้ในการคำนวณลำดับการทำงานของกระบวนการในขั้นตอนต่อไป

รายละเอียดเพิ่มเติม:

- การกำหนดค่าให้กับกระบวนการแต่ละกระบวนการเป็นการระบุค่า burst time, arrival time และ priority ตามลำดับ โดย burst time คือเวลาที่ใช้ในการทำงานของกระบวนการ arrival time คือเวลาที่กระบวนการมาถึง และ priority คือลำดับความสำคัญของกระบวนการ
- การกำหนดค่าให้กับตัวแปร F และ R เป็นการระบุตำแหน่งเริ่มต้นของตัวชี้คิววงกลม โดย F ชี้หน้า R ชี้หลัง
- การกำหนดค่าให้กับตัวแปร NG และ NT เป็นการระบุจำนวนลำดับการทำงานของกระบวนการและจำนวนเวลาตามลำดับ
- การกำหนดค่าให้กับตัวแปร NP เป็นการระบุจำนวนกระบวนการ

```
38 void pushQ(int index, int BTimeLeft){
39     if (R == F-1  || (F == 1 && R == NQ-1)) {
40         printf("OVER FLOW!!\n");
41     }else{
42         if(R == NQ-1){
43             R = 1;
44         }else{
45             R++;
46             if(F == 0)
47                 F = 1;
48         }
49         Q[R].indexP = index;              // เก็บโปรเซสไว้ในคิว
50         Q[R].BurtT  = BTimeLeft;          // เก็บเวลาการทำงานของโปรเซสที่เหลือไว้ในคิว
51     }
52 }
```

ฟังก์ชัน pushQ() ใช้สำหรับเพิ่มกระบวนการลงในคิว

พารามิเตอร์:

- index: เป็นหมายเลขของกระบวนการที่ต้องการเพิ่ม
- BTimeLeft: เป็นเวลาที่กระบวนการที่เหลืออยู่

การทำงาน:

- โค้ดส่วนแรกจะตรวจสอบว่าคิวเต็มหรือไม่ หากเต็มจะแสดงข้อความแจ้งเตือนและยุติการทำงานของฟังก์ชัน
- หากคิวไม่เต็ม โค้ดส่วนต่อไปจะตรวจสอบว่าตัวชี้คิววงกลม R ชี้ถึงตำแหน่งสุดท้ายของคิวหรือไม่ หากชี้ถึงตำแหน่งสุดท้าย โค้ดจะเลื่อนตัวชี้คิววงกลม R มาเริ่มต้นใหม่ที่ตำแหน่งที่ 1
- หากตัวชี้คิววงกลม R ไม่ชี้ถึงตำแหน่งสุดท้าย โค้ดจะเพิ่มค่าของตัวชี้คิววงกลม R ขึ้น 1 หากตัวชี้คิววงกลม R เท่ากับ 0 โค้ดจะเลื่อนตัวชี้คิววงกลม F มาชี้ที่ตำแหน่งที่ 1
- สุดท้าย โค้ดจะเก็บข้อมูลของกระบวนการที่ต้องการเพิ่มลงในคิว โดยเก็บหมายเลขของกระบวนการลงในสมาชิก indexP และเก็บเวลาที่กระบวนการที่เหลืออยู่ลงในสมาชิก BurtT

```
52  Queue popQ() {
53    if (F == 0) {
54      printf("UNDER FLOW!!\n");
55      return {};
56    } else {
57      Queue index = Q[F];
58      if (F == R) {
59        F = 0; R = 0;
60      } else {
61        if (F == NQ-1)
62          F = 1;
63        else
64          F++;
65      }
66      return index;
67    }
68  }
```

ฟังก์ชัน popQ() ทำหน้าที่นำโปรเซสออกจากคิว การทำงานของฟังก์ชันมีดังนี้

- ตรวจสอบว่าคิวว่างเปล่าหรือไม่ หากว่างเปล่า ฟังก์ชันจะพิมพ์ข้อความแจ้งและส่งค่ากลับเป็น Queue ว่างเปล่า
- หากคิวไม่ว่างเปล่า ฟังก์ชันจะดึงโปรเซสออกโดยดึง Queue ที่อยู่ตำแหน่ง F ออกจากคิว
- อัปเดตค่าของ F โดยเลื่อนค่าขึ้น 1 หาก F มีค่าเท่ากับ NQ-1 ให้ตั้ง F เป็น 1 มิฉะนั้นให้ตั้ง F เป็น F+1
- ส่งค่า Queue ของโปรเซสที่ดึงออกกลับ

ตัวอย่าง เช่น หากคิวมีโปรเซสอยู่ 2 โปรเซส โดยโปรเซสแรกมี index เท่ากับ 1 และโปรเซสที่สองมี index เท่ากับ 2 เมื่อเรียกใช้ฟังก์ชัน popQ() ครั้งแรก ฟังก์ชันจะดึงโปรเซสแรกออกจากคิวและส่งค่า Queue ของโปรเซสแรกกลับ โดย index ของโปรเซสแรกจะเป็น 1

หากเรียกใช้ฟังก์ชัน popQ() ครั้งที่สอง ฟังก์ชันจะดึงโปรเซสที่สองออกจากคิวและส่งค่า Queue ของโปรเซสที่สองกลับ โดย index ของโปรเซสที่สองจะเป็น 2

```
71  int func_FCFS(int index){                  // โปรเซสเกิดพร้อมกัน ใช้ FCFS
72      int temp[20], n = 0;
73      for (int i = 1; i <= NP; i++) {
74          if(P[index].ArrivalT == P[i].ArrivalT){
75              temp[n++] = i;
76          }
77      }
78      for (int j = 0; j < n; j++) {       // นำโปรเต้วที่เหลือไปต่อคิว ยกเว้นโปรเซสที่เมาก่อน
79          if(temp[j] != index){
80              pushQ(temp[j],P[temp[j]].BurtT);
81          }
82      }
83      return index;                      // return โปรเซสที่มาก่อน
84  }
```

ฟังก์ชัน func_FCFS() ใช้สำหรับกำหนดลำดับการทำงานของกระบวนการโดยใช้แบบ First Come First Serve (FCFS)

การทำงาน:

- โค้ดส่วนแรกจะสร้างอาร์เรย์ temp ขนาดเท่ากับจำนวนกระบวนการ (NP) โดยกำหนดค่าเริ่มต้นให้กับทุกตำแหน่งในอาร์เรย์เป็นค่า 0
- จากนั้นโค้ดจะวนลูปจาก 1 ถึง NP เพื่อตรวจสอบว่ากระบวนการ index มาถึงพร้อมกันกับกระบวนการอื่นหรือไม่ หากมาถึงพร้อมกัน โค้ดจะเก็บหมายเลขของกระบวนการนั้นไว้ในอาร์เรย์ temp
- เมื่อวนลูปครบแล้ว โค้ดจะวนลูปอีกครั้งจาก 0 ถึง n-1 เพื่อนำกระบวนการที่เหลือทั้งหมดไปต่อคิว โดยเว้นกระบวนการ index ไว้
- สุดท้าย โค้ดจะส่งคืนหมายเลขของกระบวนการ index กลับไปยังฟังก์ชันที่เรียกใช้

รายละเอียดเพิ่มเติม:

- ฟังก์ชัน func_FCFS() ทำงานได้เฉพาะกับกระบวนการที่มาถึงพร้อมกันเท่านั้น หากมีกระบวนการที่มาถึงก่อน ฟังก์ชันนี้จะทำงานผิดพลาด
- ฟังก์ชัน func_FCFS() มีประสิทธิภาพในการกำหนดลำดับการทำงานสูง เนื่องจากกำหนดลำดับการทำงานโดยพิจารณาจากเวลาที่กระบวนการมาถึงเท่านั้น

```
86  int duplincate(int index){              // ตรวจดูว่ามีโปรเซสเกิดในเวลาเดียวกันหรือไม่
87      int count = 0;
88      for (int i = 1; i <= NP; i++)
89          if(P[index].ArrivalT == P[i].ArrivalT)
90              count++;
91      if (count > 1)                      // ถ้ามีโปรเซสเกิดในเวลาเดียวกัน
92          return func_FCFS(index);        // หาโปรเซสที่ใช้เวลทำงานน้อยที่สุด
93      else                                // ถ้าไม่มีโปรเซสเกิดในเวลาเดียวกัน
94          return index;
95  }
```

ฟังก์ชัน duplincate() ใช้สำหรับตรวจสอบว่ากระบวนการ index มาถึงพร้อมกันกับกระบวนการอื่นหรือไม่

การทำงาน:

- โค้ดส่วนแรกจะสร้างตัวแปร count เพื่อเก็บจำนวนกระบวนการที่มาถึงพร้อมกันกับกระบวนการ index
- จากนั้นโค้ดจะวนลูปจาก 1 ถึง NP เพื่อตรวจสอบว่ากระบวนการ index มาถึงพร้อมกันกับกระบวนการอื่นหรือไม่ หากมาถึงพร้อมกัน โค้ดจะเพิ่มค่าของตัวแปร count ขึ้น 1
- เมื่อวนลูปครบแล้ว โค้ดจะตรวจสอบว่าตัวแปร count มากกว่า 1 หรือไม่ หากมากกว่า 1 แสดงว่ามีกระบวนการที่มาถึงพร้อมกันกับกระบวนการ index โค้ดจะส่งคืนผลลัพธ์ของฟังก์ชัน func_FCFS() ซึ่งจะหาโปรเซสที่ใช้เวลาทำงานน้อยที่สุด หากตัวแปร count เท่ากับ 1 แสดงว่าไม่มีกระบวนการที่มาถึงพร้อมกันกับกระบวนการ index โค้ดจะส่งคืนหมายเลขของกระบวนการ index กลับไปยังฟังก์ชันที่เรียกใช้

```
void Round_Robin(){
    int runingP = 0, timeleft, index, END_P = 0;
    for (int i = 0; i <NT ; i++) {                      // i แทรเวลา time
        if(i == END_P && runingP != 0){                 // ถ้าครบเวลา Quantum time โปรเซสที่อยู่ในสถานะทำงานยังเหลือเวล
            pushQ(runingP, timeleft);                   // ให้เก็บโปรเซสที่กำลังทำงาน และเวลาทำงานที่เหลือ ไว้ในคิวก่อน
        }
        for (int j = 1; j <= NP; j++) {
            if (i == P[j].ArrivalT) {                   // ณ เวลาที่ i มี process[j] เกิดขึ้น
                index = j;
                if (i >= END_P && (F ==0 && R == 0 )) { // ถ้าจบQuantum time = โปรเซสสามารเข้าไปทำงานได้
                    index = duplincate(index);           // ตรวจสอบว่า มีโปรเซสเกิดขึ้นพร้อมกัน ใช่หรือไม่ (ถ้าใช่ก่ใช่ FCFS)
                    if(P[index].BurtT <= T_SLICE) {     // ถ้าเวลาทำงานของโปรเซสมีค่าน้อยกว่าเท่ากับ Quantum time
                        END_P = i + P[index].BurtT;     // ให้ Quantum time จบตามเวลาที่น้อยกว่า
                        runingP = 0;                    // ไม่มีโปรเซสอยู่ในสถานะทำงาน (เพราะจบในเวลา Quantum time )
                    }
                    else {                              // ถ้าเวลาทำงานของโปรเซสมีค่ามากกว่า Quantum time
                        END_P = i + T_SLICE;            // เวลาจบการทำงานของโปรเซส
                        runingP = index;                // โปรเซสที่กำลังทำงาน
                        timeleft = P[index].BurtT - T_SLICE; //เวลาที่เหลือของโปรเซส
                    }
                    Gantt[NG].indexP = index;
                    Gantt[NG].startP = i;               //เก็บค่าเวลาที่โปรเซสได้เริ่มทำงาน
                    NG++;
                    break;
                } else{                                 // ถ้า Quantum time ยังไม่จบ = โปรเซสจะไม่สามารเข้าไปทำงานได้
                    pushQ(index, P[index].BurtT);       // เก็บโปรเซสที่กำลังทำงานไว้ในคิวก่อน
                    duplincate(index);                 // ถ้ามีโปรเซสเกิดขึ้นพร้อมกันเก็บให้เก็บโปรเซสไว้ในคิว
                    break;
                }
            }
        }
```

ฟังก์ชัน Round_Robin() ใช้สำหรับกำหนดลำดับการทำงานของกระบวนการโดยใช้แบบ Round Robin

การทำงาน:

โค้ดส่วนแรกจะกำหนดตัวแปร runingP เพื่อเก็บหมายเลขของกระบวนการที่กำลังทำงานอยู่ ตัวแปร timeleft เพื่อเก็บเวลาทำงานที่เหลืออยู่ของกระบวนการที่กำลังทำงานอยู่ และตัวแปร END_P เพื่อเก็บเวลาจบการทำงานของกระบวนการที่กำลังทำงานอยู่

จากนั้นโค้ดจะวนลูปจาก 0 ถึง NT เพื่อกำหนดลำดับการทำงาน

- หากเวลาปัจจุบัน (i) ถึงเวลาจบการทำงาน (END_P) ของกระบวนการที่กำลังทำงานอยู่ และกระบวนการที่กำลังทำงานอยู่มีค่าไม่เท่ากับ 0 โค้ดจะย้ายกระบวนการที่กำลังทำงานอยู่และเวลาทำงานที่เหลืออยู่ไปเก็บไว้ในคิวก่อน
- จากนั้นโค้ดจะวนลูปจาก 1 ถึง NP เพื่อตรวจสอบว่ากระบวนการ j เกิดขึ้น ณ เวลาปัจจุบันหรือไม่ หากเกิดขึ้น โค้ดจะดำเนินการดังนี้
    - หากเวลาปัจจุบัน (i) ถึงเวลาจบการทำงาน (END_P) ของกระบวนการที่กำลังทำงานอยู่ และกระบวนการที่กำลังทำงาน

```
128         if (i >= END_P && (F != 0 && R != 0)) {          // ถ้าQuantum time จบแต่ไม่มีโปรเซสไหนเกิดขึ้น แต่ในคิวยังมีโปรเซส
129             Queue indexQ;
130             indexQ = popQ();                            // นำโปรเซสในคิวเข้าทำงาน
131 
132             if (indexQ.BurtT <= T_SLICE) {              // ถ้าเวลาทำงานของโปรเซสมีค่าน้อยกว่าเท่ากับ Quantum time
133                 END_P = i + indexQ.BurtT;
134                 runingP = 0;
135             } else {                                    // ถ้าเวลาทำงานของโปรเซสมีค่ามากกว่า Quantum time
136                 END_P = i + T_SLICE;
137                 runingP = indexQ.indexP;;
138                 timeleft = indexQ.BurtT - T_SLICE;
139             }
140             Gantt[NG].indexP = indexQ.indexP;
141             Gantt[NG].startP = i;
142             NG++;
143         }
144     }
145 }
```

โค้ดส่วนนี้เป็นการประมวลผลกรณีพิเศษที่อาจเกิดขึ้นได้ในระหว่างการวนลูป

กรณีพิเศษ

- Quantum time จบแล้ว แต่ไม่มีกระบวนการใดเกิดขึ้น ณ เวลาปัจจุบัน แต่มีคิวกระบวนการรออยู่

การทำงาน

โค้ดส่วนนี้จะทำการตรวจสอบเงื่อนไขดังนี้

- เวลาปัจจุบัน (i) ถึงเวลาจบการทำงาน (END_P) ของกระบวนการที่กำลังทำงานอยู่ และกระบวนการที่กำลังทำงานอยู่มีค่าไม่เท่ากับ 0
- ตัวแปร F และ R ไม่เท่ากับ 0 แสดงว่ามีกระบวนการเกิดขึ้นแล้วอย่างน้อยหนึ่งกระบวนการ

หากเงื่อนไขทั้งสองข้อเป็นจริง โค้ดจะดำเนินการดังนี้

- โค้ดจะสร้างตัวแปร indexQ เพื่อเก็บข้อมูลของกระบวนการที่ดึงออกมาจากคิว
- โค้ดจะดึงกระบวนการออกจากคิวโดยใช้ฟังก์ชัน popQ()

จากนั้นโค้ดจะดำเนินการตามเงื่อนไขดังนี้

- หากเวลาทำงานของกระบวนการมีค่าน้อยกว่าเท่ากับ Quantum time โค้ดจะตั้งค่า END_P เป็นเวลาที่กระบวนการทำงานเสร็จสิ้น และ runingP เป็นค่า 0 เพื่อระบุว่าไม่มีกระบวนการใดทำงานอยู่
- หากเวลาทำงานของกระบวนการมีค่ามากกว่า Quantum time โค้ดจะตั้งค่า END_P เป็นเวลาที่กระบวนการทำงานเสร็จสิ้นบางส่วน และ runingP เป็นหมายเลขของกระบวนการเพื่อระบุว่ามีกระบวนการทำงานอยู่

สุดท้าย โค้ดจะบันทึกข้อมูลของกระบวนการลงในตาราง Gantt ดังนี้

- หมายเลขของกระบวนการ (indexP)
- เวลาเริ่มทำงาน (startP)

```
147  float waitProcess(int indexP){                                  // คำนวณหา เวลาที่โปรเซสรอ
148      int count = 0;
149      float waitT =0, end = 0;
150      for (int i = 0; i <NG ; i++) {
151          if(Gantt[i].indexP == indexP){
152              if(count == 0){                                     // เวลาโปรเซสได้เข้าทำงาน - เวลาแกิดของโปรเซส
153                  waitT += (float)Gantt[i].startP - P[indexP].ArrivalT;
154                  end    = (float)Gantt[i+1].startP;
155                  count++;
156              }else{
157                  waitT += (float)Gantt[i].startP - end;           // เวลที่โปรเซสได้เข้าทำงานอีกครั้ง - เวลาที่โปรเซสจบการทำ
158                  end    = (float)Gantt[i+1].startP;
159              }
160          }
161      }
162      return waitT;
163  }
```

โค้ดส่วนนี้ใช้สำหรับคำนวณหาเวลาที่โปรเซสรอ

การทำงาน

โค้ดส่วนนี้จะทำการตรวจสอบเงื่อนไขดังนี้

- หมายเลขของกระบวนการที่ส่งเข้ามามีค่ามากกว่าหรือเท่ากับ 1

หากเงื่อนไขเป็นจริง โค้ดจะดำเนินการดังนี้

- โค้ดจะสร้างตัวแปร count เพื่อเก็บจำนวนครั้งที่กระบวนการ indexP ทำงาน
- โค้ดจะสร้างตัวแปร end เพื่อเก็บเวลาเริ่มทำงานของกระบวนการ indexP ครั้งถัดไป

จากนั้นโค้ดจะวนลูปจาก 0 ถึง NG เพื่อตรวจสอบว่ากระบวนการ indexP ทำงานในตาราง Gantt หรือไม่

หากกระบวนการ indexP ทำงานในตาราง Gantt โค้ดจะดำเนินการดังนี้

- หาก count เท่ากับ 0 แสดงว่ากระบวนการ indexP ทำงานครั้งแรก โค้ดจะคำนวณเวลารอของกระบวนการดังนี้

  ```
  waitT += (float)Gantt[i].startP - P[indexP].ArrivalT;
  ```

- หาก count ไม่เท่ากับ 0 แสดงว่ากระบวนการ indexP ทำงานครั้งต่อๆ มา โค้ดจะคำนวณเวลารอของกระบวนการดังนี้

  ```
  waitT += (float)Gantt[i].startP - end;
  ```

  โดย end จะเก็บเวลาเริ่มทำงานของกระบวนการ indexP ครั้งถัดไป

สุดท้าย โค้ดจะส่งคืนค่าเวลาที่โปรเซสรอ

```
165  void calNT(){
166      int sumBurt = 0;
167      int minArrival = P[1].ArrivalT;
168      for (int i = 1; i <=NP; ++i) {
169          if(P[i].ArrivalT < minArrival){
170              minArrival = P[i].ArrivalT;
171          }
172          sumBurt += P[i].BurtT;
173      }
174      NT = minArrival + sumBurt;
175  }
```

ฟังก์ชัน calNT() ใช้สำหรับคำนวณหาเวลาทำงานทั้งหมดของกระบวนการ (NT)

การทำงาน:

กำหนดตัวแปร:

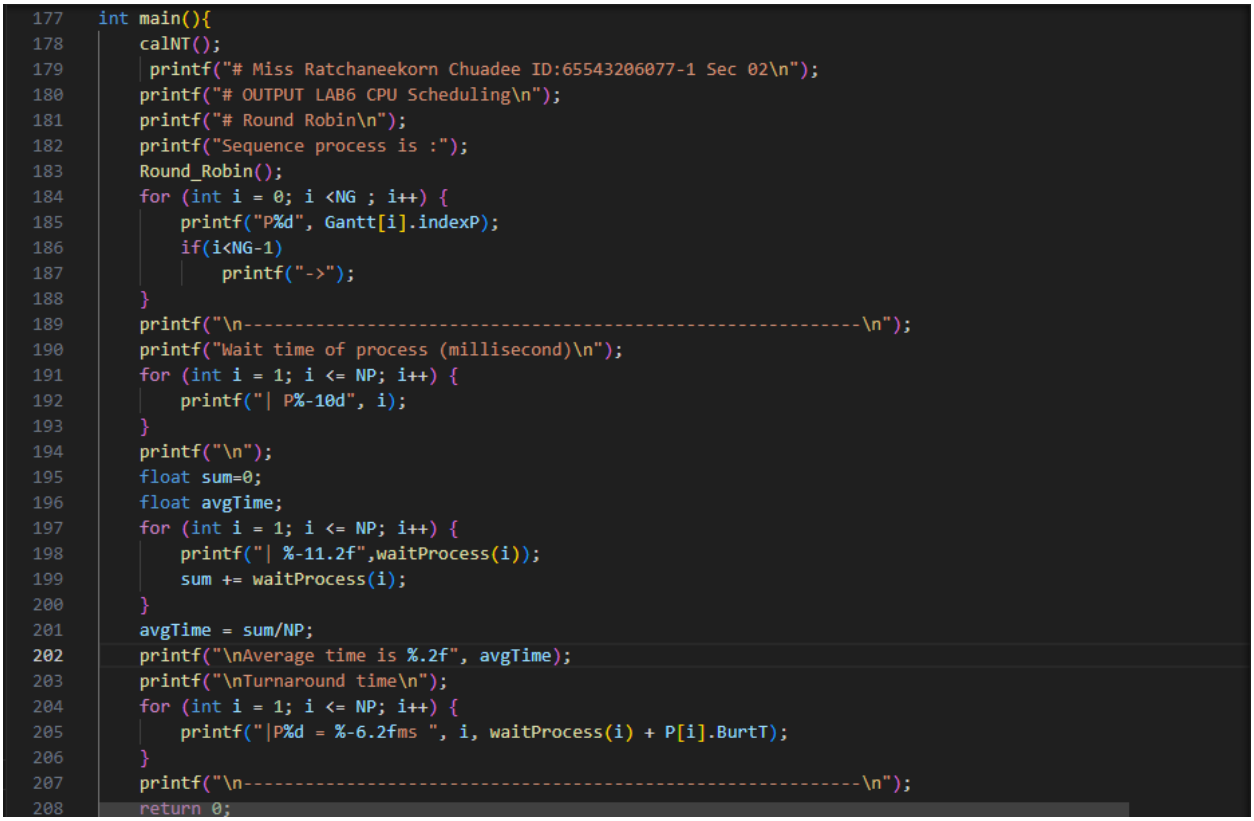
- sumBurt เพื่อเก็บผลรวมของเวลาทำงานของทุกกระบวนการ
- minArrival เพื่อเก็บเวลาที่กระบวนการมาถึงเร็วที่สุด
- NT เพื่อเก็บเวลาทำงานทั้งหมดของกระบวนการ

วนลูปผ่านกระบวนการทั้งหมด:

- วนลูปตั้งแต่ i = 1 ถึง NP เพื่อพิจารณากระบวนการทั้งหมด
- ตรวจสอบว่า P[i].ArrivalT มีค่าน้อยกว่า minArrival หรือไม่ หากใช่ ให้อัปเดต minArrival เป็น P[i].ArrivalT เพื่อเก็บเวลาที่กระบวนการมาถึงเร็วที่สุด
- เพิ่ม P[i].BurtT ลงใน sumBurt เพื่อสะสมผลรวมของเวลาทำงานของกระบวนการ

คำนวณ NT:

- หลังจบการวนลูป ให้คำนวณ NT โดยบวก minArrival กับ sumBurt
- NT จะเก็บค่าเวลาที่กระบวนการทั้งหมดทำงานเสร็จสิ้น โดยคำนวณจากเวลาที่กระบวนการมาถึงเร็วที่สุด รวมกับผลรวมของเวลาทำงานของกระบวนการทุกตัว



```c
int main(){
    calNT();
     printf("# Miss Ratchaneekorn Chuadee ID:65543206077-1 Sec 02\n");
    printf("# OUTPUT LAB6 CPU Scheduling\n");
    printf("# Round Robin\n");
    printf("Sequence process is :");
    Round_Robin();
    for (int 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 (int i = 1; i <= NP; i++) {
        printf("| P%-10d", i);
    }
    printf("\n");
    float sum=0;
    float avgTime;
    for (int 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 (int i = 1; i <= NP; i++) {
        printf("|P%d = %-6.2fms ", i, waitProcess(i) + P[i].BurtT);
    }
    printf("\n-------------------------------------------------------------\n");
    return 0;
```

ส่วน main() ของโปรแกรม มีหน้าที่ดังนี้

คำนวณ NT:

- เรียกใช้ฟังก์ชัน calNT() เพื่อคำนวณหาเวลาทำงานทั้งหมดของกระบวนการ

พิมพ์ข้อมูลเบื้องต้น:

- พิมพ์ข้อมูลชื่อ รหัสนักศึกษา และหัวข้อของโปรแกรม
- ระบุว่าใช้อัลกอริทึม Round Robin

กำหนดลำดับการทำงาน:

- เรียกใช้ฟังก์ชัน Round_Robin() เพื่อกำหนดลำดับการทำงานของกระบวนการโดยใช้ Round Robin

พิมพ์ลำดับการทำงาน:

- วนลูปผ่านตาราง Gantt และพิมพ์ลำดับการทำงานของกระบวนการ

พิมพ์เวลารอ:

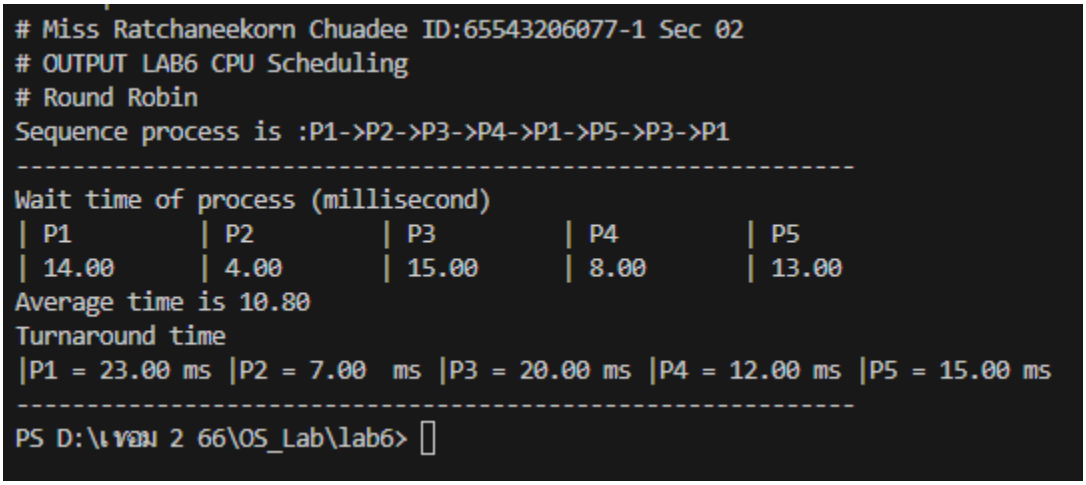
- พิมพ์หัวข้อ "Wait time of process"
- วนลูปผ่านกระบวนการทั้งหมด และพิมพ์เวลารอของแต่ละกระบวนการโดยใช้ฟังก์ชัน waitProcess(i)
- คำนวณหาเวลารอเฉลี่ยและพิมพ์ผลลัพธ์

พิมพ์ Turnaround time:

- พิมพ์หัวข้อ "Turnaround time"
- วนลูปผ่านกระบวนการทั้งหมด และพิมพ์ Turnaround time ของแต่ละกระบวนการ

ส่วน main() ทำหน้าที่ควบคุมการไหลของโปรแกรมโดยเริ่มต้นจากการคำนวนเวลาทำงานทั้งหมด กำหนดลำดับการทำงาน พิมพ์ข้อมูลต่างๆ เช่น ลำดับการทำงาน เวลารอ และ Turnaround time ของแต่ละกระบวนการออกมา

ผลลัพธ์



ลำดับการทำงานของกระบวนการ:

- P1->P2->P3->P4->P1->P5->P3->P1

เวลารอของกระบวนการ (Wait time):

- P1: 14.00 ms
- P2: 4.00 ms
- P3: 15.00 ms
- P4: 8.00 ms
- P5: 13.00 ms

เวลารอเฉลี่ย: 10.80 ms

Turnaround time:

- P1: 23.00 ms
- P2: 7.00 ms
- P3: 20.00 ms
- P4: 12.00 ms
- P5: 15.00 ms

Priority scheduling

- Priority (SJF Preemptive)

```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
```

Header:

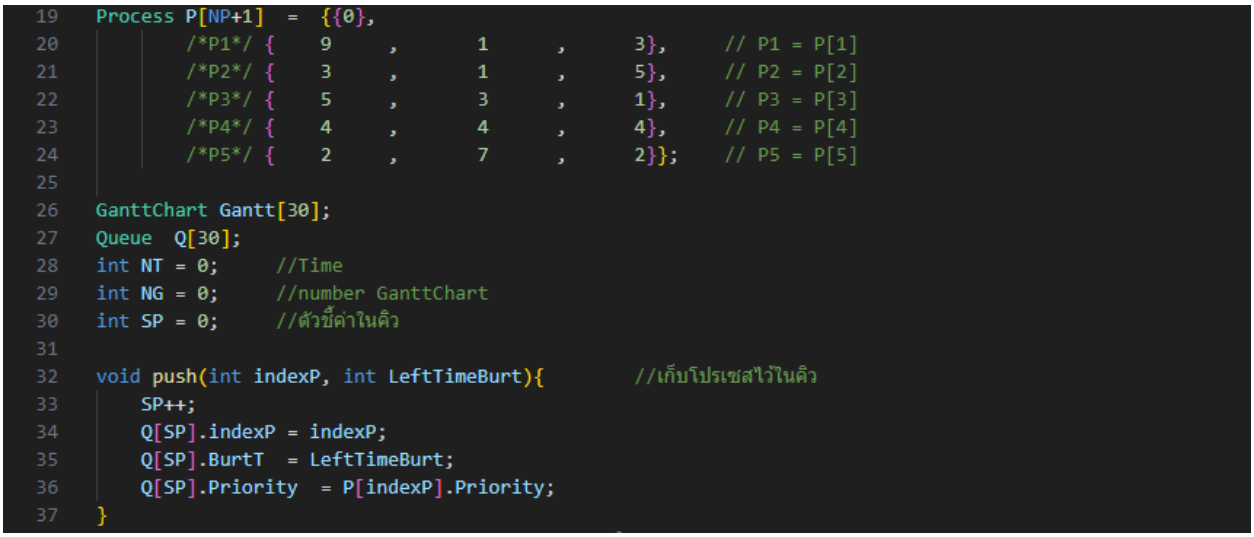
- #include <stdio.h>: เรียกใช้ไลบรารีมาตรฐานสำหรับ input/output functions เช่น printf และ scanf

Macro:

- #define NP 5: กำหนดค่าคงที่ NP เท่ากับ 5 เพื่อแทนจำนวนกระบวนการในโปรแกรม

Structures:

- **GanttChart:** ใช้เก็บข้อมูลสำหรับสร้างแผนภูมิ Gantt
    - indexP: เก็บหมายเลขลำดับของกระบวนการที่ทำงานในช่วงเวลานั้น
    - startP: เก็บเวลาที่กระบวนการเริ่มต้นทำงาน
- **Queue:** ใช้สำหรับเก็บข้อมูลของกระบวนการใน queue เพื่อรอคิวทำงาน
    - indexP: เก็บหมายเลขลำดับของกระบวนการ
    - BurtT: เก็บเวลาทำงาน (burst time) ของกระบวนการ
    - Priority: เก็บลำดับความสำคัญ (priority) ของกระบวนการ
- **Process:** ใช้สำหรับเก็บข้อมูลของกระบวนการแต่ละกระบวนการ
    - BurtT: เก็บเวลาทำงาน (burst time) ของกระบวนการ
    - ArrivalT: เก็บเวลาที่กระบวนการมาถึง (arrival time)
    - Priority: เก็บลำดับความสำคัญ (priority) ของกระบวนการ



```
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;     //ตัวชี้ค่าในคิว

void push(int indexP, int LeftTimeBurt){        //เก็บโปรเซสไว้ในคิว
    SP++;
    Q[SP].indexP = indexP;
    Q[SP].BurtT  = LeftTimeBurt;
    Q[SP].Priority  = P[indexP].Priority;
}
```

ประกาศตัวแปร:

- **Process P[NP+1]:** อาร์เรย์ของโครงสร้าง Process มีขนาด NP+1 เพื่อเก็บข้อมูลของกระบวนการทั้งหมด รวมถึงกระบวนการที่ไม่มีอยู่จริง (dummy process)
- **GanttChart Gantt[30]:** อาร์เรย์ของโครงสร้าง GanttChart มีขนาด 30 เพื่อเก็บข้อมูลสำหรับสร้างแผนภูมิ Gantt
- **Queue Q[30]:** อาร์เรย์ของโครงสร้าง Queue มีขนาด 30 เพื่อเก็บข้อมูลของกระบวนการใน queue เพื่อรอคิวทำงาน
- **int NT, NG, SP:** ตัวแปรสำหรับเก็บข้อมูลเวลาปัจจุบัน (NT), จำนวนช่วงเวลาในแผนภูมิ Gantt (NG) และตัวชี้ค่าในคิว (SP)

ฟังก์ชัน push():

- ฟังก์ชันนี้ใช้สำหรับเก็บกระบวนการไว้ใน queue
- กำหนดค่า SP++ เพื่อเพิ่มค่าตัวชี้ค่าในคิว
- กำหนดค่าให้กับสมาชิกของโครงสร้าง Queue ตามข้อมูลของกระบวนการที่ส่งเข้ามา

```
38  void pop(){                   //นำโปรเซสออกจากคิว
39      if(SP == 0)
40          printf("UNDER FLOW!!!\n");
41      SP--;
42  }
43
44  Queue sortQ(){                //เรียงข้อมูลจากน้อยไปมาก
45      Queue tempP;
46      int j;
47      for (int i = 1; i <SP ; ++i) {
48          j = i+1;
49          if(Q[i].Priority <= Q[j].Priority){
50              tempP = Q[i];
51              Q[i]  = Q[j];
52              Q[j]  = tempP;
53          }
54      }
55      return Q[SP];
56  }
```

ฟังก์ชัน pop():

- ฟังก์ชันนี้ใช้สำหรับนำกระบวนการออกจาก queue
- ตรวจสอบค่า SP ว่ามีค่าเท่ากับ 0 หรือไม่ หากเท่ากับ 0 แสดงว่า queue ว่างเปล่า โปรแกรมจะพิมพ์ข้อความแจ้งเตือน
- กำหนดค่า SP—เพื่อลดค่าตัวชี้ค่าในคิว

ฟังก์ชัน sortQ():

- ฟังก์ชันนี้ใช้สำหรับเรียงลำดับข้อมูลใน queue จากน้อยไปมาก
- ประกาศตัวแปร tempP และ j สำหรับการ swap ข้อมูล
- ทำซ้ำ for loop ตั้งแต่ i = 1 ถึง SP-1
- เปรียบเทียบลำดับความสำคัญของกระบวนการที่ i และ j หากลำดับความสำคัญของกระบวนการที่ i น้อยกว่าหรือเท่ากับลำดับความสำคัญของกระบวนการที่ j
- swap ข้อมูลของกระบวนการที่ i และ j
- คืนค่ากระบวนการที่ SP

```
58  int minPriority(int indexP, int tempP[], int *countP) {  //ในกรณีที่มีโปรเซสเกิดในเวลาเดียวกัน หาค่าโปรเซสที่มีลำดับความ
59      int minPriority = P[indexP].Priority;
60      int minPro = indexP;
61      *countP = 0;
62      for (int i = 1; i <= NP; ++i)
63          if (P[indexP].ArrivalT == P[i].ArrivalT) {
64              tempP[*countP] = i;                    //เก็บโปรเซสที่เกิดในเวลาเดียวกัน
65              *countP+=1;                            //จำนวนโปรเซสที่เกิดในเวลาเดียวกัน
66              if(P[i].Priority < minPriority ){
67                  minPriority = P[i].Priority;
68                  minPro = i;
69              }
70          }
71      return minPro;                                 //โปรเซสที่มีลำดับความสำคัญสูงที่สุด
72  }
```

ฟังก์ชัน minPriority():

- ฟังก์ชันนี้ใช้สำหรับหาค่าโปรเซสที่มีลำดับความสำคัญสูงที่สุด ในกรณีที่มีกระบวนการเกิดขึ้นในเวลาเดียวกัน
- ประกาศตัวแปร minPriority และ minPro สำหรับเก็บค่าลำดับความสำคัญและหมายเลขลำดับของกระบวนการที่มีลำดับความสำคัญสูงที่สุด
- กำหนดค่า *countP = 0 เพื่อเริ่มต้นค่าตัวชี้จำนวนกระบวนการที่เกิดในเวลาเดียวกัน
- ทำซ้ำ for loop ตั้งแต่ i = 1 ถึง NP
- ตรวจสอบว่ากระบวนการที่ i เกิดขึ้นในเวลาเดียวกันกับกระบวนการที่ indexP
- หากใช่ เก็บหมายเลขลำดับของกระบวนการที่ i ไว้ในอาร์เรย์ tempP และเพิ่มค่าตัวชี้จำนวนกระบวนการที่เกิดในเวลาเดียวกัน
- หากลำดับความสำคัญของกระบวนการที่ i ต่ำกว่าลำดับความสำคัญของกระบวนการที่มีลำดับความสำคัญสูงที่สุดในปัจจุบัน
- อัปเดตค่าลำดับความสำคัญและหมายเลขลำดับของกระบวนการที่มีลำดับความสำคัญสูงที่สุด
- คืนค่าหมายเลขลำดับของกระบวนการที่มีลำดับความสำคัญสูงที่สุด

รายละเอียดเพิ่มเติม:

- ฟังก์ชัน minPriority() ใช้อาร์เรย์ tempP เพื่อเก็บหมายเลขลำดับของกระบวนการที่เกิดในเวลาเดียวกัน และใช้ตัวชี้จำนวนกระบวนการที่เกิดในเวลาเดียวกัน (*countP) เพื่อนับจำนวนกระบวนการที่เกิดในเวลาเดียวกัน
- ฟังก์ชันจะคืนค่าหมายเลขลำดับของกระบวนการที่มีลำดับความสำคัญสูงที่สุด ในกรณีที่มีกระบวนการเกิดในเวลาเดียวกันหลายกระบวนการ

```
74  void pushSynchronous(int indexP, int tempP[], int countP) {//เก็บโปรเซสที่เกิดพร้อมกันไว้ในคิว(ในเวลาเดียวกัน)
75      for (int i = 0; i < countP ; ++i)
76          if(tempP[i] != indexP )                          //ยกเว้นโปรเซสที่มีลำดับความสำคัญสูงที่สุด
77              push(tempP[i],P[tempP[i]].BurtT);
78  }
79
80  void getData(int indexP, int time_i){
81      Gantt[NG].indexP = indexP;          //เก็บโปรเซสเพื่อทำ GanttChart
82      Gantt[NG].startP = time_i;          //เก็บเวลาเริ่มทำงานโปรเซสเพื่อทำ GanttChart
83      NG++;
84  }
```

ฟังก์ชัน pushSynchronous():

- ฟังก์ชันนี้ใช้สำหรับเก็บกระบวนการที่เกิดพร้อมกันไว้ใน queue
- ทำซ้ำ for loop ตั้งแต่ i = 0 ถึง countP
- ตรวจสอบว่ากระบวนการที่ i เกิดในเวลาเดียวกันกับกระบวนการที่ indexP หรือไม่
- หากไม่ใช่ เก็บกระบวนการที่ i ไว้ใน queue

ฟังก์ชัน getData():

- ฟังก์ชันนี้ใช้สำหรับเก็บข้อมูลของกระบวนการเพื่อทำแผนภูมิ Gantt
- กำหนดค่าให้กับสมาชิกของโครงสร้าง GanttChart ตามข้อมูลของกระบวนการที่ส่งเข้ามา

รายละเอียดเพิ่มเติม:

- ฟังก์ชัน pushSynchronous() จะใช้ในกรณีที่มีกระบวนการเกิดในเวลาเดียวกันหลายกระบวนการ โดยฟังก์ชันจะเก็บกระบวนการทั้งหมดที่เกิดในเวลาเดียวกันยกเว้นกระบวนการที่มีลำดับความสำคัญสูงที่สุด
- ฟังก์ชัน getData() จะใช้สำหรับเก็บข้อมูลของกระบวนการที่ทำงานในแต่ละช่วงเวลาเพื่อทำแผนภูมิ Gantt โดยฟังก์ชันจะเก็บหมายเลขลำดับของกระบวนการและเวลาเริ่มทำงานของกระบวนการ

```
86  void P_Priority(){
87      int indexP = 0, ENDPro = 0, runningP = 0;
88      int tempP[NP], countP = 0;
89      for (int i = 0; i <= NT; ++i) {              //i แทนเวลา (Time)
90          for (int j = 1; j <= NP; ++j) {
91              if(i == P[j].ArrivalT){              //ณ เวลาที่ i มีโปรเซส 1 2 3 ...N เกิดขึ้นใหม่
92                  indexP = minPriority(j,tempP,&countP);
93                  if(i >= ENDPro){      //ณ เวลาที่ i ถ้าไม่มีโปรเซสใดกำลังทำงาน(ไม่มีการใช้ทรัพยากร)
94                      pushSynchronous(indexP,tempP,countP);//ถ้ามีโปรเซสเกิดขึ้นพร้อกัน(ในเวลาเดียวกัน)
95                      ENDPro = i + P[indexP].BurtT;//เวลาที่โปรเซสจะจบการทำงาน
96                      runningP = indexP;           //โปรเซสที่กำลังทำงาน
97                      getData(indexP,i);           //เก็บข้อมูลโปรเซส
98                  }else{                           //ณ เวลาที่ i ถ้ามีโปรเซสใดกำลังทำงาน(มีการใช้ทรัพยากร) หรือ มีโปรเซสออยู่ใ
99                      if(P[indexP].Priority < (P[runningP].Priority) && i < ENDPro){//ให้เช็คว่าโปรเซสที่เกิดใหม่มีลำดับควา
100                         pushSynchronous(indexP,tempP,countP);
101                         push(runningP, (P[runningP].BurtT - i));  //เก็บโปรเซสที่กำลังทำงานไว้ในคิว และเวลาทำงานที่เหลือ
102                         ENDPro = i + P[indexP].BurtT;
103                         runningP = indexP;                    //โปรเซสเกิดใหม่ที่มีลำดับความสำคัญสูงกว่า เริ่มทำงาน
104                         getData(indexP, i);
105                     }else {
106                         push(indexP, P[indexP].BurtT);//เก็บค่าโปรเซสที่เกิดใหม่ ณ เวลาที่ i ไว้ในคิว
107                         pushSynchronous(indexP, tempP, countP);
108                     }
109                 }
110                 break;
111             }
112         }
```

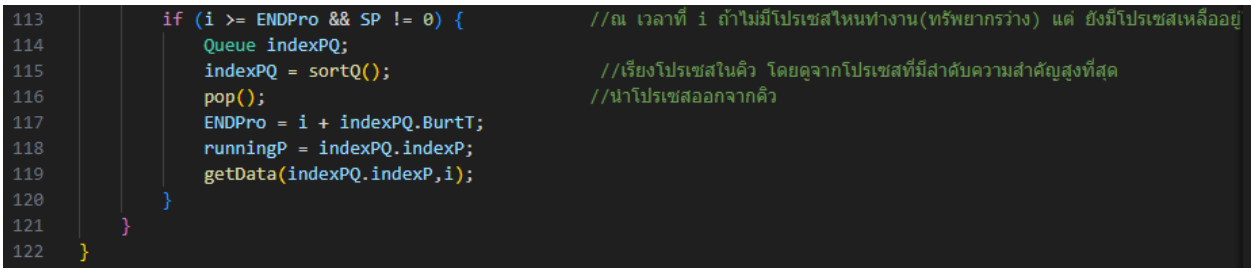
ฟังก์ชัน P_Priority():

- ฟังก์ชันนี้ใช้สำหรับจำลองการ scheduling แบบ Priority
- ประกาศตัวแปร indexP, ENDPro, runningP, tempP[NP] และ countP สำหรับเก็บค่าต่างๆ ที่จำเป็น
- ทำซ้ำ for loop ตั้งแต่ i = 0 ถึง NT
- ทำซ้ำ for loop ตั้งแต่ j = 1 ถึง NP
- ตรวจสอบว่ากระบวนการที่ j มาถึง (arrived) ในเวลา i หรือไม่
- หากใช่ เรียกใช้ฟังก์ชัน minPriority() เพื่อหากระบวนการที่มีลำดับความสำคัญสูงที่สุดที่มาถึงในเวลา i
- ตรวจสอบว่าไม่มีกระบวนการที่กำลังทำงานอยู่หรือไม่
- หากไม่มีกระบวนการที่กำลังทำงานอยู่
    - เรียกใช้ฟังก์ชัน pushSynchronous() เพื่อเก็บกระบวนการที่พบลงใน queue
    - กำหนดค่า ENDPro เป็นเวลาที่กระบวนการจะจบการทำงาน
    - กำหนดค่า runningP เป็นหมายเลขลำดับของกระบวนการ
    - เรียกใช้ฟังก์ชัน getData() เพื่อเก็บข้อมูลของกระบวนการ
- หากมีกระบวนการที่กำลังทำงานอยู่
    - ตรวจสอบว่ากระบวนการที่พบมีลำดับความสำคัญสูงกว่ากระบวนการที่กำลังทำงานอยู่หรือไม่
    - หากใช่

        - เรียกใช้ฟังก์ชัน pushSynchronous() เพื่อเก็บกระบวนการที่กำลังทำงานอยู่ลงใน queue
        - เรียกใช้ฟังก์ชัน push() เพื่อเก็บกระบวนการที่พบลงใน queue
        - กำหนดค่า ENDPro เป็นเวลาที่กระบวนการที่พบจะจบการทำงาน
        - กำหนดค่า runningP เป็นหมายเลขลำดับของกระบวนการ
        - เรียกใช้ฟังก์ชัน getData() เพื่อเก็บข้อมูลของกระบวนการ
    - หากไม่ใช่
        - เรียกใช้ฟังก์ชัน push() เพื่อเก็บกระบวนการที่พบลงใน queue
        - เรียกใช้ฟังก์ชัน pushSynchronous() เพื่อเก็บกระบวนการที่พบลงใน queue

รายละเอียดเพิ่มเติม:

- ฟังก์ชัน minPriority() จะหากระบวนการที่มีลำดับความสำคัญสูงที่สุดที่มาถึงในเวลา i โดยพิจารณากระบวนการทั้งหมดที่เกิดในเวลาเดียวกันด้วย
- ฟังก์ชัน pushSynchronous() จะเก็บกระบวนการที่เกิดพร้อมกันไว้ใน queue โดยยกเว้นกระบวนการที่มีลำดับความสำคัญสูงที่สุด
- ฟังก์ชัน getData() จะเก็บข้อมูลของกระบวนการที่ทำงานในแต่ละช่วงเวลาเพื่อทำแผนภูมิ Gantt



```
113         if (i >= ENDPro && SP != 0) {            //ณ เวลาที่ i ถ้าไม่มีโปรเซสไหนทำงาน(ทรัพยากรว่าง) แต่ ยังมีโปรเซสเหลืออยู่
114             Queue indexPQ;
115             indexPQ = sortQ();                    //เรียงโปรเซสในคิว โดยดูจากโปรเซสที่มีลำดับความสำคัญสูงที่สุด
116             pop();                               //นำโปรเซสออกจากคิว
117             ENDPro = i + indexPQ.BurtT;
118             runningP = indexPQ.indexP;
119             getData(indexPQ.indexP,i);
120         }
121     }
122 }
```

เงื่อนไข:

- เงื่อนไข i >= ENDPro && SP != 0 หมายความว่า ในเวลา i กระบวนการที่กำลังทำงานอยู่ (runningP) ได้ทำงานเสร็จแล้ว และยังมีกระบวนการเหลืออยู่ใน queue

การกระทำ:

- เรียกใช้ฟังก์ชัน sortQ() เพื่อเรียงลำดับกระบวนการใน queue โดยดูจากกระบวนการที่มีลำดับความสำคัญสูงที่สุด
- เรียกใช้ฟังก์ชัน pop() เพื่อนำกระบวนการที่มีลำดับความสำคัญสูงสุดออกจาก queue
- กำหนดค่า ENDPro เป็นเวลาที่กระบวนการที่ออกมาจาก queue จะจบการทำงาน
- กำหนดค่า runningP เป็นหมายเลขลำดับของกระบวนการที่ออกมาจาก queue
- เรียกใช้ฟังก์ชัน getData() เพื่อเก็บข้อมูลของกระบวนการ

รายละเอียดเพิ่มเติม:

- เงื่อนไข i >= ENDPro && SP != 0 เป็นการรับประกันว่า ไม่มีกระบวนการที่กำลังทำงานอยู่ (runningP) ณ เวลาที่ i
- ฟังก์ชัน sortQ() จะเรียงลำดับกระบวนการใน queue โดยดูจากลำดับความสำคัญ โดยกระบวนการที่มีลำดับความสำคัญสูงที่สุดจะอยู่ด้านบนของ queue
- ฟังก์ชัน pop() จะนำกระบวนการที่อยู่บนสุดของ queue ออกจาก queue
- ฟังก์ชัน getData() จะเก็บข้อมูลของกระบวนการที่ทำงานในแต่ละช่วงเวลาเพื่อทำแผนภูมิ Gantt

```
124  float waitProcess(int indexP){                              // คำนวณหา เวลารอของโปรเซสที่ i
125      int count = 0;
126      float waitT =0, end = 0;
127      for (int i = 0; i <NG ; i++) {
128          if(Gantt[i].indexP == indexP){
129              if(count == 0){                                 // เวลาโปรเซสได้เข้าทำงาน - เวลาเกิดของโปรเซส
130                  waitT += (float)Gantt[i].startP - P[indexP].ArrivalT;
131                  end    = (float)Gantt[i+1].startP;
132                  count++;
133              }else{
134                  waitT += (float)Gantt[i].startP - end;       // เวล่ที่โปรเซสได้เข้าทำงานอีกครั้ง - เวลาที่โปรเซสจบการทำงานค
135                  end    = (float)Gantt[i+1].startP;
136              }
137          }
138      }
139      return waitT;
140  }
```

ฟังก์ชัน waitProcess():

- ฟังก์ชันนี้ใช้สำหรับคำนวณหาเวลารอของกระบวนการที่ระบุด้วยหมายเลขลำดับ indexP
- ประกาศตัวแปร count และ end สำหรับเก็บค่าต่างๆ ที่จำเป็น

การคำนวณเวลารอ:

- ทำซ้ำ for loop ตั้งแต่ i = 0 ถึง NG
- ตรวจสอบว่ากระบวนการที่ i มีหมายเลขลำดับเท่ากับ indexP หรือไม่
- หากใช่
    - ตรวจสอบว่านับเป็นครั้งแรกหรือไม่
        - หากนับเป็นครั้งแรก
            - คำนวณเวลารอเป็นเวลาที่กระบวนการได้เข้าทำงาน - เวลาเกิดของกระบวนการ
            - กำหนดค่า end เป็นเวลาที่กระบวนการถัดไปจะเริ่มทำงาน
            - เพิ่มค่า count ขึ้น 1
    - หากไม่ใช่
        - คำนวณเวลารอเป็นเวลาที่กระบวนการได้เข้าทำงานอีกครั้ง – เวลาที่กระบวนการจบการทำงานครั้งก่อน
        - กำหนดค่า end เป็นเวลาที่กระบวนการถัดไปจะเริ่มทำงาน
- เพิ่มค่า waitT ด้วยเวลารอที่คำนวณได้

รายละเอียดเพิ่มเติม:

- ฟังก์ชัน waitProcess() คำนวณหาเวลารอของกระบวนการโดยพิจารณาเวลาทำงานของกระบวนการทั้งหมดในแผนภูมิ Gantt

```
142  void calNT(){                                          //คำนวณหาผลรวมของ burt time
143      int sumBurt = 0;
144      int minArrival = P[1].ArrivalT;
145      for (int i = 1; i <= NP; ++i) {
146          if(P[i].ArrivalT < minArrival){
147              minArrival = P[i].ArrivalT;
148          }
149          sumBurt += P[i].BurtT;
150      }
151      NT = minArrival + sumBurt;                         //เวลาที่เริ่มเกิดโปรเซสตัวแรก + ผลรวมเวลาที่ใช้ในการทำงานของโปร
152  }
```

ฟังก์ชัน calNT():

- ฟังก์ชันนี้ใช้สำหรับคำนวณหาผลรวมของ BurtT ของทุกกระบวนการ
- ประกาศตัวแปร sumBurt และ minArrival สำหรับเก็บค่าต่างๆ ที่จำเป็น

การคำนวณผลรวมของ BurtT:

- กำหนดค่า minArrival เป็นเวลาที่กระบวนการแรกเริ่มทำงาน
- ทำซ้ำ for loop ตั้งแต่ i = 1 ถึง NP
    - ตรวจสอบว่าเวลาที่กระบวนการ i เริ่มทำงานน้อยกว่า minArrival หรือไม่
        - หากใช่
            - กำหนดค่า minArrival เป็นเวลาที่กระบวนการ i เริ่มทำงาน
    - เพิ่มค่า sumBurt ด้วย BurtT ของกระบวนการ i

การกำหนดค่า NT:

- กำหนดค่า NT เป็นเวลาที่กระบวนการแรกเริ่มทำงาน + ผลรวมของ BurtT ของทุกกระบวนการ

รายละเอียดเพิ่มเติม:

- ฟังก์ชัน calNT() จำเป็นสำหรับการกำหนดจำนวนช่วงเวลาทั้งหมด (NT) ในแผนภูมิ Gantt

```c
int main(){
    calNT();
    printf("# Miss Ratchaneekorn Chuadee ID:65543206077-1 Sec 02\n");
    printf("# OUTPUT LAB6 CPU Scheduling\n");
    printf("# Priority (SJF Preemptive) \n");
    printf("Sequence process is :");
    P_Priority();
    for (int 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 (int i = 1; i <= NP; i++) {
        printf("| P%-10d", i);
    }
    printf("\n");
    float sum = 0, avgTime;
    for (int 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 (int i = 1; i <= NP; i++) {
        printf("|P%d = %-6.2fms ", i, waitProcess(i) + P[i].BurtT);
    }
    printf("\n-----------------------------------------------------------\n");
    return 0;
}
```

ฟังก์ชัน main():

- เรียกใช้ฟังก์ชัน calNT() เพื่อคำนวณหาผลรวมของ BurtT ของทุกกระบวนการ
- พิมพ์ข้อมูลเบื้องต้น เช่น ชื่อ รหัสนักศึกษา หัวข้อแล็บ
- พิมพ์ข้อความ "Sequence process is :"
- เรียกใช้ฟังก์ชัน P_Priority() เพื่อจำลองการ scheduling แบบ Priority
- พิมพ์ลำดับการทำงานของกระบวนการโดยอ้างอิงจากแผนภูมิ Gantt
- พิมพ์ข้อมูลเวลารอของกระบวนการแต่ละกระบวนการ
- คำนวณหาเวลารอเฉลี่ยของกระบวนการทั้งหมด
- พิมพ์ข้อมูล turnaround time ของกระบวนการแต่ละกระบวนการ

รายละเอียดเพิ่มเติม:

- ฟังก์ชัน main() เป็นจุดเริ่มต้นของโปรแกรม
- ฟังก์ชัน calNT() จำเป็นสำหรับการกำหนดจำนวนช่วงเวลาทั้งหมด (NT) ในแผนภูมิ Gantt
- ฟังก์ชัน P_Priority() จำลองการ scheduling แบบ Priority
- ฟังก์ชัน waitProcess() คำนวณหาเวลารอของกระบวนการ

ผลลัพธ์

```
# Miss Ratchaneekorn Chuadee ID:65543206077-1 Sec 02
# OUTPUT LAB6 CPU Scheduling
# Priority (SJF Preemptive)
Sequence process is :P1->P3->P5->P1->P4->P2
------------------------------------------------------------
Wait time of process (millisecond)
| P1         | P2         | P3         | P4         | P5
| 7.00       | 19.00      | 0.00       | 12.00      | 1.00
Average time is 7.80
Turnaround time
|P1 = 16.00 ms |P2 = 22.00 ms |P3 = 5.00  ms |P4 = 16.00 ms |P5 = 3.00  ms
------------------------------------------------------------
PS D:\เทอม 2 66\OS_Lab\lab6>
```

ลำดับการทำงานของกระบวนการ:

- P1 -> P3 -> P5 -> P1 -> P4 -> P2

เวลารอของกระบวนการ:

- P1: 7.00 ms
- P2: 19.00 ms
- P3: 0.00 ms
- P4: 12.00 ms
- P5: 1.00 ms
- เวลารอเฉลี่ย: 7.80 ms

Turnaround time:

- P1: 16.00 ms
- P2: 22.00 ms
- P3: 5.00 ms
- P4: 16.00 ms
- P5: 3.00 ms

ผลลัพธ์แสดงให้เห็นว่าลำดับการทำงานของกระบวนการเป็นไปตามการจัดลำดับความสำคัญ (Priority)
กระบวนการ P3 มีเวลารอน้อยที่สุด (0.00 ms) เนื่องจากมีลำดับความสำคัญสูงที่สุด
กระบวนการ P2 มีเวลารอมากที่สุด (19.00 ms) อาจเป็นเพราะมีลำดับความสำคัญต่ำที่สุด หรือเกิดหลังจากกระบวนการอื่น
เวลารอเฉลี่ยของกระบวนการทั้งหมดคือ 7.80 ms
Turnaround time ของกระบวนการแต่ละกระบวนการแตกต่างกันไปขึ้นอยู่กับเวลารอและเวลาดำเนินการ (BurtT)

สรุปผลการทดลอง

จากการทดลองทั้ง 4 แบบ พบว่า

- ลำดับการทำงานของกระบวนการขึ้นอยู่กับการจัดลำดับความสำคัญ (Priority) หรือเวลาดำเนินการ (BurtT) ของกระบวนการ
- เวลารอของกระบวนการจะลดลงเมื่อจำนวนกระบวนการที่มีลำดับความสำคัญสูง หรือมีเวลาดำเนินการสั้นเพิ่มขึ้น
- Turnaround time ของกระบวนการแต่ละกระบวนการแตกต่างกันไปขึ้นอยู่กับเวลารอและเวลาดำเนินการ (BurtT)

โดยสรุปแล้ว ประเภทของการ scheduling ที่เหมาะสมจะขึ้นอยู่กับปัจจัยต่างๆ เช่น ลำดับความสำคัญและเวลาดำเนินการของกระบวนการ

สื่อ / เอกสารอ้างอิง

อาจารย์ปิยพล ยืนยงสถาวร: เอกสารประกอบการสอน 6 CPU Scheduling