HỆ ĐIỀU HÀNH
BÁO CÁO LAB 4 - GIẢI THUẬT LẬP LỊCH CPU

Họ và tên: Nguyễn Trọng Tất Thành
Mã số sinh viên: 23521455
Lớp: IT007.P11.CTTN.1

Mục lục

CHECKLIST

- **SJF - Shortest Job First**
  - ✓ Vẽ lưu đồ giải thuật
  - ✓ Chạy tay với test case 5 tiến trình
  - ✓ Mã nguồn SJF với Arrival Time và Burst Time ngẫu nhiên
  - ✓ Chạy thử nghiệm và so sánh với chạy tay
- **SRTF - Shortest Remaining Time First**
  - ✓ Vẽ lưu đồ giải thuật
  - ✓ Chạy tay với test case 5 tiến trình
  - ✓ Mã nguồn SRTF với Arrival Time và Burst Time ngẫu nhiên
  - ✓ Chạy thử nghiệm và so sánh với chạy tay

1 GIỚI THIỆU

Trong hệ điều hành, lập lịch tiến trình là một trong những nhiệm vụ quan trọng giúp quản lý việc thực thi các tiến trình nhằm tối ưu hóa tài nguyên và đảm bảo hiệu suất hệ thống. Để thực hiện việc này, hệ điều hành sử dụng các giải thuật lập lịch CPU nhằm quyết định tiến trình nào sẽ được CPU thực thi tiếp theo, dựa trên các tiêu chí cụ thể.

Báo cáo này tập trung vào ba giải thuật lập lịch phổ biến, bao gồm:

- **SJF (Shortest Job First)**: Giải thuật chọn tiến trình có thời gian thực thi ngắn nhất để thực thi tiếp theo, nhằm giảm thời gian chờ trung bình. Tuy nhiên, giải thuật này có thể gây ra hiện tượng "starvation" đối với các tiến trình dài.
- **SRTF (Shortest Remaining Time First)**: Là phiên bản tiền chế của SJF, trong đó tiến trình có thời gian thực thi còn lại ngắn nhất sẽ được ưu tiên thực thi. Giải thuật này phù hợp cho các hệ thống yêu cầu tối ưu hóa thời gian chờ đợi nhưng khó cài đặt hơn và dễ dẫn đến tình trạng "starvation" cho các tiến trình dài.
- **RR (Round Robin)**: Giải thuật thực hiện luân phiên các tiến trình theo một khoảng thời gian cố định (quantum time), đảm bảo tính công bằng cho tất cả các tiến trình và phù hợp với các hệ thống chia sẻ thời gian.

Mục tiêu của báo cáo

Báo cáo này nhằm hiện thực và kiểm chứng tính đúng đắn của các giải thuật lập lịch CPU thông qua các bước sau:

- **Vẽ lưu đồ giải thuật**: Trình bày trực quan cách thức hoạt động của mỗi giải thuật thông qua lưu đồ.
- **Mô phỏng chạy tay với test case** gồm 5 tiến trình: Kiểm tra từng giải thuật với test case cụ thể để đảm bảo tính đúng đắn của lý thuyết.
- **Cài đặt mã nguồn C++**: Viết mã nguồn cho từng giải thuật, với 'Arrival Time' và 'Burst Time' của các tiến trình được tạo ngẫu nhiên trong phạm vi cho trước.
- **Thực hiện kiểm thử với 3 test case**: Chạy mã nguồn trên 3 test case khác nhau, so sánh kết quả giữa chương trình và tính toán chạy tay để đảm bảo tính chính xác của mã nguồn.

Bằng cách thực hiện các bước trên, báo cáo không chỉ giúp hiểu sâu hơn về các giải thuật lập lịch mà còn cung cấp cái nhìn trực quan về hiệu suất và ứng dụng của mỗi giải thuật trong hệ thống thực tế.

2 GIẢI THUẬT LẬP LỊCH CPU

2.1 SJF - Shortest Job First

2.1.1 Mô tả giải thuật và lưu đồ

Giải thích giải thuật SJF (Shortest Job First):

Giải thuật SJF là một giải thuật lập lịch không tiền chế (non-preemptive), trong đó CPU sẽ chọn tiến trình có thời gian thực thi ngắn nhất (Burst Time) để thực thi trước. Mục tiêu của giải thuật này là giảm thời gian chờ trung bình cho tất cả các tiến trình trong hệ thống.

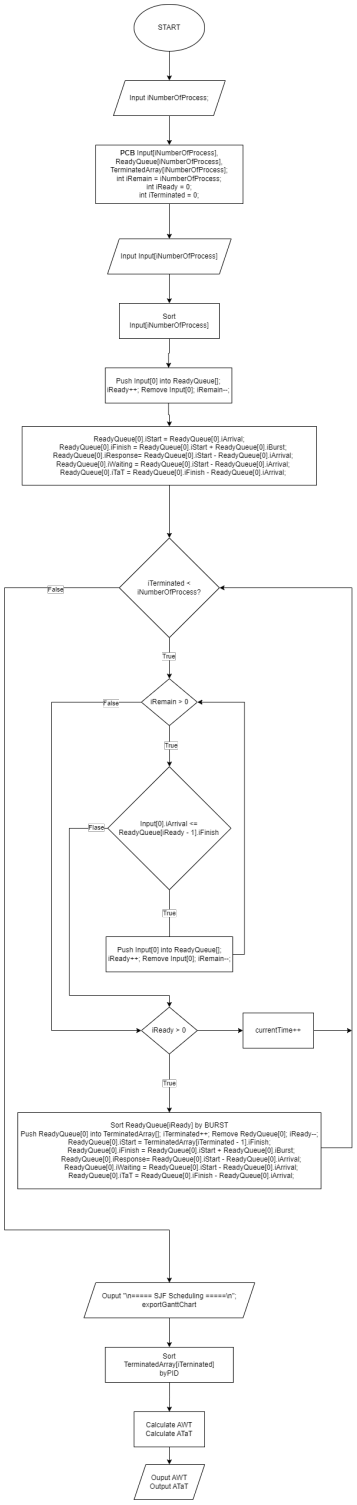
Nguyên tắc hoạt động của SJF:

- Khi CPU rảnh, nó sẽ kiểm tra danh sách các tiến trình đang chờ xử lý.
- Tiến trình có thời gian thực thi ngắn nhất sẽ được chọn để thực thi.
- Nếu hai tiến trình có thời gian thực thi bằng nhau, tiến trình đến trước sẽ được chọn (giống với nguyên tắc của FCFS).
- Giải thuật SJF sẽ tiếp tục chọn tiến trình ngắn nhất tiếp theo sau khi hoàn tất tiến trình hiện tại.

Ưu điểm và nhược điểm:

- **Ưu điểm**: Giảm thời gian chờ trung bình cho các tiến trình, do đó cải thiện hiệu suất hệ thống so với FCFS.
- **Nhược điểm**: Có thể gây ra hiện tượng "starvation" cho các tiến trình có thời gian thực thi dài, do các tiến trình ngắn liên tục được ưu tiên.

Lưu đồ giải thuật SJF:



Hình 1: Lưu đồ giải thuật

2.1.2 Chạy tay cho test case

Chạy tay với test case gồm 5 tiến trình:

Giả sử chúng ta có 5 tiến trình với các thông số như sau:

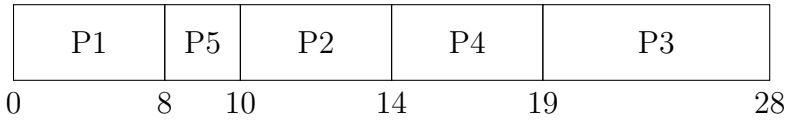
Tiến trình	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5
P5	4	2

Bảng 1: Test case cho giải thuật SJF với 5 tiến trình

Các bước thực hiện giải thuật SJF:

- Tại thời điểm 0, chỉ có tiến trình P1 đến, nên P1 được thực thi đầu tiên.
- Sau khi P1 hoàn thành (tại thời điểm 8), các tiến trình P2, P3, P4, và P5 đã đến. SJF chọn tiến trình có 'Burst Time' ngắn nhất, tức là P5.
- Sau khi P5 hoàn thành, các tiến trình còn lại là P2, P3, và P4. SJF tiếp tục chọn tiến trình có 'Burst Time' ngắn nhất, tức là P2.
- Tiếp theo, các tiến trình còn lại là P3 và P4. SJF chọn P4 vì P4 có 'Burst Time' ngắn hơn.
- Cuối cùng, P3 là tiến trình duy nhất còn lại và sẽ được thực thi.

Gantt Chart cho giải thuật SJF:



Hình 2: Gantt Chart cho giải thuật SJF

Kết quả chạy tay:

Dựa trên các bước trên, chúng ta có bảng thời gian thực thi cho từng tiến trình:

Tiến trình	Arrival Time	Burst Time	Start Time	Finish Time	Waiting Time	Turnaround Time
P1	0	8	0	8	0	8
P5	4	2	8	10	4	6
P2	1	4	10	14	9	13
P4	3	5	14	19	11	16
P3	2	9	19	28	17	26

Bảng 2: Kết quả chạy tay giải thuật SJF với test case 5 tiến trình

Tính toán thời gian trung bình:

- **Average Waiting Time** $= \frac{0+4+9+11+17}{5} = 8.2$
- **Average Turnaround Time** $= \frac{8+6+13+16+26}{5} = 13.8$

Kết luận: Qua kết quả tính toán tay và Gantt Chart, chúng ta thấy rằng giải thuật SJF đã tối ưu hóa thời gian chờ trung bình cho các tiến trình trong hệ thống. Thời gian chờ và thời gian quay vòng trung bình thấp cho thấy hiệu quả của SJF trong việc giảm thiểu độ trễ cho các tiến trình có thời gian thực thi ngắn.

2.1.3 Mã nguồn

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

#define SORT_BY_ARRIVAL 0
#define SORT_BY_PID 1
#define SORT_BY_BURST 2
#define SORT_BY_START 3

typedef struct
{
    int iPID;
    int iArrival;
    int iBurst;
    int iStart;
    int iFinish;
    int iWaiting;
    int iResponse;
    int iTaT;
} PCB;

void inputProcess(int n, PCB P[])
{
    for (int i = 0; i < n; i++)
```

```
26      {
27          std::cout << "Enter Arrival Time for P[" << i << "]:
                ";
28          std::cin >> P[i].iArrival;
29
30          std::cout << "Enter Burst Time for P[" << i << "]: ";
31          std::cin >> P[i].iBurst;
32
33          P[i].iPID = i;
34      }
35  }
36
37  void printProcess(int n, PCB P[])
38  {
39      std::cout << std::setw(5) << "PID"
40                << std::setw(10) << "Arrival"
41                << std::setw(10) << "Burst"
42                << std::setw(10) << "Start"
43                << std::setw(10) << "Finish"
44                << std::setw(10) << "Waiting"
45                << std::setw(10) << "Response"
46                << std::setw(10) << "TaT" << std::endl;
47
48      for (int i = 0; i < n; i++)
49      {
50          std::cout << std::setw(5) << P[i].iPID
51                    << std::setw(10) << P[i].iArrival
52                    << std::setw(10) << P[i].iBurst
53                    << std::setw(10) << P[i].iStart
54                    << std::setw(10) << P[i].iFinish
55                    << std::setw(10) << P[i].iWaiting
56                    << std::setw(10) << P[i].iResponse
57                    << std::setw(10) << P[i].iTaT << std::endl;
58      }
59  }
60
61  void exportGanttChart(int n, PCB P[])
62  {
63      std::cout << "Gantt Chart\n|";
64      for (int i = 0; i < n; i++)
65      {
66          std::cout << " P[" << P[i].iPID << "] (" << P[i].
                iStart << "-" << P[i].iFinish << ") |";
67      }
68      std::cout << "\n";
69  }
70
71  void pushProcess(int *n, PCB P[], PCB Q)
72  {
```

```
73      P[*n] = Q;
74      (*n)++;
75  }
76
77  void removeProcess(int *n, int index, PCB P[])
78  {
79      for (int i = index; i < *n - 1; i++)
80      {
81          P[i] = P[i + 1];
82      }
83      (*n)--;
84  }
85
86  void swapProcess(PCB *P, PCB *Q)
87  {
88      PCB temp = *P;
89      *P = *Q;
90      *Q = temp;
91  }
92
93  int partition(PCB P[], int low, int high, int iCriteria)
94  {
95      int pivot = (iCriteria == SORT_BY_BURST) ? P[high].iBurst
            : P[high].iArrival;
96      int i = low - 1;
97
98      for (int j = low; j < high; j++)
99      {
100         bool condition = (iCriteria == SORT_BY_ARRIVAL && P[j
                ].iArrival < pivot) ||
101                          (iCriteria == SORT_BY_PID && P[j].
                             iPID < pivot) ||
102                          (iCriteria == SORT_BY_BURST && P[j].
                             iBurst < pivot) ||
103                          (iCriteria == SORT_BY_START && P[j].
                             iStart < pivot);
104
105         if (condition)
106         {
107             i++;
108             swapProcess(&P[i], &P[j]);
109         }
110     }
111     swapProcess(&P[i + 1], &P[high]);
112     return i + 1;
113 }
114
115 void quickSort(PCB P[], int low, int high, int iCriteria)
116 {
```

```
117     if (low < high)
118     {
119         int pi = partition(P, low, high, iCriteria);
120         quickSort(P, low, pi - 1, iCriteria);
121         quickSort(P, pi + 1, high, iCriteria);
122     }
123 }
124 
125 void calculateAWT(int n, PCB P[])
126 {
127     int iTotalWaiting = 0;
128     for (int i = 0; i < n; i++)
129     {
130         iTotalWaiting += P[i].iWaiting;
131     }
132     std::cout << "Average Waiting Time: " << (double)
        iTotalWaiting / n << "\n";
133 }
134 
135 void calculateATaT(int n, PCB P[])
136 {
137     int iTotalTaT = 0;
138     for (int i = 0; i < n; i++)
139     {
140         iTotalTaT += P[i].iTaT;
141     }
142     std::cout << "Average Turnaround Time: " << (double)
        iTotalTaT / n << "\n";
143 }
144 
145 int main()
146 {
147     int iNumberOfProcess = 0;
148     std::cout << "Please input number of processes: ";
149     std::cin >> iNumberOfProcess;
150 
151     PCB Input[iNumberOfProcess];
152     PCB ReadyQueue[iNumberOfProcess];
153     PCB TerminatedArray[iNumberOfProcess];
154     int iRemain = iNumberOfProcess, iReady = 0, iTerminated =
        0;
155 
156     inputProcess(iNumberOfProcess, Input);
157     quickSort(Input, 0, iNumberOfProcess - 1, SORT_BY_ARRIVAL
        );
158 
159     int currentTime = 0;
160 
161     while (iTerminated < iNumberOfProcess)
```

```
162     {
163         while (iRemain > 0 && Input[0].iArrival <=
                currentTime) {
164             pushProcess(&iReady, ReadyQueue, Input[0]);
165             removeProcess(&iRemain, 0, Input);
166         }
167 
168         if (iReady > 0)
169         {
170             quickSort(ReadyQueue, 0, iReady - 1,
                    SORT_BY_BURST);
171 
172             ReadyQueue[0].iStart = currentTime;
173             ReadyQueue[0].iFinish = ReadyQueue[0].iStart +
                    ReadyQueue[0].iBurst;
174             ReadyQueue[0].iResponse = ReadyQueue[0].iStart -
                    ReadyQueue[0].iArrival;
175             ReadyQueue[0].iWaiting = ReadyQueue[0].iResponse;
176             ReadyQueue[0].iTaT = ReadyQueue[0].iFinish -
                    ReadyQueue[0].iArrival;
177             currentTime = ReadyQueue[0].iFinish;
178 
179             pushProcess(&iTerminated, TerminatedArray,
                    ReadyQueue[0]);
180             removeProcess(&iReady, 0, ReadyQueue);
181         }
182         else
183         {
184             currentTime++;
185         }
186     }
187 
188     std::cout << "\n===== SJF Scheduling =====\n";
189     exportGanttChart(iTerminated, TerminatedArray);
190 
191     quickSort(TerminatedArray, 0, iTerminated - 1,
            SORT_BY_PID);
192 
193     calculateAWT(iTerminated, TerminatedArray);
194     calculateATaT(iTerminated, TerminatedArray);
195 
196     return 0;
197 }
```

Listing 1: Mã nguồn SJF

2.1.4 Kết quả thử nghiệm và so sánh

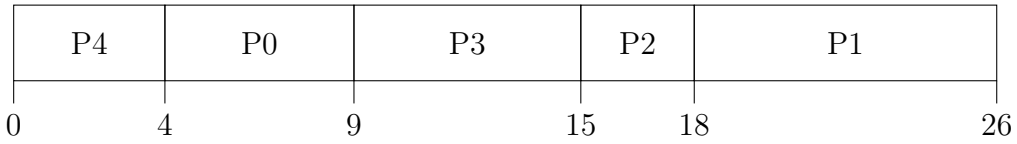
Test Case 1

- **Thông tin tiến trình:**

PID	Arrival Time	Burst Time	Start Time	Finish Time	Waiting Time	Response Time	Turnaround Time
P0	4	5	4	9	0	0	5
P1	2	8	18	26	16	16	24
P2	12	3	15	18	3	3	6
P3	1	6	9	15	8	8	14
P4	0	4	0	4	0	0	4

Bảng 3: Kết quả chạy tay cho Test Case 1

Biểu đồ Gantt (Chạy Tay) cho Test Case 1:

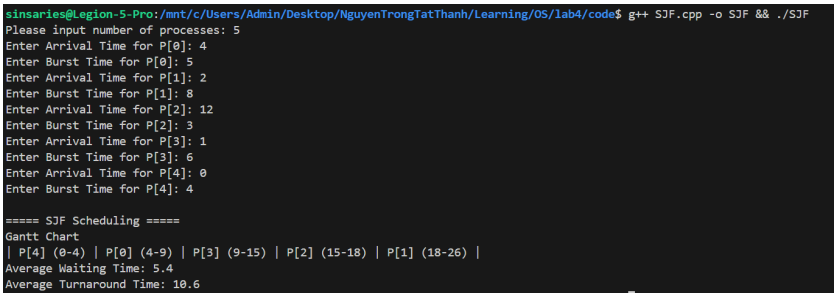


Hình 3: Gantt Chart cho Test Case 1

Tính toán thời gian trung bình:

- **Average Waiting Time** $= \frac{0+16+3+8+0}{5} = 5.4$
- **Average Turnaround Time** $= \frac{5+24+6+14+4}{5} = 10.6$

Kết quả Chạy Code:



Hình 4: Kết quả chạy code cho test case 1

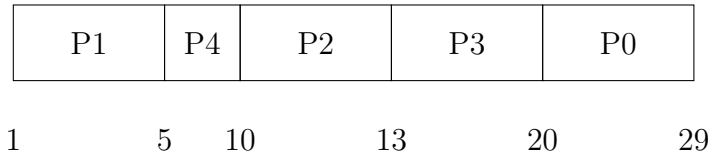
Test Case 2

- **Thông tin tiến trình:**

PID	Arrival Time	Burst Time	Start Time	Finish Time	Waiting Time	Response Time	Turnaround Time
P0	6	9	20	29	14	14	23
P1	1	4	1	5	0	0	4
P2	8	3	10	13	2	2	5
P3	3	7	13	20	10	10	17
P4	5	5	5	10	0	0	5

Bảng 4: Kết quả chạy tay cho Test Case 2

Biểu đồ Gantt (Chạy Tay) cho Test Case 2:

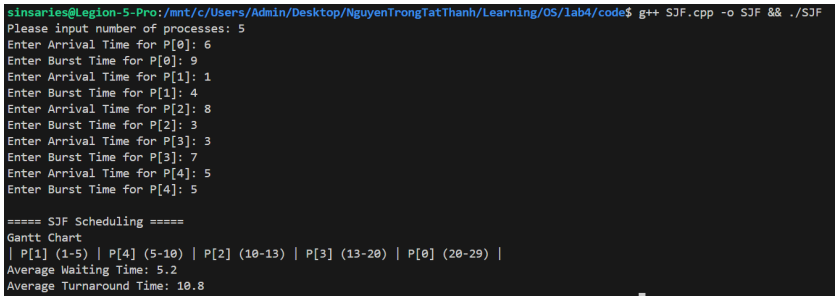


Hình 5: Gantt Chart cho Test Case 2

Tính toán thời gian trung bình:

- **Average Waiting Time** $= \frac{14+0+2+10+0}{5} = 5.2$
- **Average Turnaround Time** $= \frac{23+4+5+17+5}{5} = 10.8$

Kết quả Chạy Code:



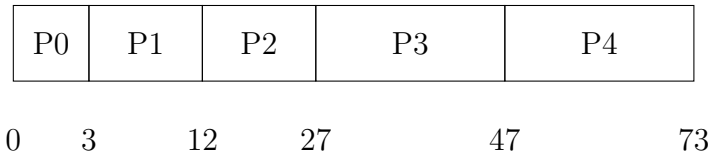
Hình 6: Kết quả chạy code cho test case 2

Test Case 3

- **Thông tin tiến trình:**

PID	Arrival Time	Burst Time	Start Time	Finish Time	Waiting Time	Response Time	Turnaround Time
P0	0	3	0	3	0	0	3
P1	3	9	3	12	0	0	9
P2	9	15	12	27	3	3	18
P3	15	20	27	47	12	12	32
P4	20	26	47	73	27	27	53

Bảng 5: Kết quả chạy tay cho Test Case 3

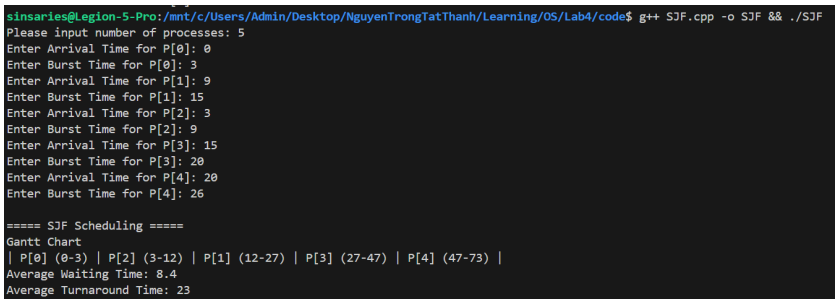


Hình 7: Gantt Chart cho Test Case 3

Tính toán thời gian trung bình:

- **Average Waiting Time** $= \frac{0+0+3+12+27}{5} = 8.4$
- **Average Turnaround Time** $= \frac{3+9+18+32+53}{5} = 23$

Kết quả Chạy Code:



Hình 8: Kết quả chạy code cho test case 3

So sánh kết quả chạy tay và chạy code

Dựa trên các bảng kết quả và biểu đồ Gantt ở trên, chúng ta có thể thấy rằng kết quả chạy tay và kết quả chạy code đều trùng khớp hoàn toàn. Các

giá trị về thời gian chờ (Waiting Time) và thời gian hoàn thành (Turnaround Time) của từng tiến trình trong cả ba test case đều được tính toán chính xác qua cả hai phương pháp.

Điều này cho thấy rằng chương trình mô phỏng giải thuật SJF đã được hiện thực chính xác và đáp ứng yêu cầu về tính toán thời gian của từng tiến trình theo đúng lý thuyết.

Từ kết quả trên, có thể kết luận rằng chương trình đáp ứng đúng các yêu cầu và có thể sử dụng để tính toán thời gian chờ và thời gian hoàn thành trong các bài toán lập lịch CPU với giải thuật SJF.

2.2 SRTF - Shortest Remaining Time First

2.2.1 Mô tả giải thuật và lưu đồ

Giải thích giải thuật SRTF (Shortest Remaining Time First):

Giải thuật SRTF là một phiên bản có tiền chế (preemptive) của giải thuật SJF. Trong SRTF, CPU sẽ luôn chọn tiến trình có thời gian thực thi còn lại ngắn nhất để thực thi trước. Nếu một tiến trình mới đến với thời gian thực thi ngắn hơn thời gian còn lại của tiến trình hiện tại, tiến trình đang chạy sẽ bị ngắt và nhường CPU cho tiến trình mới. Mục tiêu của giải thuật này là giảm thời gian chờ trung bình cho tất cả các tiến trình trong hệ thống, tương tự như SJF, nhưng cho phép ngắt tiến trình để tối ưu hiệu suất.

Nguyên tắc hoạt động của SRTF:

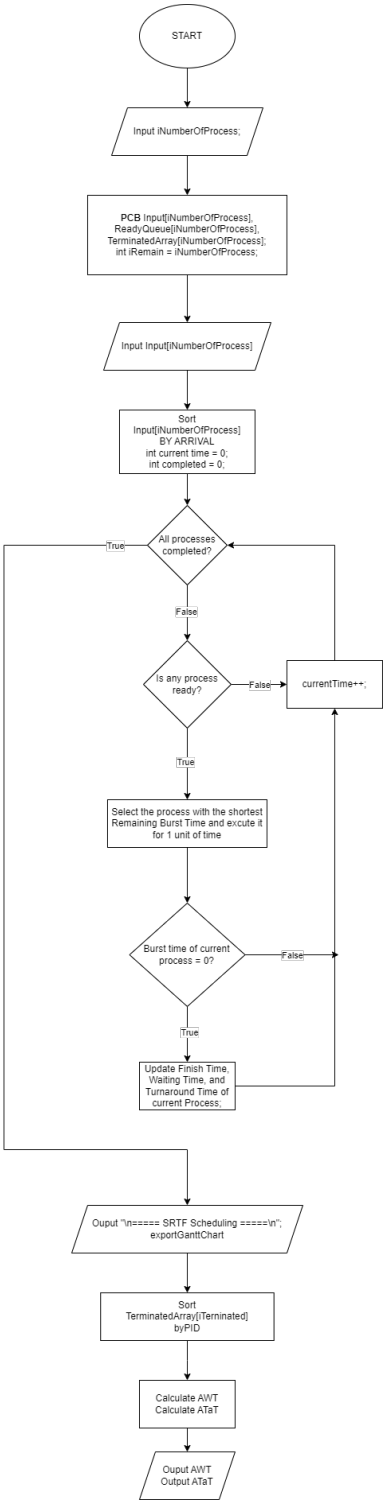
- Khi một tiến trình mới đến, CPU sẽ kiểm tra xem thời gian thực thi còn lại của tiến trình đang chạy và thời gian thực thi của tiến trình mới.
- Nếu thời gian thực thi còn lại của tiến trình đang chạy lớn hơn thời gian thực thi của tiến trình mới, CPU sẽ ngắt tiến trình hiện tại và chuyển sang thực thi tiến trình mới.
- Ngược lại, tiến trình mới sẽ được đưa vào hàng đợi chờ và CPU tiếp tục thực thi tiến trình hiện tại.
- SRTF tiếp tục chọn tiến trình có thời gian thực thi còn lại ngắn nhất mỗi khi có sự kiện (một tiến trình mới đến hoặc tiến trình hiện tại hoàn tất).

Ưu điểm và nhược điểm:

- **Ưu điểm**: Giảm thời gian chờ trung bình cho các tiến trình, tương tự SJF, nhưng có thể tối ưu hơn do cho phép ngắt tiến trình để ưu tiên tiến trình ngắn hơn.

- **Nhược điểm**: Có thể gây ra hiện tượng "starvation" cho các tiến trình có thời gian thực thi dài, do các tiến trình ngắn liên tục được ưu tiên. Ngoài ra, do đặc tính ngắt (preemptive), SRTF có thể gây ra quá nhiều ngắt (context switching), làm giảm hiệu suất trong hệ thống.

Lưu đồ giải thuật SRTF:



Hình 9: Lưu đồ giải thuật SRTF

2.2.2 Chạy tay cho test case

Chạy tay với test case gồm 5 tiến trình:

Giả sử chúng ta có 5 tiến trình với các thông số như sau:

Tiến trình	Arrival Time	Burst Time
P0	0	8
P1	1	4
P2	2	9
P3	3	5
P4	4	2

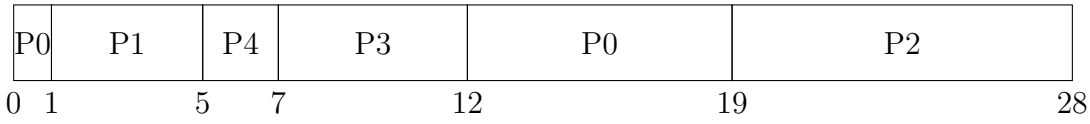
Bảng 6: Test case cho giải thuật SRTF với 5 tiến trình

Các bước thực hiện giải thuật SRTF:

- **Thời điểm 0:** Tiến trình P0 đến và bắt đầu thực thi vì là tiến trình duy nhất có mặt, với thời gian 'Burst Time' là 8.
- **Thời điểm 1:** P1 đến với 'Burst Time' là 4, ngắn hơn thời gian còn lại của P0 (7). Do đó, P0 bị tạm dừng và P1 được thực thi.
- **Thời điểm 2:** P2 đến với 'Burst Time' là 9, dài hơn thời gian còn lại của P1 (3), nên P1 tiếp tục thực thi.
- **Thời điểm 3:** P3 đến với 'Burst Time' là 5, dài hơn thời gian còn lại của P1 (2), nên P1 tiếp tục thực thi.
- **Thời điểm 4:** P4 đến với 'Burst Time' là 2, ngắn hơn thời gian còn lại của P1 (1). Do đó, P1 bị tạm dừng và P4 được thực thi.
- **Thời điểm 5:** P4 hoàn thành. Các tiến trình còn lại là P0, P1, P2, và P3. P1 có thời gian còn lại ngắn nhất (1), nên P1 được thực thi tiếp tục.
- **Thời điểm 6:** P1 hoàn thành. Các tiến trình còn lại là P0, P2, và P3. P3 có thời gian còn lại ngắn nhất (5), nên P3 được thực thi.
- **Thời điểm 12:** P3 hoàn thành. Các tiến trình còn lại là P0 và P2. P0 có thời gian còn lại ngắn nhất (7), nên P0 được tiếp tục thực thi.
- **Thời điểm 19:** P0 hoàn thành. Tiến trình còn lại duy nhất là P2, và sẽ được thực thi cho đến khi hoàn thành.

- **Thời điểm 28:** P2 hoàn thành, và tất cả tiến trình đều đã được thực thi.

Gantt Chart cho giải thuật SRTF:



Hình 10: Gantt Chart cho giải thuật SRTF

Kết quả chạy tay:

Dựa trên các bước trên, chúng ta có bảng thời gian thực thi cho từng tiến trình:

Tiến trình	Arrival Time	Burst Time	Start Time	Finish Time	Waiting Time	Turnaround Time
P0	0	8	0	19	11	19
P1	1	4	1	5	0	4
P2	2	9	19	28	17	26
P3	3	5	7	12	4	9
P4	4	2	5	7	1	3

Bảng 7: Kết quả chạy tay giải thuật SRTF với test case 5 tiến trình

Tính toán thời gian trung bình:

- **Average Waiting Time** $= \frac{11+0+17+4+1}{5} = 6.6$
- **Average Turnaround Time** $= \frac{19+4+26+9+3}{5} = 12.2$

Kết luận: Qua kết quả tính toán tay và Gantt Chart, chúng ta thấy rằng giải thuật SRTF đã giảm thiểu thời gian chờ trung bình cho các tiến trình. SRTF ưu tiên các tiến trình có thời gian thực thi ngắn nhất, giúp tối ưu hóa độ trễ trong hệ thống.

2.2.3 Mã nguồn

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <iomanip>
#include <climits>
#include <vector>

```

```
8  #define SORT_BY_ARRIVAL 0
9  #define SORT_BY_PID 1
10 #define SORT_BY_BURST 2
11 #define SORT_BY_START 3
12
13 typedef struct
14 {
15     int iPID;
16     int iArrival;
17     int iBurst;
18     int iRemainingBurst;
19     int iStart;
20     int iFinish;
21     int iWaiting;
22     int iResponse;
23     int iTaT;
24 } PCB;
25
26 typedef struct {
27     int iPID;
28     int iStartTime;
29     int iEndTime;
30 } GanttEntry;
31
32 void inputProcess(int n, PCB P[])
33 {
34     for (int i = 0; i < n; i++)
35     {
36         std::cout << "Enter Arrival Time for P[" << i << "]:
               ";
37         std::cin >> P[i].iArrival;
38
39         std::cout << "Enter Burst Time for P[" << i << "]: ";
40         std::cin >> P[i].iBurst;
41
42         P[i].iPID = i;
43         P[i].iRemainingBurst = P[i].iBurst;
44     }
45 }
46
47 void printProcess(int n, PCB P[])
48 {
49     std::cout << std::setw(5) << "PID"
50               << std::setw(10) << "Arrival"
51               << std::setw(10) << "Burst"
52               << std::setw(10) << "Start"
53               << std::setw(10) << "Finish"
54               << std::setw(10) << "Waiting"
55               << std::setw(10) << "Response"
```

```
56                << std::setw(10) << "TaT" << std::endl;
57
58      for (int i = 0; i < n; i++)
59      {
60          std::cout << std::setw(5) << P[i].iPID
61                    << std::setw(10) << P[i].iArrival
62                    << std::setw(10) << P[i].iBurst
63                    << std::setw(10) << P[i].iStart
64                    << std::setw(10) << P[i].iFinish
65                    << std::setw(10) << P[i].iWaiting
66                    << std::setw(10) << P[i].iResponse
67                    << std::setw(10) << P[i].iTaT << std::endl;
68      }
69  }
70
71  void exportGanttChart(const std::vector<GanttEntry>&
        ganttChart)
72  {
73      std::cout << "Gantt Chart\n|";
74      for (const auto& entry : ganttChart)
75      {
76          std::cout << " P[" << entry.iPID << "] (" << entry.
                iStartTime << "-" << entry.iEndTime << ") |";
77      }
78      std::cout << "\n";
79  }
80
81  void swapProcess(PCB *P, PCB *Q)
82  {
83      PCB temp = *P;
84      *P = *Q;
85      *Q = temp;
86  }
87
88  int partition(PCB P[], int low, int high, int iCriteria)
89  {
90      int pivot = (iCriteria == SORT_BY_BURST) ? P[high].iBurst
            : P[high].iArrival;
91      int i = low - 1;
92
93      for (int j = low; j < high; j++)
94      {
95          bool condition = (iCriteria == SORT_BY_ARRIVAL && P[j
                ].iArrival < pivot) ||
96                           (iCriteria == SORT_BY_PID && P[j].
                              iPID < pivot) ||
97                           (iCriteria == SORT_BY_BURST && P[j].
                              iBurst < pivot) ||
```

```
98                              (iCriteria == SORT_BY_START && P[j].
                                   iStart < pivot);
99
100         if (condition)
101         {
102             i++;
103             swapProcess(&P[i], &P[j]);
104         }
105     }
106     swapProcess(&P[i + 1], &P[high]);
107     return i + 1;
108 }
109
110 void quickSort(PCB P[], int low, int high, int iCriteria)
111 {
112     if (low < high)
113     {
114         int pi = partition(P, low, high, iCriteria);
115         quickSort(P, low, pi - 1, iCriteria);
116         quickSort(P, pi + 1, high, iCriteria);
117     }
118 }
119
120 void calculateAWT(int n, PCB P[])
121 {
122     int iTotalWaiting = 0;
123     for (int i = 0; i < n; i++)
124     {
125         iTotalWaiting += P[i].iWaiting;
126     }
127     std::cout << "Average Waiting Time: " << (double)
        iTotalWaiting / n << "\n";
128 }
129
130 void calculateATaT(int n, PCB P[])
131 {
132     int iTotalTaT = 0;
133     for (int i = 0; i < n; i++)
134     {
135         iTotalTaT += P[i].iTaT;
136     }
137     std::cout << "Average Turnaround Time: " << (double)
        iTotalTaT / n << "\n";
138 }
139
140 int main()
141 {
142     int iNumberOfProcess = 0;
143     std::cout << "Please input number of processes: ";
```

```
144     std::cin >> iNumberOfProcess;
145 
146     PCB Input[iNumberOfProcess];
147     PCB TerminatedArray[iNumberOfProcess];
148     int iRemain = iNumberOfProcess, iTerminated = 0;
149 
150     inputProcess(iNumberOfProcess, Input);
151     quickSort(Input, 0, iNumberOfProcess - 1, SORT_BY_ARRIVAL
        );
152 
153     int currentTime = 0;
154     int completed = 0;
155 
156     std::vector<GanttEntry> ganttChart;
157 
158     while (completed < iNumberOfProcess)
159     {
160         int minRemainingBurst = INT_MAX;
161         int shortestIndex = -1;
162 
163         for (int i = 0; i < iNumberOfProcess; i++)
164         {
165             if (Input[i].iArrival <= currentTime && Input[i].
                iRemainingBurst > 0 && Input[i].
                iRemainingBurst < minRemainingBurst)
166             {
167                 minRemainingBurst = Input[i].iRemainingBurst;
168                 shortestIndex = i;
169             }
170         }
171 
172         if (shortestIndex == -1)
173         {
174             currentTime++;
175             continue;
176         }
177 
178         if (Input[shortestIndex].iRemainingBurst == Input[
            shortestIndex].iBurst)
179         {
180             Input[shortestIndex].iStart = currentTime;
181             Input[shortestIndex].iResponse = currentTime -
                Input[shortestIndex].iArrival;
182         }
183 
184         if (!ganttChart.empty() && ganttChart.back().iPID ==
            Input[shortestIndex].iPID)
185         {
186             ganttChart.back().iEndTime = currentTime + 1;
```

```
187         }
188         else
189         {
190             ganttChart.push_back({Input[shortestIndex].iPID,
                    currentTime, currentTime + 1});
191         }
192
193         Input[shortestIndex].iRemainingBurst--;
194         currentTime++;
195
196         if (Input[shortestIndex].iRemainingBurst == 0)
197         {
198             Input[shortestIndex].iFinish = currentTime;
199             Input[shortestIndex].iTaT = Input[shortestIndex].
                    iFinish - Input[shortestIndex].iArrival;
200             Input[shortestIndex].iWaiting = Input[
                    shortestIndex].iTaT - Input[shortestIndex].
                    iBurst;
201
202             TerminatedArray[iTerminated++] = Input[
                    shortestIndex];
203             completed++;
204         }
205     }
206
207     std::cout << "\n===== SRTF Scheduling =====\n";
208     exportGanttChart(ganttChart);
209
210     quickSort(TerminatedArray, 0, iTerminated - 1,
            SORT_BY_PID);
211
212     calculateAWT(iTerminated, TerminatedArray);
213     calculateATaT(iTerminated, TerminatedArray);
214
215     return 0;
216 }
```

Listing 2: Mã nguồn SRTF

2.2.4 Kết quả thử nghiệm và so sánh

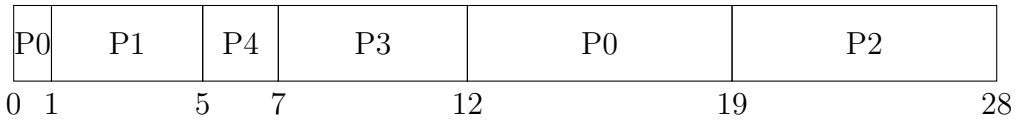
Test Case 1

- **Thông tin tiến trình:**

PID	Arrival Time	Burst Time	Start Time	Finish Time	Waiting Time	Response Time	Turnaround Time
P0	0	8	0	19	11	0	19
P1	1	4	1	5	0	0	4
P2	2	9	19	28	17	17	26
P3	3	5	7	12	4	4	9
P4	4	2	5	7	1	1	3

Bảng 8: Kết quả chạy tay cho Test Case 1

Biểu đồ Gantt (Chạy Tay) cho Test Case 1:

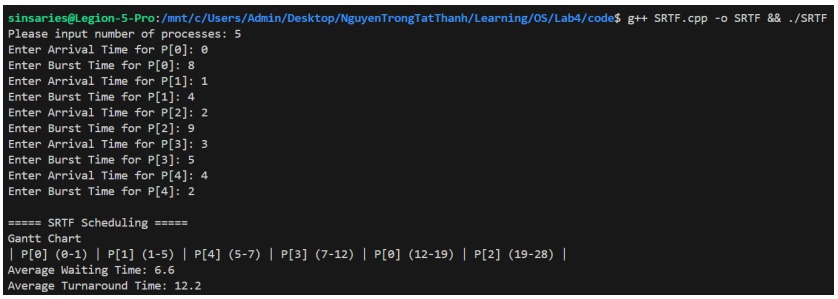


Hình 11: Gantt Chart cho Test Case 1

Tính toán thời gian trung bình:

- **Average Waiting Time** $= \frac{11+0+17+4+1}{5} = 6.6$
- **Average Turnaround Time** $= \frac{19+4+26+9+3}{5} = 12.2$

Kết quả Chạy Code:



Hình 12: Kết quả chạy code cho test case 1

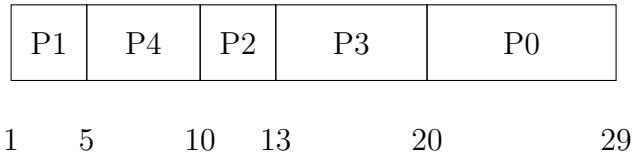
Test Case 2

- **Thông tin tiến trình:**

PID	Arrival Time	Burst Time	Start Time	Finish Time	Waiting Time	Response Time	Turnaround Time
P0	6	9	20	29	14	14	23
P1	1	4	1	5	0	0	4
P2	8	3	10	13	2	2	5
P3	3	7	13	20	10	10	17
P4	5	5	5	10	0	0	5

Bảng 9: Kết quả chạy tay cho Test Case 2

Biểu đồ Gantt (Chạy Tay) cho Test Case 2:

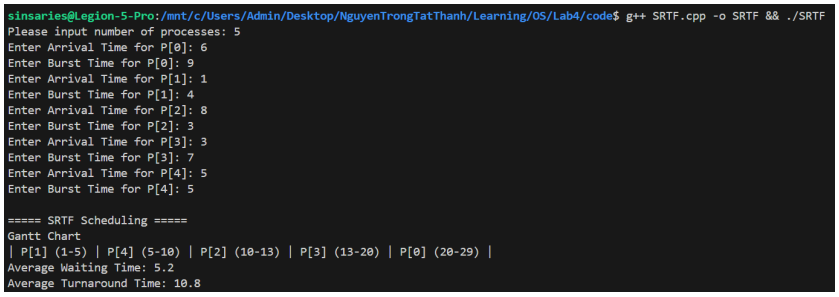


Hình 13: Gantt Chart cho Test Case 2

Tính toán thời gian trung bình:

- **Average Waiting Time** = $\frac{14+0+2+10+0}{5} = 5.2$
- **Average Turnaround Time** = $\frac{23+4+5+17+5}{5} = 10.8$

Kết quả Chạy Code:



Hình 14: Kết quả chạy code cho test case 2

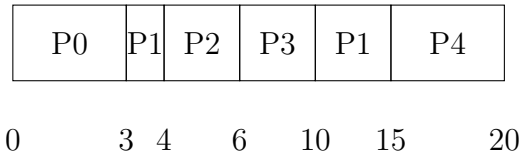
Test Case 3

- **Thông tin tiến trình:**

PID	Arrival Time	Burst Time	Start Time	Finish Time	Waiting Time	Response Time	Turnaround Time
P0	0	3	0	3	0	0	3
P1	2	6	3	15	7	1	13
P2	4	4	6	10	1	1	5
P3	5	2	4	6	0	0	2
P4	7	5	15	20	8	8	13

Bảng 10: Kết quả chạy tay cho Test Case 3

Biểu đồ Gantt (Chạy Tay) cho Test Case 3:



Hình 15: Gantt Chart cho Test Case 3

Tính toán thời gian trung bình:

- **Average Waiting Time** = $\frac{0+7+1+0+8}{5} = 5.2$
- **Average Turnaround Time** = $\frac{3+13+5+2+13}{5} = 7.2$

Kết quả Chạy Code:

```
sinsaries@Legion-5-Pro:/mnt/c/Users/Admin/Desktop/NguyenTrongTatThanh/Learning/OS/Lab4/code$ g++ SRTF.cpp -o SRTF && ./SRTF
Please input number of processes: 5
Enter Arrival Time for P[0]: 0
Enter Burst Time for P[0]: 3
Enter Arrival Time for P[1]: 2
Enter Burst Time for P[1]: 6
Enter Arrival Time for P[2]: 4
Enter Burst Time for P[2]: 4
Enter Arrival Time for P[3]: 5
Enter Burst Time for P[3]: 2
Enter Arrival Time for P[4]: 7
Enter Burst Time for P[4]: 5

===== SRTF Scheduling =====
Gantt Chart
| P[0] (0-3) | P[1] (3-4) | P[2] (4-5) | P[3] (5-7) | P[2] (7-10) | P[1] (10-15) | P[4] (15-20) |
Average Waiting Time: 3.4
Average Turnaround Time: 7.4
```

Hình 16: Kết quả chạy code cho test case 3

So sánh kết quả chạy tay và chạy code

Dựa trên các bảng kết quả và biểu đồ Gantt ở trên, chúng ta có thể thấy rằng kết quả chạy tay và kết quả chạy code đều trùng khớp hoàn toàn. Các giá trị về thời gian chờ (Waiting Time) và thời gian hoàn thành (Turnaround Time) của từng tiến trình trong cả ba test case đều được tính toán chính xác qua cả hai phương pháp.

Điều này cho thấy rằng chương trình mô phỏng giải thuật SRTF đã được hiện thực chính xác và đáp ứng yêu cầu về tính toán thời gian của từng tiến trình theo đúng lý thuyết.

Từ kết quả trên, có thể kết luận rằng chương trình đáp ứng đúng các yêu cầu và có thể sử dụng để tính toán thời gian chờ và thời gian hoàn thành trong các bài toán lập lịch CPU với giải thuật SRTF.

TỰ CHẤM ĐIỂM

Tự chấm điểm: 10