University of Prince Mugrin
College of Computer and Cyber Sciences
Department of Software Engineering

SE431- Software Maintenance and Evolution

Course Project – Semester I (Fall 2023)

Program Slicing on

First Come First Served (FCFS) Algorithm

Team Members:

Bassant Mahmoud | 4010064

Salwa Shamma | 4010405

Samah Channa | 4010318

Samah Shamma | 4010403

Sana Shamma | 4010404

Instructor:

Dr. Ftoon Kedwan

December 17, 2023

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 Code Description (First Come First Served Algorithm)

This project uses a code that simulates a common algorithm used for assigning the CPU to processes, known as First Come First Served (FCFS). The goal of this algorithm is to assign the CPU based on the order of requests. It is a simple approach where processes keep running on the CPU until they are blocked or terminated. The table 1.1.1 below shows the advantages and disadvantages of this algorithm:

FCFS Scheduling	
Advantages	Simplicity, in terms of implementation.
Disadvantages	Short jobs can get stuck behind long jobs.

Table 1.1.1

Before delving deeper into this project, there are some terminologies that should be highlighted:

- **Arrival Time:** The time when a process enters the ready queue.
- **Burst Time:** The time required for a process to complete its execution.
- **Start Time:** The time when a process begins its execution on the CPU.
- **End Time:** The time when a process finishes its execution and completes its task.
- **Turnaround Time**: The time from submission to completion.
- **Waiting Time:** The amount of time a process has been waiting in the ready queue.

One of the constraints of this algorithm is that it requires sorting. Here is an example to demonstrate this concept.

Process	Arrive time	Burst time
P1	0	10
P2	3	18

Table 1.1.2

The table 1.1.2 shows two processes. The first process (P1) has an arrival time of 0, which is the time it enters the queue, and a burst time of 10, which is the time required for the process to complete. Similarly, the second process (P2) has an arrival time of 3, the time it enters the queue, and a burst time of 18, which is the time required for it to complete.

To calculate the turnaround time and wait time for each process, we need to know the start and end times for each process.

For the first process (P1):

The start time is when the CPU starts working on this process. Since it's the first process in the queue, the start time will be 0 (which is equal to the arrival time in this case). The end time is when the CPU is done with this process, which is equal to the current time.

- To calculate the **wait time** for P1, we subtract the time when the process starts to be handled by the CPU (**start time**) from the time the process entered the queue (**arrival time**). In this case, it will be 0 - 0 = 0. Therefore, the wait time for P1 is zero.
- To calculate **turnaround time**, which represents the total time from when the process enters the queue to when the CPU is done with it. It can be calculated by subtracting the end time of the CPU from this process (**end time**) from the time the process enters the queue (**arrival time**), it will be 10 - 0 = 10. Therefore, the turnaround time will be 10.

For the second process (P2):

The start time is when the CPU starts working on this process. Since it's the second process in the queue, the start time will be 10 (which is equal to end time for the previous process). The end time is when the CPU is done with this process, which is equal to the current time.

- To calculate the **wait time** for P2, we subtract the time when the process starts to be handled by the CPU (**start time**) from the time the process entered the queue (**arrival time**). In this case, it will be 10 - 3 = 7. Therefore, the wait time for P1 is 7.
- To calculate **turnaround time**, which represents the total time from when the process enters the queue to when the CPU is done with it. It can be calculated by subtracting the end time of the CPU from this process (**end time** which equals the current time) from the time the process enters the queue (**arrival time**), it will be 28 - 3 = 25. Therefore, the turnaround time will be 10.

The table 1.1.3 summarizes these calculations:

	P1	P2
Waiting Time (start time - arrive time)	0 - 0 = 0	10 - 3 = 7
Turnaround Time (end time - arrive time)	10 - 0 = 10	(10 + 18) - 3 = 25

Table 1.1.3

To view these calculations in terms of a programmatic perspective, the code snippet below, Figure 1.1.1, demonstrates the FCFS algorithm implemented in the C language. The main parts of this code are highlighted as follows:

- Lines 1 to 10: These lines define a structure (similar to a class) that will be used later to create an array of objects with specific attributes.
- Lines 11 to 26: These lines are responsible for calculating the wait and turnaround times.
- Lines 27 to 38: These lines prompt the user for input, specifically the arrival time and turnaround time.
- Line 39: This line executes the function.
- Lines 40 to 49: These lines handle the printing of the results.

FCFS Algorithm Code:

```c
#include <stdio.h>
typedef struct {
    int arrivalTime;
    int burstTime;
    int startTime;
    int endTime;
    int waitTime;
    int turnaroundTime;
    int id;
} Process;
void calculateTimes(Process processes[], int numProcesses) {
    for (int i = 0; i < numProcesses - 1; i++) {
        for (int j = 0; j < numProcesses - i - 1; j++) {
            if (processes[j].arrivalTime > processes[j + 1].arrivalTime) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;}}}
    int currentTime = 0;
    for (int i = 0; i < numProcesses; i++) {
        if (currentTime < processes[i].arrivalTime)
            currentTime = processes[i].arrivalTime;
        processes[i].startTime = currentTime;
        currentTime += processes[i].burstTime;
        processes[i].endTime = currentTime;
        processes[i].turnaroundTime = processes[i].endTime - processes[i].arrivalTime;
        processes[i].waitTime = processes[i].startTime - processes[i].arrivalTime;}}
int main() {
    int numProcesses;
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);
    Process processes[numProcesses];
    int totalTurnaroundTime = 0, totalWaitTime = 0;
    for (int i = 0; i < numProcesses; i++) {
        printf("\nEnter the arrival time for process %d: ", i + 1);
        scanf("%d", &processes[i].arrivalTime);
        printf("Enter the burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burstTime);
        processes[i].id = i + 1;}
    calculateTimes(processes, numProcesses);
    printf("\nProcess\tStart Time\tEnd Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < numProcesses; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].startTime, processes[i].endTime, processes[i].turnaroundTime, processes[i].waitTime);
        totalTurnaroundTime += processes[i].turnaroundTime;
        totalWaitTime += processes[i].waitTime;}
    double averageTurnaroundTime = (double)totalTurnaroundTime / numProcesses;
    double averageWaitTime = (double)totalWaitTime / numProcesses;
    printf("\nAverage Turnaround Time: %.2lf\n", averageTurnaroundTime);
    printf("Average Waiting Time: %.2lf\n", averageWaitTime);
    return 0;}
```

Figure 1.1.1 - FCFS Algorithm

CHAPTER 2: ANALYSIS AND DISCUSSION

2.1 Static Slicing

In this section, we applied static slicing, specifically backward slicing, to seven variables: arrival time, burst time, start time, end time, turnaround time, wait time, and NumProcess. For each variable, we provided the reduced code as well as an explanation.

Note: The gray color represents data flow dependencies, while the blue color represents control flow dependencies.

A. A slicing criterion of a program FCFS is <35; arrivalTime>

```
#include <stdio.h>
typedef struct {
    int arrivalTime;
    int burstTime;
    int startTime;
    int endTime;
    int waitTime;
    int turnaroundTime;
    int id;
} Process;
void calculateTimes(Process processes[], int numProcesses) {
    for (int i = 0; i < numProcesses - 1; i++) {
        for (int j = 0; j < numProcesses - i - 1; j++) {
            if (processes[j].arrivalTime > processes[j + 1].arrivalTime) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;}}}
    int currentTime = 0;
    for (int i = 0; i < numProcesses; i++) {
        if (currentTime < processes[i].arrivalTime)
            currentTime = processes[i].arrivalTime;
        processes[i].startTime = currentTime;
        currentTime += processes[i].burstTime;
        processes[i].endTime = currentTime;
        processes[i].turnaroundTime = processes[i].endTime - processes[i].arrivalTime;
        processes[i].waitTime = processes[i].startTime - processes[i].arrivalTime;}}
int main() {
    int numProcesses;
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);
    Process processes[numProcesses];
    int totalTurnaroundTime = 0, totalWaitTime = 0;
    for (int i = 0; i < numProcesses; i++) {
        printf("\nEnter the arrival time for process %d: ", i + 1);
        scanf("%d", &processes[i].arrivalTime);
        printf("Enter the burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burstTime);
        processes[i].id = i + 1;}
    calculateTimes(processes, numProcesses);
    printf("\nProcess\tStart Time\tEnd Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < numProcesses; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].startTime, processes[i].endTime, processes[i].turnaroundTime, processes[i].waitTime);
        totalTurnaroundTime += processes[i].turnaroundTime;
        totalWaitTime += processes[i].waitTime;}
    double averageTurnaroundTime = (double)totalTurnaroundTime / numProcesses;
    double averageWaitTime = (double)totalWaitTime / numProcesses;
    printf("\nAverage Turnaround Time: %.2lf\n", averageTurnaroundTime);
    printf("Average Waiting Time: %.2lf\n", averageWaitTime);
    return 0;}
```

Figure 2.1.1 - FCFS <35; arrivalTime>

To perform static slicing on the "arrivalTime" variable, we first need to determine the program point where the slicing will begin. In this case, we have chosen line 35 in Figure 2.1.1 as the starting point for backward slicing. Next, we trace back from the identified program point and examine all instructions and predicates that affect the value of "arrivalTime" at line 35. We then remove the parts of the code that are not affected by the removal of "arrivalTime".

```c
#include <stdio.h>

typedef struct {
    int arrivalTime;
} Process;

int main() {
    int numProcesses;
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);

    Process processes[numProcesses];

    for (int i = 0; i < numProcesses; i++) {
        printf("\nEnter the arrival time for process %d: ", i + 1);
        scanf("%d", &processes[i].arrivalTime);
    }
    return 0;
}
```

Figure 2.1.2- Output of FCFS <35; arrivalTime>

We start slicing from the main function, specifically from lines 35 to 33. These lines are responsible for obtaining the values of "arrivalTime" from user input, and they directly impact the computation involving "arrivalTime". Line 32 is deleted as it does not contribute to the computation of "arrivalTime". Line 31, on the other hand, affects the computation of "arrivalTime" as it determines the number of processes that will be created, each carrying a value of "arrivalTime". Lines 30 to 27 prompt the user to enter the number of processes that will utilize "arrivalTime", thus impacting its

computation. Lines 26 to 11 do not directly affect the computation of "arrivalTime", but they utilize the values of "arrivalTime" to calculate another variable. Finally, lines 10 to 2 contain the structure where "arrivalTime" is initialized but the only lines that we are concerned about are line 10, 3 ,2 and line 1 is essential for running the C file. The final code is shown in Figure 2.1.2.

The summary of this process:

- **The slicing lines are**: [35, 34, 33, 31, 30, 29, 28, 27, 10, 3, 2, 1]
- **The data dependency lines are:** [35, 34, 3]
- **The variable dependencies:** This variable does not depend on the other one.

B. A slicing criterion of a program FCFS is < 37; burstTime>

```c
#include <stdio.h>
typedef struct {
    int arrivalTime;
    int burstTime;
    int startTime;
    int endTime;
    int waitTime;
    int turnaroundTime;
    int id;
} Process;
void calculateTimes(Process processes[], int numProcesses) {
    for (int i = 0; i < numProcesses - 1; i++) {
        for (int j = 0; j < numProcesses - i - 1; j++) {
            if (processes[j].arrivalTime > processes[j + 1].arrivalTime) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;}}}
    int currentTime = 0;
    for (int i = 0; i < numProcesses; i++) {
        if (currentTime < processes[i].arrivalTime)
            currentTime = processes[i].arrivalTime;
        processes[i].startTime = currentTime;
        currentTime += processes[i].burstTime;
        processes[i].endTime = currentTime;
        processes[i].turnaroundTime = processes[i].endTime - processes[i].arrivalTime;
        processes[i].waitTime = processes[i].startTime - processes[i].arrivalTime;}}
int main() {
    int numProcesses;
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);
    Process processes[numProcesses];
    int totalTurnaroundTime = 0, totalWaitTime = 0;
    for (int i = 0; i < numProcesses; i++) {
        printf("\nEnter the arrival time for process %d: ", i + 1);
        scanf("%d", &processes[i].arrivalTime);
        printf("Enter the burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burstTime);
        processes[i].id = i + 1;}
    calculateTimes(processes, numProcesses);
    printf("\nProcess\tStart Time\tEnd Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < numProcesses; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].startTime, processes[i].endTime, processes[i].turnaroundTime, processes[i].waitTime);
        totalTurnaroundTime += processes[i].turnaroundTime;
        totalWaitTime += processes[i].waitTime;}
    double averageTurnaroundTime = (double)totalTurnaroundTime / numProcesses;
    double averageWaitTime = (double)totalWaitTime / numProcesses;
    printf("\nAverage Turnaround Time: %.2lf\n", averageTurnaroundTime);
    printf("Average Waiting Time: %.2lf\n", averageWaitTime);
    return 0;}
```

Figure 2.1.3 - FCFS < 37; burstTime>

To perform static slicing on the "burstTime" variable, we first need to determine the program point where the slicing will begin. In this case, we have chosen line 37 in Figure 2.1.3 as the starting point for backward slicing. Next, we trace back from the identified program point and examine all instructions that affect the value of "burstTime" at line 37. We then remove the statements that may not affect the value of the variable "burstTime".

```c
#include <stdio.h>

typedef struct {
    int burstTime;
} Process;

int main() {
    int numProcesses;
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);

    Process processes[numProcesses];

    for (int i = 0; i < numProcesses; i++) {
        printf("Enter the burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burstTime);
    }

    return 0;
}
```

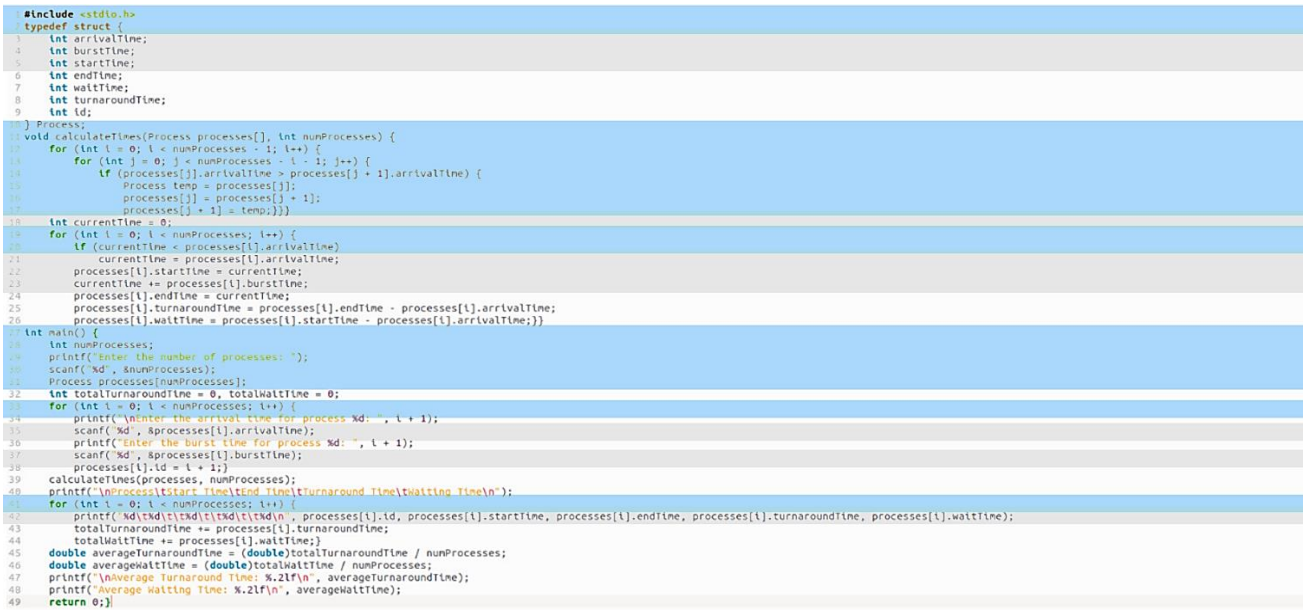
Figure 2.1.4 - Output of FCFS < 37; burstTime>

We start slicing from the main function, specifically from lines 37, 36 and 33. These lines are responsible for obtaining the values of "burstTime" from user input, and they directly impact the computation involving "burstTime". Line 32 is deleted as it does not contribute to the computation of "burstTime". Line 31, on the other hand, affects the computation of "burstTime" as it determines the number of processes that will be created, each carrying a value of " burstTime". Lines 30 to 28 prompt the user to enter the number of processes that will utilize "burstTime", thus impacting its computation. Lines 26 to 11 do not affect the computation of "burstTime", but they utilize the values of "burstTime" to calculate another variable. Finally, lines 10 to 2 contain the structure where "burstTime" is initialized, the remaining attributes were deleted because there is no relation between them and "burstTime". The final code is shown in Figure 2.1.4.

The summary of this process:

- **The slicing lines are**: [37, 36, 33, 31, 30, 29, 28, 27, 10, 4, 2, 1].
- **The data dependency lines are:** [37, 36, 4]
- **The variable dependencies:** This variable does not depend on the other one.

C. A slicing criterion of a program FCFS is < 42, startTime >



```c
#include <stdio.h>
typedef struct {
    int arrivalTime;
    int burstTime;
    int startTime;
    int endTime;
    int waitTime;
    int turnaroundTime;
    int id;
} Process;
void calculateTimes(Process processes[], int numProcesses) {
    for (int i = 0; i < numProcesses - 1; i++) {
        for (int j = 0; j < numProcesses - i - 1; j++) {
            if (processes[j].arrivalTime > processes[j + 1].arrivalTime) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;}}}
    int currentTime = 0;
    for (int i = 0; i < numProcesses; i++) {
        if (currentTime < processes[i].arrivalTime)
            currentTime = processes[i].arrivalTime;
        processes[i].startTime = currentTime;
        currentTime += processes[i].burstTime;
        processes[i].endTime = currentTime;
        processes[i].turnaroundTime = processes[i].endTime - processes[i].arrivalTime;
        processes[i].waitTime = processes[i].startTime - processes[i].arrivalTime;}}
int main() {
    int numProcesses;
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);
    Process processes[numProcesses];
    int totalTurnaroundTime = 0, totalWaitTime = 0;
    for (int i = 0; i < numProcesses; i++) {
        printf("\nEnter the arrival time for process %d: ", i + 1);
        scanf("%d", &processes[i].arrivalTime);
        printf("Enter the burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burstTime);
        processes[i].id = i + 1;}
    calculateTimes(processes, numProcesses);
    printf("\nProcess\tStart Time\tEnd Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < numProcesses; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].startTime, processes[i].endTime, processes[i].turnaroundTime, processes[i].waitTime);
        totalTurnaroundTime += processes[i].turnaroundTime;
        totalWaitTime += processes[i].waitTime;}
    double averageTurnaroundTime = (double)totalTurnaroundTime / numProcesses;
    double averageWaitTime = (double)totalWaitTime / numProcesses;
    printf("\nAverage Turnaround Time: %.2lf\n", averageTurnaroundTime);
    printf("Average Waiting Time: %.2lf\n", averageWaitTime);
    return 0;}
```

Figure 2.1.5 - FCFS < 42, startTime >

To initiate I started looking for the variable in Line 42 in Figure 2.1.5 before extracting the slicing portion, that is mainly the slicing criterion of a program FIFO is S <42, processes[i].startTime>. The next step involves tracing back from this chosen starting point, via inspecting the code to find instructions and conditions that influence the "startTime" variable's value. The goal is to retain the portions of the code that are relevant to the "startTime" computation and eliminate the parts that aren't affected by the removal of "startTime."

To perform backward slicing on the variable StartTime, I start at the end of the function calculateTimes(), where the variable is used, and then I identify all statements that can affect its value. These statements are the assignment statement processes[i].startTime = currentTime; and the increment statement currentTime += processes[i].burstTime;. I then recursively identify all statements that can affect the value of currentTime, and so on. This process terminates when I reach statements that cannot affect the value of StartTime in any way. In the given C code, the only statements that can affect the value of StartTime are the ones in the loop that calculates the start time for each process. This means that the backward slice of StartTime consists of the entire loop. The final code is shown in Figure 2.1.6.

```c
#include <stdio.h>

typedef struct {
    int arrivalTime;
    int burstTime;
    int startTime;
} Process;

void calculateTimes(Process processes[], int numProcesses) {
    for (int i = 0; i < numProcesses - 1; i++) {
        for (int j = 0; j < numProcesses - i - 1; j++) {
            if (processes[j].arrivalTime > processes[j + 1].arrivalTime) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }

    int currentTime = 0;

    for (int i = 0; i < numProcesses; i++) {
        if (currentTime < processes[i].arrivalTime)
            currentTime = processes[i].arrivalTime;

        processes[i].startTime = currentTime;
        currentTime += processes[i].burstTime;

    }
}
```

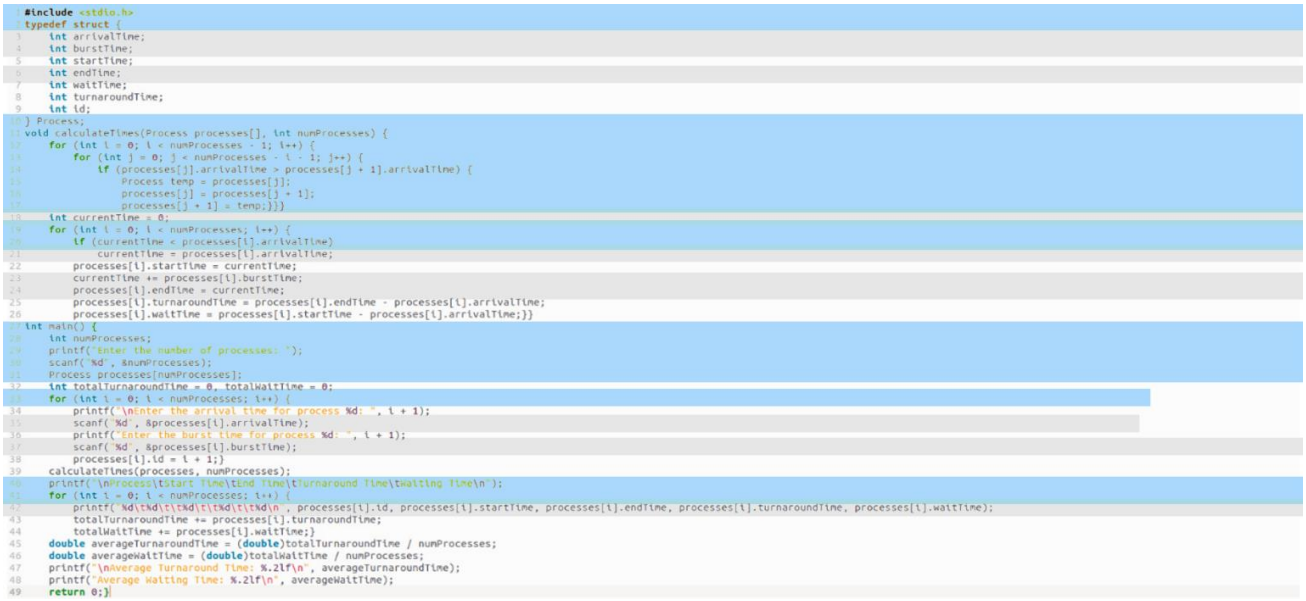
```c
int main() {
    int numProcesses;
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);

    Process processes[numProcesses];


    for (int i = 0; i < numProcesses; i++) {
        printf("\nEnter the arrival time for process %d: ", i + 1);
        scanf("%d", &processes[i].arrivalTime);

        printf("Enter the burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burstTime);


    }

    calculateTimes(processes, numProcesses);

    printf("Start Time\n");
    for (int i = 0; i < numProcesses; i++) {
        printf("%d\n", processes[i].startTime);
        }


    return 0;
}
```

Figure 2.1.6- Output of FCFS < 42, startTime >

The summary of this process:

- **The Static lines**:[42,41,37,35,33,31,30,29,28,27,23,22,21,20,19,18,17,16,15,14,13,12,11,10,5,4,3,2,1]
- **The data dependency lines are:** [37, 35, 23, 22, 21, 18, 5, 4, 3]
- **The variable dependencies:** [startTime, currentTime, arrivalTime, burstTime]

D. A slicing criterion of a program FCFS is <42, endTime >



```
#include <stdio.h>
typedef struct {
    int arrivalTime;
    int burstTime;
    int startTime;
    int endTime;
    int waitTime;
    int turnaroundTime;
    int id;
} Process;
void calculateTimes(Process processes[], int numProcesses) {
    for (int i = 0; i < numProcesses - 1; i++) {
        for (int j = 0; j < numProcesses - i - 1; j++) {
            if (processes[j].arrivalTime > processes[j + 1].arrivalTime) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;}}}
    int currentTime = 0;
    for (int i = 0; i < numProcesses; i++) {
        if (currentTime < processes[i].arrivalTime)
            currentTime = processes[i].arrivalTime;
        processes[i].startTime = currentTime;
        currentTime += processes[i].burstTime;
        processes[i].endTime = currentTime;
        processes[i].turnaroundTime = processes[i].endTime - processes[i].arrivalTime;
        processes[i].waitTime = processes[i].startTime - processes[i].arrivalTime;}}
int main() {
    int numProcesses;
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);
    Process processes[numProcesses];
    int totalTurnaroundTime = 0, totalWaitTime = 0;
    for (int i = 0; i < numProcesses; i++) {
        printf("\nEnter the arrival time for process %d: ", i + 1);
        scanf("%d", &processes[i].arrivalTime);
        printf("Enter the burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burstTime);
        processes[i].id = i + 1;}
    calculateTimes(processes, numProcesses);
    printf("\nProcess\tStart Time\tEnd Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < numProcesses; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].startTime, processes[i].endTime, processes[i].turnaroundTime, processes[i].waitTime);
        totalTurnaroundTime += processes[i].turnaroundTime;
        totalWaitTime += processes[i].waitTime;}
    double averageTurnaroundTime = (double)totalTurnaroundTime / numProcesses;
    double averageWaitTime = (double)totalWaitTime / numProcesses;
    printf("\nAverage Turnaround Time: %.2lf\n", averageTurnaroundTime);
    printf("Average Waiting Time: %.2lf\n", averageWaitTime);
    return 0;}
```

Figure 2.1.7- FCFS <42, endTime >

To start slicing, it is essential to determine the slicing technique and know the starting point and the variable that needs to be traced. In this case, the variable that needs to be traced is "processes[i].endTime", and the slicing technique used is backwards. Therefore, our starting point is line 42 in the main method.

As seen in Figure 2.1.7, lines 42, 41, and 40 are responsible for printing the "endTime" results, so we will keep them. Considering that the "numProcesses" variable impacts the printing of the "endTime" value, we will keep every line of code that includes this variable, namely lines 28, 29, 30, and 31 in the main method. Moving up in the code, we have the "calculateTime" method. When examining it from the bottom, at line 26, our variable "endTime" appears to be assigned. Thus, we can remove lines 26 and 25. Line 24 shows that "endTime" depends on the "currentTime" variable, so we will keep this variable and any lines of code that involve it, such as lines 23 and lines from 21 to 18.

As mentioned in line 23, "currentTime" depends on "burstTime", so we will keep any lines of code that involve this variable as well, such as lines 36 and 37 in the main method. Since these lines are inside a for loop, we will keep the for loop at line 33 and any variables required for its execution. Going back to line 23 in the "calculateTime" method, since it is inside a for loop, we will keep it and also keep line 19. Moving further up to the "process" structure at line 2, as mentioned earlier, "endTime" depends on "currentTime", and "currentTime" depends on "burstTime" and "arrivalTime". These variables will be retained inside the "process" structure, and the remaining variables will be removed. Finally, the remaining line of code that was not mentioned will be removed as well. In the end, we obtain a simplified program as shown in Figure 2.1.8.

```c
typedef struct {
    int arrivalTime;
    int burstTime;
    int endTime;
} Process;

void calculateTimes(Process processes[], int numProcesses) {
    // Bubble sort processes based on arrival times
    for (int i = 0; i < numProcesses - 1; i++) {
        for (int j = 0; j < numProcesses - i - 1; j++) {
            if (processes[j].arrivalTime > processes[j + 1].arrivalTime) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }

    int currentTime = 0;

    for (int i = 0; i < numProcesses; i++) {
        if (currentTime < processes[i].arrivalTime)
            currentTime = processes[i].arrivalTime;

        currentTime += processes[i].burstTime;
        processes[i].endTime = currentTime;

    }
}
```

```c
int main() {
    int numProcesses;
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);

    Process processes[numProcesses];

    for (int i = 0; i < numProcesses; i++) {
        printf("\nEnter the arrival time for process %d: ", i + 1);
        scanf("%d", &processes[i].arrivalTime);

        printf("Enter the burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burstTime);

    }

    calculateTimes(processes, numProcesses);

    printf("End Time\n");
    for (int i = 0; i < numProcesses; i++) {
         printf("%d\n", processes[i].endTime);
    }

    return 0;
}
```

Figure 2.1.8 - FCFS <42, endTime >

The summary of this process:

- **The slicing lines are**: [42, 41, 40, 37, 35, 33, 31, 30, 29, 28, 27, 24, 23, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 6, 4, 3, 2,1]
- **The data dependency lines are:** [37, 35, 24, 23, 21, 18, 6, 4, 3]
- **The variable dependencies:** [endTime, currentTime, arrivalTime, burstTime]

E. A slicing criterion of a program FCFS is <42; turnaroundTime>

```
#include <stdio.h>
typedef struct {
    int arrivalTime;
    int burstTime;
    int startTime;
    int endTime;
    int waitTime;
    int turnaroundTime;
    int id;
} Process;
void calculateTimes(Process processes[], int numProcesses) {
    for (int i = 0; i < numProcesses - 1; i++) {
        for (int j = 0; j < numProcesses - i - 1; j++) {
            if (processes[j].arrivalTime > processes[j + 1].arrivalTime) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;}}}
    int currentTime = 0;
    for (int i = 0; i < numProcesses; i++) {
        if (currentTime < processes[i].arrivalTime)
            currentTime = processes[i].arrivalTime;
        processes[i].startTime = currentTime;
        currentTime += processes[i].burstTime;
        processes[i].endTime = currentTime;
        processes[i].turnaroundTime = processes[i].endTime - processes[i].arrivalTime;
        processes[i].waitTime = processes[i].startTime - processes[i].arrivalTime;}}
int main() {
    int numProcesses;
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);
    Process processes[numProcesses];
    int totalTurnaroundTime = 0, totalWaitTime = 0;
    for (int i = 0; i < numProcesses; i++) {
        printf("\nEnter the arrival time for process %d: ", i + 1);
        scanf("%d", &processes[i].arrivalTime);
        printf("Enter the burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burstTime);
        processes[i].id = i + 1;}
    calculateTimes(processes, numProcesses);
    printf("\nProcess\tStart Time\tEnd Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < numProcesses; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].startTime, processes[i].endTime, processes[i].turnaroundTime, processes[i].waitTime);
        totalTurnaroundTime += processes[i].turnaroundTime;
        totalWaitTime += processes[i].waitTime;}
    double averageTurnaroundTime = (double)totalTurnaroundTime / numProcesses;
    double averageWaitTime = (double)totalWaitTime / numProcesses;
    printf("\nAverage Turnaround Time: %.2lf\n", averageTurnaroundTime);
    printf("Average Waiting Time: %.2lf\n", averageWaitTime);
    return 0;}
```

Figure 2.1.9- <42; turnaroundTime>

To perform static slicing on the "turnaroundTime" variable, we initiate the process by identifying the program point from which the slicing will begin. In this case, we have designated line 42 in Figure 2.1.9 as the starting point for conducting backward slicing. Next, we trace back from the identified program point and examine all statements that affect the value of "turnaroundTime" and remove the statements that may not affect the value of this variable.

Our slicing begins from the main function, specifically from lines 42 to 40, which are responsible for printing the value of the "turnaroundTime" variable. Line 39 is necessary for calculating the "turnaroundTime" variable, while we skip lines 38 to 33 in the main function during the initial examination, as there appear to be no apparent connections between them and the "turnaroundTime" variable. Line 32 is removed as it does not contribute to the computation of "turnaroundTime". On the other hand, line 31 impacts the calculation of "turnaroundTime" by determining the number of processes to be created, each carrying a value of "turnaroundTime". Lines 30 to 28 prompt the user to input the number of processes that will utilize "turnaroundTime", thus affecting its computation.

We then backtrack to the calculateTime function, specifically line 25, where the values of "turnaroundTime" are calculated based on two variables: "endTime" and "arrivalTime." Consequently, we need to calculate the "endTime" variable, as evident in line 24, where "endTime" depends on "currentTime." Lines 23 to 18 are responsible for calculating the value of "currentTime," which relies on the "arrivalTime." The "arrivalTime" indirectly impacts the computation of "endTime" through lines 17 to 11, which ensures that the processes enter the queue based on their arrival time. Lines 10 to 2 encompass the structure where "turnaroundTime," "arrivalTime," "endTime," and "burstTime" are initialized, and line 1 is crucial for executing the C file.

Finally, we return to the main method, armed with the knowledge that "turnaroundTime" depends on "arrivalTime" and "burstTime." Lines 37 to 33 are responsible for obtaining the values of "arrivalTime" and "burstTime" from the user as input, indirectly influencing the computation involving "turnaroundTime". The final code is shown in Figure 2.1.10.

```c
#include <stdio.h>

typedef struct {
    int arrivalTime;
    int burstTime;
    int endTime;
    int turnaroundTime;
} Process;

void calculateTimes(Process processes[], int numProcesses) {
    for (int i = 0; i < numProcesses - 1; i++) {
        for (int j = 0; j < numProcesses - i - 1; j++) {
            if (processes[j].arrivalTime > processes[j + 1].arrivalTime) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }

    int currentTime = 0;

    for (int i = 0; i < numProcesses; i++) {
        if (currentTime < processes[i].arrivalTime)
            currentTime = processes[i].arrivalTime;

        currentTime += processes[i].burstTime;
        processes[i].endTime = currentTime;

        processes[i].turnaroundTime = processes[i].endTime - processes[i].arrivalTime;
    }
}
```

```c
int main() {
    int numProcesses;
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);

    Process processes[numProcesses];

    for (int i = 0; i < numProcesses; i++) {
        printf("\nEnter the arrival time for process %d: ", i + 1);
        scanf("%d", &processes[i].arrivalTime);

        printf("Enter the burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burstTime);
    }

    calculateTimes(processes, numProcesses);

    printf("\nEnd Time\tTurnaround Time\n");
    for (int i = 0; i < numProcesses; i++) {
        printf("%d\t\t%d\n", processes[i].endTime, processes[i].turnaroundTime);
    }

    return 0;
}
```

Figure 2.1.10- <42; turnaroundTime>

The summary of this process:

- **The slicing lines are**: [42, 41,40, 37, 35, 33, 31, 30, 29, 28, 27, 25, 24, 23, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 8, 6, 4, 3, 2,1]
- **The data dependency lines are:** [37, 35, 25, 24, 23, 21, 18, 8, 6, 4, 3]
- **The variable dependencies:** [turnaroundTime, endTime, currentTime, arrivalTime, burstTime]

F. A slicing criterion of a program FCFS is <42, waitTime >

```
#include <stdio.h>
typedef struct {
    int arrivalTime;
    int burstTime;
    int startTime;
    int endTime;
    int waitTime;
    int turnaroundTime;
    int id;
} Process;
void calculateTimes(Process processes[], int numProcesses) {
    for (int i = 0; i < numProcesses - 1; i++) {
        for (int j = 0; j < numProcesses - i - 1; j++) {
            if (processes[j].arrivalTime > processes[j + 1].arrivalTime) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;}}}
    int currentTime = 0;
    for (int i = 0; i < numProcesses; i++) {
        if (currentTime < processes[i].arrivalTime)
            currentTime = processes[i].arrivalTime;
        processes[i].startTime = currentTime;
        currentTime += processes[i].burstTime;
        processes[i].endTime = currentTime;
        processes[i].turnaroundTime = processes[i].endTime - processes[i].arrivalTime;
        processes[i].waitTime = processes[i].startTime - processes[i].arrivalTime;}}
int main() {
    int numProcesses;
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);
    Process processes[numProcesses];
    int totalTurnaroundTime = 0, totalWaitTime = 0;
    for (int i = 0; i < numProcesses; i++) {
        printf("\nEnter the arrival time for process %d: ", i + 1);
        scanf("%d", &processes[i].arrivalTime);
        printf("Enter the burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burstTime);
        processes[i].id = i + 1;}
    calculateTimes(processes, numProcesses);
    printf("\n\nProcess\tstart time\tend time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < numProcesses; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].startTime, processes[i].endTime, processes[i].turnaroundTime, processes[i].waitTime);
        totalTurnaroundTime += processes[i].turnaroundTime;
        totalWaitTime += processes[i].waitTime;}
    double averageTurnaroundTime = (double)totalTurnaroundTime / numProcesses;
    double averageWaitTime = (double)totalWaitTime / numProcesses;
    printf("\nAverage Turnaround Time: %.2lf\n", averageTurnaroundTime);
    printf("Average Waiting Time: %.2lf\n", averageWaitTime);
    return 0;}
```

Figure 2.1.11 - FCFS <42, waitTime >

To perform static slicing on the "waitTime" variable, we initiate the process by identifying the program point from which the slicing will begin. In this scenario, we have designated line 42 in Figure 2.1.11 as the starting point for conducting backward slicing. Subsequently, we backtrack from the determined program point and thoroughly analyze all instructions that influence the value of the "waitTime" variable at line 42. We eliminate the portions of the code that remain unaffected by the removal of "waitTime".

Our slicing begins from the main function, specifically from lines 42 to 40, which are responsible for printing the value of the "waitTime" variable. Line 39 is necessary for calculating the "waitTime" variable, while we skip lines 38 to 33 in the main function during the initial examination, as there appear to be no apparent connections between these lines and the "waitTime" variable. Line 32 is removed as it does not contribute to the computation of "waitTime". On the other hand, line 31 impacts the calculation of "waitTime" by determining the number of processes to be created, each carrying a value of "waitTime". Lines 30 to 28 prompt the user to input the number of processes that will utilize "waitTime," thus affecting its computation.

We then backtrack to the calculateTime function, specifically line 26, where the values of "waitTime" are calculated based on two variables: "startTime" and "arrivalTime." Consequently, we need to calculate the "startTime" variable, as evident in line 22, where "startTime" depends on "currentTime." Lines 23 to 18 are responsible for calculating the value of "currentTime," which relies on the "arrivalTime." The "arrivalTime" indirectly impacts the computation of "startTime" through lines 17 to 11, which ensures that the processes enter the queue based on their arrival time. Lines 10 to 2 encompass the structure where "waitTime," "arrivalTime," "startTime," and "burstTime" are initialized, and line 1 is crucial for executing the C file.

Finally, we return to the main method, armed with the knowledge that "waitTime" depends on "arrivalTime" and "burstTime." Lines 37 to 33 are responsible for obtaining the values of "arrivalTime" and "burstTime" from the user as input, indirectly influencing the computation involving "waitTime." The final code is shown in Figure 2.1.12.

```c
typedef struct {
    int arrivalTime;
    int burstTime;
    int startTime;
    int waitTime;
} Process;

void calculateTimes(Process processes[], int numProcesses) {
    // Bubble sort processes based on arrival times
    for (int i = 0; i < numProcesses - 1; i++) {
        for (int j = 0; j < numProcesses - i - 1; j++) {
            if (processes[j].arrivalTime > processes[j + 1].arrivalTime) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }

    int currentTime = 0;

    for (int i = 0; i < numProcesses; i++) {
        if (currentTime < processes[i].arrivalTime)
            currentTime = processes[i].arrivalTime;

        processes[i].startTime = currentTime;
        currentTime += processes[i].burstTime;

        processes[i].waitTime = processes[i].startTime - processes[i].arrivalTime;
    }
}
```

```c
int main() {
    int numProcesses;
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);

    Process processes[numProcesses];

    for (int i = 0; i < numProcesses; i++) {
        printf("\nEnter the arrival time for process %d: ", i + 1);
        scanf("%d", &processes[i].arrivalTime);

        printf("Enter the burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burstTime);

    }

    calculateTimes(processes, numProcesses);

    printf("\nWaiting Time\n");
    for (int i = 0; i < numProcesses; i++) {
        printf("%d\n",processes[i].waitTime);

    }

    return 0;
}
```

Figure 2.1.12- FCFS <42, waitTime >

The summary of this process:

- **The slicing lines are**: [2, 3, 4, 5, 7, 10, 11, 12,13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 26, 28, 29, 30, 31, 33, 34, 35, 36, 37, 39, 40, 42]
- **The data dependency lines are:** [37, 35, 26, 23, 22, 21, 18, 7, 5, 4, 3]
- **The variable dependencies:** [startTime, currentTime, arrivalTime, burstTime, arrivalTime]

G. A slicing criterion of a program FCFS is <30; NumProcess>

```c
#include <stdio.h>
typedef struct {
    int arrivalTime;
    int burstTime;
    int startTime;
    int endTime;
    int waitTime;
    int turnaroundTime;
    int id;
} Process;
void calculateTimes(Process processes[], int numProcesses) {
    for (int i = 0; i < numProcesses - 1; i++) {
        for (int j = 0; j < numProcesses - i - 1; j++) {
            if (processes[j].arrivalTime > processes[j + 1].arrivalTime) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;}}}
    int currentTime = 0;
    for (int i = 0; i < numProcesses; i++) {
        if (currentTime < processes[i].arrivalTime)
            currentTime = processes[i].arrivalTime;
        processes[i].startTime = currentTime;
        currentTime += processes[i].burstTime;
        processes[i].endTime = currentTime;
        processes[i].turnaroundTime = processes[i].endTime - processes[i].arrivalTime;
        processes[i].waitTime = processes[i].startTime - processes[i].arrivalTime;}}
int main() {
    int numProcesses;
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);
    Process processes[numProcesses];
    int totalTurnaroundTime = 0, totalWaitTime = 0;
    for (int i = 0; i < numProcesses; i++) {
        printf("\nEnter the arrival time for process %d: ", i + 1);
        scanf("%d", &processes[i].arrivalTime);
        printf("Enter the burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burstTime);
        processes[i].id = i + 1;}
    calculateTimes(processes, numProcesses);
    printf("\nProcess\tStart Time\tEnd Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < numProcesses; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].startTime, processes[i].endTime, processes[i].turnaroundTime, processes[i].waitTime);
        totalTurnaroundTime += processes[i].turnaroundTime;
        totalWaitTime += processes[i].waitTime;}
    double averageTurnaroundTime = (double)totalTurnaroundTime / numProcesses;
    double averageWaitTime = (double)totalWaitTime / numProcesses;
    printf("\nAverage Turnaround Time: %.2lf\n", averageTurnaroundTime);
    printf("Average Waiting Time: %.2lf\n", averageWaitTime);
    return 0;}
```

Figure 2.1.13 - FCFS <30; NumProcess>

```c
#include <stdio.h>
int main() {
    int numProcesses;
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);
    return 0;
}
```

To perform static slicing on the “numProcesses” variable, we first need to determine the program point where the slicing will begin. In this case, let's choose line 30 in Figure 2.1.13 as the starting point for backward slicing. Next, we trace back from the identified program point and examine all instructions and predicates that affect the value of “numProcesses”.

Figure 2.1.14- Output of FCFS <30; NumProcess>

We can see that the value of “numProcesses” is obtained from user input using the “scanf” function at line 30. Therefore, line 30 plays a crucial role in determining the value of “numProcesses”. In other words, the lines 30 to 28 prompt the user to enter the number of processes that will utilize “numProcesses”, thus impacting its computation. Finally, lines from 26 to 2 do not affect the computation of “numProcesses”, but they utilize the value of “numProcesses”to calculate other variables. So, we can remove these lines as well. The final code is shown in Figure 2.1.14.

The summary of this process:

- **The slicing lines are**: [27, 28, 29, 30]
- **The data dependency lines are:** [28, 29, 30]
- **The variable dependencies:** This variable does not depend on the other one.

2.2 Dynamic Slicing

In this section, we applied dynamic slicing, specifically backward slicing, to two variables: end time, and NumProcess. For each variable, we provided the reduced code as well as an explanation.

Note: *The grey colour represents data flow dependencies, while the blue colour represents control flow dependencies.*

A. Slicing criterion of a program FCFS is < [1, [1], [3]], 42, endTime >

In this sample, we began slicing at line 42 for the variable "endTime" with an input array of [2, [1], [3]]. Taking advantage of the static slicing result for this variable specifically, the static slicing line [42, 41, 40, 39, 37, 35, 33, 31, 30, 29, 28, 27, 24, 23, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 6, 4, 3, 2, 1] we started from line 42 and traced back to line 41. From there, we continued up to line 40, where a print statement is executed. Moving up, we reached line 39, which enters the calculateTimes method. Starting from line 24, which indicates that "endTime" depends on "currentTime," we considered lines 6, 2, and 1 for “endTime” as well.

Taking into account the "currentTime" variable, we found ourselves inside the loop at lines 19, 20, and 21. Line 21 shows that "currentTime" depends on "arrivalTime," and line 23 indicates a dependency on "burstTime." Considering line 18, we identified no relevance from lines 12 to 17. Going up to line 11, 10, 4, and 3, we considered lines 37 and 35 for "arrivalTime" and "burstTime," as well as lines 33, 31, 30, 29, 28, and 27 for the variables in the condition of the for loops. The resulting Figure 2.2.1 illustrates that:

```
#include <stdio.h>
typedef struct {
    int arrivalTime;
    int burstTime;
    int startTime;
    int endTime;
    int waitTime;
    int turnaroundTime;
    int id;
} Process;
void calculateTimes(Process processes[], int numProcesses) {
    for (int i = 0; i < numProcesses - 1; i++) {
        for (int j = 0; j < numProcesses - i - 1; j++) {
            if (processes[j].arrivalTime > processes[j + 1].arrivalTime) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;}}}
    int currentTime = 0;
    for (int i = 0; i < numProcesses; i++) {
        if (currentTime < processes[i].arrivalTime)
            currentTime = processes[i].arrivalTime;
        processes[i].startTime = currentTime;
        currentTime += processes[i].burstTime;
        processes[i].endTime = currentTime;
        processes[i].turnaroundTime = processes[i].endTime - processes[i].arrivalTime;
        processes[i].waitTime = processes[i].startTime - processes[i].arrivalTime;}}
int main() {
    int numProcesses;
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);
    Process processes[numProcesses];
    int totalTurnaroundTime = 0, totalWaitTime = 0;
    for (int i = 0; i < numProcesses; i++) {
        printf("\nEnter the arrival time for process %d: ", i + 1);
        scanf("%d", &processes[i].arrivalTime);
        printf("Enter the burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burstTime);
        processes[i].id = i + 1;}
    calculateTimes(processes, numProcesses);
    printf("\nProcess\tStart Time\tEnd Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < numProcesses; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].startTime, processes[i].endTime, processes[i].turnaroundTime, processes[i].waitTime);
        totalTurnaroundTime += processes[i].turnaroundTime;
        totalWaitTime += processes[i].waitTime;}
    double averageTurnaroundTime = (double)totalTurnaroundTime / numProcesses;
    double averageWaitTime = (double)totalWaitTime / numProcesses;
    printf("\nAverage Turnaround Time: %.2lf\n", averageTurnaroundTime);
    printf("Average Waiting Time: %.2lf\n", averageWaitTime);
    return 0;}
```

Figure 2.2.1 - Output of FCFS < [1, [1], [3]], 42, endTime >

The summary of this process:

- The slicing lines as [42, 41, 40, 39, 24, 6, 2, 1, 18, 19, 20, 21, 23, 11, 10, 4, 3, 37, 35, 33, 31, 30, 29, 28, 27].
- The data dependency lines are [37, 35, 24, 23, 21, 18, 6, 4, 3].
- The variable dependencies are ["endTime", "currentTime", "arrivalTime", "burstTime"].

As observed, for this particular value of the variable, the slicing line differs from the static one, where lines from 17 to 12 are not executed. However, the data dependency lines remain the same, as well as the variable dependencies.

B. A slicing criterion of a program FCFS is <1, 30, NumProcess>

In this sample, we initiated slicing at line 30 for the variable "NumProcess," which has a value equal to one. Similar to the approach taken for "endTime," we took advantage of the static slicing result for this variable. The static slicing lines are: [27, 28, 29, 30]. We began eliminating lines that would not be executed when the "NumProcess" value is equal to one. Our approach was as follows: starting at line 30, we traced back to line 29, which is essential, and then to line 28, and line 27, which also affects the "NumProcess" variable and happens to be the last line in the program. The result is shown in Figure 2.2.2.

```
1  #include <stdio.h>
2  typedef struct {
3      int arrivalTime;
4      int burstTime;
5      int startTime;
6      int endTime;
7      int waitTime;
8      int turnaroundTime;
9      int id;
10 } Process;
11 void calculateTimes(Process processes[], int numProcesses) {
12     for (int i = 0; i < numProcesses - 1; i++) {
13         for (int j = 0; j < numProcesses - i - 1; j++) {
14             if (processes[j].arrivalTime > processes[j + 1].arrivalTime) {
15                 Process temp = processes[j];
16                 processes[j] = processes[j + 1];
17                 processes[j + 1] = temp;}}}
18     int currentTime = 0;
19     for (int i = 0; i < numProcesses; i++) {
20         if (currentTime < processes[i].arrivalTime)
21             currentTime = processes[i].arrivalTime;
22         processes[i].startTime = currentTime;
23         currentTime += processes[i].burstTime;
24         processes[i].endTime = currentTime;
25         processes[i].turnaroundTime = processes[i].endTime - processes[i].arrivalTime;
26         processes[i].waitTime = processes[i].startTime - processes[i].arrivalTime;}}
27 int main() {
28     int numProcesses;
29     printf("Enter the number of processes: ");
30     scanf("%d", &numProcesses);
31     Process processes[numProcesses];
32     int totalTurnaroundTime = 0, totalWaitTime = 0;
33     for (int i = 0; i < numProcesses; i++) {
34         printf("\nEnter the arrival time for process %d: ", i + 1);
35         scanf("%d", &processes[i].arrivalTime);
36         printf("Enter the burst time for process %d: ", i + 1);
37         scanf("%d", &processes[i].burstTime);
38         processes[i].id = i + 1;}
39     calculateTimes(processes, numProcesses);
40     printf("\nProcess\tStart Time\tEnd Time\tTurnaround Time\tWaiting Time\n");
41     for (int i = 0; i < numProcesses; i++) {
42         printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].startTime, processes[i].endTime, processes[i].turnaroundTime, processes[i].waitTime);
43         totalTurnaroundTime += processes[i].turnaroundTime;
44         totalWaitTime += processes[i].waitTime;}
45     double averageTurnaroundTime = (double)totalTurnaroundTime / numProcesses;
46     double averageWaitTime = (double)totalWaitTime / numProcesses;
47     printf("\nAverage Turnaround Time: %.2lf\n", averageTurnaroundTime);
48     printf("Average Waiting Time: %.2lf\n", averageWaitTime);
49     return 0;}
```

Figure 2.2.2 - FCFS <1, 30; NumProcess>

The summary of this process:

- **The slicing lines are**: [27, 28, 29, 30]
- **The data dependency lines are:** [28, 29, 30]
- **The variable dependencies:** This variable does not depend on the other one.

It is noteworthy that the dynamic slicing result for this variable with a value equal to one is identical to the static one, which was expected because this variable doesn't depend on other variables.

2.3 Automatic Slicing Tool Design and Implementation

A. Design for Automatic Slicing

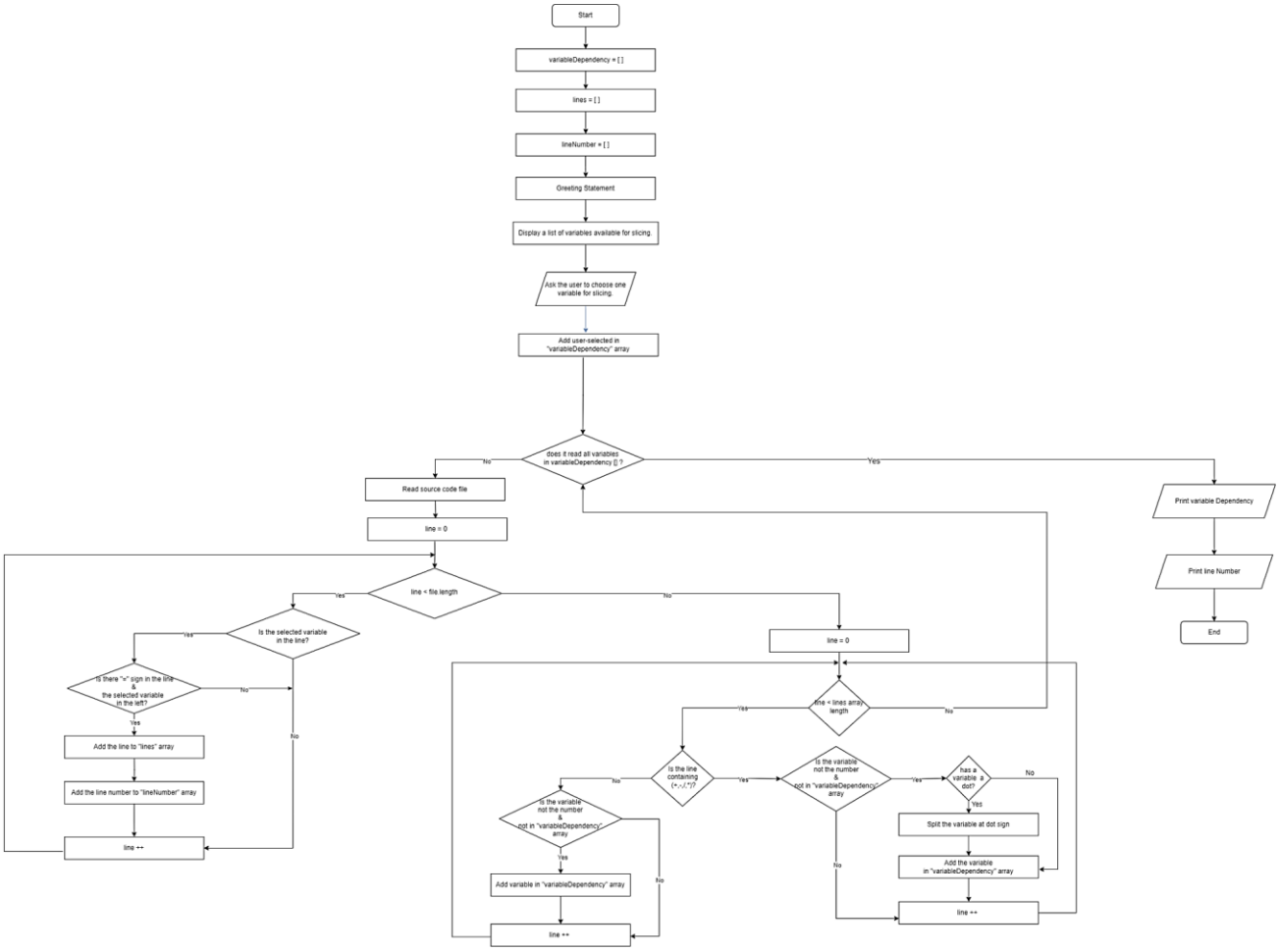


Figure 2.3.1 – Flowchart for Automatic Slicing

Link for flowchart of automatic slicing:

https://drive.google.com/file/d/15CQ1Akp22q0KxE3gGWvIOCcHyPD91yly/view?usp=sharing

B. Pseudocode for Automatic Slicing

Constraints:

1. The code should be written in Python.
2. There shouldn't be a shorthand method to increment a variable like i+=1. Instead, you should use a complete assignment statement such as i = i + 1.
3. Unassigned variables are not supported, and any initialized variable must be assigned a value.
4. Print statements are disregarded in this version of the algorithms.

Pseudocode:

1. Greeting Statement.
2. Display a list of variables available for slicing: arrivalTime, burstTime, startTime, endTime, waitTime, turnaroundTime, and NumProcess.
3. Ask the user to choose one variable from the provided list.
4. Add the variable that is selected by the user in the "variableDependency" array.
5. Do a "For Loop" in "variableDependency" array.
    a. Analysis (Read) the entire file and search for the user-selected variable in the button-top approach.
    b. If the line contains the variable that user selected:
        i. Check if the line contains an equal sign && the variable on the left.
            1. Add the line in the "lines" array.
            2. Add the line number in the "lineNumber" array.
    c. Do a "For Loop" in the "lines" array.
        i. Check if the line contains mathematical operations (+, -, /, *)
            1. If it has, check the variables to the left and right of these operations
                a. If the variables are numbers, ignore them
                b. If the variables are not numbers:
                    1. Split the variables at the dot sign, and take the part after the dot sign (if any).
                    2. Check if the variables are not already in the "variableDependency" array.
                        a. Add the variable to the "variableDependency" array.
        ii. If it doesn’t have mathematical operations.
            1. Check there is no number after the equal sign && it is not in the "variableDependency" array.
                a. Add the variable to the "variableDependency" array.
6. Display to the user the variables that depend on the variable that he/she chose with their line numbers.

C. Implementing Automatic Slicing

We have developed a powerful tool using the Python programming language to perform static slicing using backward techniques and fix breakpoints. In this version of the tool, there are certain limitations and restrictions to ensure its proper functioning, which are:

- The tool requires the code to be written in the Python programming language.
- It does not support shorthand methods like 'i += 1' and requires the use of complete assignment statements.
- Unassigned variables are not supported, and any initialized variable must be assigned a value.
- Print statements are disregarded during the slicing process.

The tool primarily focuses on data dependency rather than control flow. It analyzes the relationships between variables and identifies dependencies to provide insights into how data flows through the code.

In addition to the previously mentioned features, our tool provides an output that includes variable dependencies and their corresponding allocated lines.

Exciting Findings 💡:

We ran the tool in our program, but in a different programming language - Python, to be precise. Guess what? We obtained the same data dependency as in the C program. This means that the slicing techniques are independent of the programming languages used.

```python
import re

def reverse_file(source_file, target_file):
    with open(source_file, "r") as original_file:
        original_lines = original_file.readlines()

    reversed_lines = reversed(original_lines)

    with open(target_file, "w") as reversed_file:
        reversed_file.writelines(reversed_lines)

def count_lines(file_path):
    with open(file_path, 'r') as file:
        line_count = sum(1 for line in file)
    return line_count

variableDependency = []
lines = []
lineNumber = []
excluded_vars = ["i", "j", "k"]
variables = ["arrivalTime", "burstTime", "startTime", "endTime", "waitTime", "turnaroundTime", "numProcess"]

sourceCode = "fifo.py"
targetFile = "reversedfifo.py"
reverse_file(sourceCode, targetFile)
line_count = count_lines(targetFile) + 1
```

```
print("\033[1;34mWelcome to our Static Variable Slicing Tool (Backwards Technique)! 🚀✨")
print("Here, we'll display variable dependencies and line numbers. Let's get started!🔥 \033[0m")
print("\033[1;90mNote: This tool has some restrictions:\033[0m")
print("1. The code should be written in Python.")
print("2. Shorthand methods like 'i += 1' are not supported. Please use complete assignment statements.")
print("3. Unassigned variables are not supported, and any initialized variable must be assigned a value.")
print("4. Print statements are disregarded.")

print("\033[1;34mBelow is the list of variables 👇 :\033[0m")
for i, variable in enumerate(variables, start=1):
    print(f"{i}. {variable}")


choice = input("\033[1;34mEnter a one variable you'd like to slice from the List: \033[0m")

if choice in variables:
    variableDependency.append(choice)
else:
    print("\033[1;90m⚠️  The chosen variable is not in the list."
          " Please select a variable from the available options next time.\033[0m")
    exit(0)

for var in variableDependency:
    with open(targetFile, "r") as file:
            for j, line in enumerate(file, start=1):
                i = line_count - j
                if var in line:
                    if "=" in line and var in line.split('=')[0]:
                        lines.append(line)
                        lineNumber.append(i)

            for i, line in enumerate(lines):
                delimiters = r"(?=[- + * / = ])"
                tokens = re.split(delimiters, line)
                tokens = [token.rstrip('\n') for token in tokens if token.strip()]
                for i, token in enumerate(tokens):
                    if '=' == token:
                        var_part = tokens[i + 1].replace(" ", "")
                        if "." in var_part:
                            var_part = var_part.split(".")[1]
                        if var_part.isidentifier() and var_part not in variableDependency:
                            variableDependency.append(var_part)

                    if token in "+-*/":
                        prev_token = tokens[i - 1]
                        if "." in prev_token:
                            prev_token = prev_token.split(".")[1]
                        next_token = tokens[i + 1]
                        if "." in next_token:
                            next_token = next_token.split(".")[1]

                        if prev_token.isidentifier() and prev_token not in variableDependency:
                            variableDependency.append(prev_token)
                        if next_token.isidentifier() and next_token not in variableDependency:
                            variableDependency.append(next_token)
```

Figure 2.3.2- Code for Automatic Slicing Tool

2.4 Comparison between Manual Slicing and Automatic Slicing Results

A. A slice for "arrivalTime" variable

```
Welcome to our Static Variable Slicing Tool (Backwards Technique)! 🚀✨
Here, we'll display variable dependencies and line numbers. Let's get started!🔥
Note: This tool has some restrictions:
1. The code should be written in Python.
2. Shorthand methods like 'i += 1' are not supported. Please use complete assignment statements.
3. Unassigned variables are not supported, and any initialized variable must be assigned a value.
4. Print statements are disregarded.
Below is the list of variables 👇 :
1. arrivalTime
2. burstTime
3. startTime
4. endTime
5. waitTime
6. turnaroundTime
7. numProcess
Enter a one variable you'd like to slice from the List: arrivalTime
Variable Dependencies:
['arrivalTime']
Line Numbers:
[42, 4]
Happy slicing! ✂️
```

Figure 2.4.1- Results of the "arrivalTime" variable slicing

B. A slice for "burstTime" variable

```
Welcome to our Static Variable Slicing Tool (Backwards Technique)! 🚀✨
Here, we'll display variable dependencies and line numbers. Let's get started!🔥
Note: This tool has some restrictions:
1. The code should be written in Python.
2. Shorthand methods like 'i += 1' are not supported. Please use complete assignment statements.
3. Unassigned variables are not supported, and any initialized variable must be assigned a value.
4. Print statements are disregarded.
Below is the list of variables 👇 :
1. arrivalTime
2. burstTime
3. startTime
4. endTime
5. waitTime
6. turnaroundTime
7. numProcess
Enter a one variable you'd like to slice from the List: burstTime
Variable Dependencies:
['burstTime']
Line Numbers:
[43, 5]
Happy slicing! ✂️
```

Figure 2.4.2- Results of the "burstTime" variable slicing

C. A slice for "startTime" variable

```
Welcome to our Static Variable Slicing Tool (Backwards Technique)! 🚀✨
Here, we'll display variable dependencies and line numbers. Let's get started!🔥
Note: This tool has some restrictions:
1. The code should be written in Python.
2. Shorthand methods like 'i += 1' are not supported. Please use complete assignment statements.
3. Unassigned variables are not supported, and any initialized variable must be assigned a value.
4. Print statements are disregarded.
Below is the list of variables 👇 :
1. arrivalTime
2. burstTime
3. startTime
4. endTime
5. waitTime
6. turnaroundTime
7. numProcess
Enter a one variable you'd like to slice from the List: startTime
Variable Dependencies:
['startTime', 'currentTime', 'burstTime', 'arrivalTime']
Line Numbers:
[26, 6, 27, 24, 20, 43, 5, 42, 4]
Happy slicing! ✂️
```

Figure 2.4.3- Results of the "startTime" variable slicing

D. A slice for "endTime" variable

```
Welcome to our Static Variable Slicing Tool (Backwards Technique)! 🚀✨
Here, we'll display variable dependencies and line numbers. Let's get started!🔥
Note: This tool has some restrictions:
1. The code should be written in Python.
2. Shorthand methods like 'i += 1' are not supported. Please use complete assignment statements.
3. Unassigned variables are not supported, and any initialized variable must be assigned a value.
4. Print statements are disregarded.
Below is the list of variables 👇 :
1. arrivalTime
2. burstTime
3. startTime
4. endTime
5. waitTime
6. turnaroundTime
7. numProcess
Enter a one variable you'd like to slice from the List: endTime
Variable Dependencies:
['endTime', 'currentTime', 'burstTime', 'arrivalTime']
Line Numbers:
[28, 7, 27, 24, 20, 43, 5, 42, 4]
Happy slicing! ✂️
```

Figure 2.4.4- Results of the "endTime" variable slicing

E. A slice for "turnaroundTime" variable

```
Welcome to our Static Variable Slicing Tool (Backwards Technique)! 🚀✨
Here, we'll display variable dependencies and line numbers. Let's get started!🔥
Note: This tool has some restrictions:
1. The code should be written in Python.
2. Shorthand methods like 'i += 1' are not supported. Please use complete assignment statements.
3. Unassigned variables are not supported, and any initialized variable must be assigned a value.
4. Print statements are disregarded.
Below is the list of variables 👇 :
1. arrivalTime
2. burstTime
3. startTime
4. endTime
5. waitTime
6. turnaroundTime
7. numProcess
Enter a one variable you'd like to slice from the List: turnaroundTime
Variable Dependencies:
['turnaroundTime', 'endTime', 'arrivalTime', 'currentTime', 'burstTime']
Line Numbers:
[30, 9, 28, 7, 42, 4, 27, 24, 20, 43, 5]
Happy slicing! ✂️
```

Figure 2.4.5- Results of the "turnaroundTime" variable slicing

F. A slice for "waitTime" variable

```
Welcome to our Static Variable Slicing Tool (Backwards Technique)! 🚀✨
Here, we'll display variable dependencies and line numbers. Let's get started!🔥
Note: This tool has some restrictions:
1. The code should be written in Python.
2. Shorthand methods like 'i += 1' are not supported. Please use complete assignment statements.
3. Unassigned variables are not supported, and any initialized variable must be assigned a value.
4. Print statements are disregarded.
Below is the list of variables 👇 :
1. arrivalTime
2. burstTime
3. startTime
4. endTime
5. waitTime
6. turnaroundTime
7. numProcess
Enter a one variable you'd like to slice from the List: waitTime
Variable Dependencies:
['waitTime', 'startTime', 'arrivalTime', 'currentTime', 'burstTime']
Line Numbers:
[31, 8, 26, 6, 42, 4, 27, 24, 20, 43, 5]
Happy slicing! ✂️
```

Figure 2.4.6- Results of the "waitTime" variable slicing

G. A slice for "NumProcess" variable

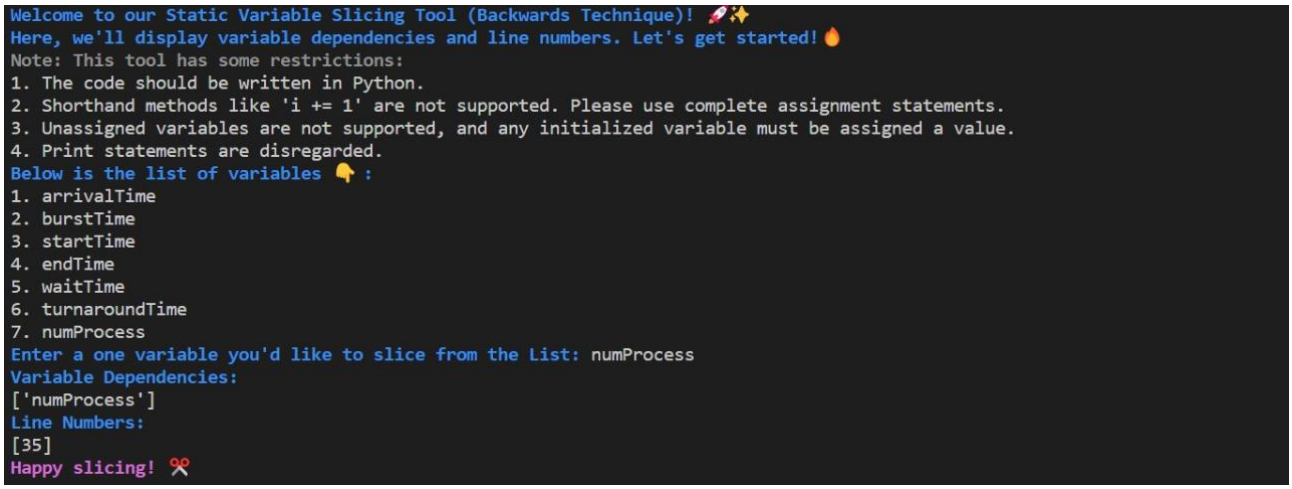


Figure 2.4.7- Results of the "NumProcess" variable slicing

Table 2.3.1 displays the comparison between manual slicing and automatic slicing results. As is evident, both methods yield similar outputs.

	Manual Slicing	Automatic Slicing	Result
arrivalTime	'arrivalTime'	'arrivalTime'	Match ✅
burstTime	'burstTime'	'burstTime'	Match ✅
startTime	startTime, currentTime, arrivalTime, burstTime	startTime, currentTime, arrivalTime, burstTime	Match ✅
endTime	'endTime', 'currentTime', 'arrivalTime', 'burstTime'	'endTime', 'currentTime', 'arrivalTime', 'burstTime'	Match ✅
turnaroundTime	'turnaroundTime', 'endTime', 'arrivalTime', 'currentTime', 'burstTime'	'turnaroundTime', 'endTime', 'arrivalTime', 'currentTime', 'burstTime'	Match ✅
waitTime	'waitTime', 'startTime', 'arrivalTime','currentTime', 'burstTime'	'waitTime', 'startTime', 'arrivalTime','currentTime', 'burstTime'	Match ✅
NumProcess	'numProcess'	'numProcess'	Match ✅

Table 2.4.1- Comparison Between Manual and Automatic Slicing Results

CHAPTER 3: REVERSE ENGINEERING

3.1 Comparison Between Static and Dynamic Slicing

When delving into program slicing for legacy systems, two prominent techniques emerge as the go-to choices: dynamic slicing, and static slicing. These methods present distinct approaches to program analysis, each with its own merits. In the table below, we explore the disparities between dynamic and static slicing techniques, shedding light on their contrasting features:[1]

Dynamic Slicing	Static Slicing
It contains all the statements that actually affect the value of a variable at any point for a particular execution of the program.	It contains all statements that may affect the value of a variable at any point for any arbitrary execution of the program.
Generally smaller.	Generally larger.
It considers only a particular execution of the program.	It considers every possible execution of the program.
Useful for debugging, understanding program behavior, and reducing code to examine.	Useful for understanding program behavior, optimizing code, and detecting potential issues

3.2 The Significance of Program Slicing Technique for Legacy Systems

Program slicing is a technique for extracting relevant portions of a program that directly impact a specific computation or variable [2]. This technique has proven to be particularly beneficial for analysing and maintaining legacy systems, which are often large, complex, and lack thorough documentation. By focusing on specific slices of code, program slicing facilitates various software engineering tasks, including program comprehension, debugging, testing, maintenance, and complexity measurement. Legacy systems benefit significantly from program slicing's ability to isolate relevant code segments. The numerous applications of program slicing highlight its importance in various software engineering contexts:

1. **Program Comprehension:** Understanding legacy systems can be difficult due to their complexity and continuous evolution. Program slicing simplifies this process by isolating the code relevant to a specific point of interest, providing insights into the program's structure and behavior [2]. This helps developers grasp data flow, identify dependencies, and localize potential issues.

2. **Parallelization**: Amorphous program slicing maintains the semantic meaning of the original program while allowing for more aggressive code reduction. It utilizes a broader set of transformation rules, including statement elimination [3]. While beneficial for program comprehension, this technique has also proven effective for program parallelization, enabling the identification of independent software components, and facilitating the extraction of reusable functionalities [2].

3. **Cohesion Measurement:** Slicing plays a crucial role in cohesion measurement within object-oriented programming systems. Encapsulation, a fundamental concept in object-oriented programming, dictates that a well-encapsulated code module encapsulates all necessary data and associated functions for that entity. A program slice captures a thread of execution that is involved in calculating a particular variable [3]. According to the principle of functional cohesion, a component procedure should perform related tasks. A method can be sliced into multiple slices, each corresponding to a different variable [3]. If these slices share significant amounts of code, it is likely that the variables are interrelated.

4. **Debugging**: Slicing Debugging legacy systems is often time-consuming and error-prone due to their intricate design. Program slicing reduces the debugging effort by narrowing down the search for errors to the code that directly affects the observed behaviour. This allows developers to focus on specific areas of the code, saving time and improving debugging accuracy [4].

3.3 Program Slicing Technique VS. Other Reverse Engineering Techniques

To compare program slicing with other reverse engineering techniques, such as lexical analysis, syntactic analysis, control flow analysis, data flow analysis, visualization, and program metrics, we first need to understand the definition of program slicing. Program slicing involves extracting a subset of a program's statements that are contribute to the desired output or relevant to a specific behaviour or, while maintaining the same behaviour as the original program but with a reduced size [5]. Now, let's proceed to the comparison of program slicing with other reverse engineering techniques:

1. **Lexical Analysis:** Lexical analysis is the process of breaking down the source code into tokens, such as keywords and operators, in order to understand the structure of the program [5]. On the other hand, program slicing focuses on extracting relevant parts of the code based on program behaviour, rather than just tokenizing. It helps in debugging, testing, and maintaining software systems.

2. **Syntactic Analysis (also known as parsing):** Syntactic Analysis is the process of examining the structure of a program using grammar rules to understand how various elements relate to one another and to detect syntactic issues [5]. However, program slicing goes beyond syntactic analysis by taking control and data dependencies into account to extract the relevant portion of the code.

3. **Control Flow Analysis and Data Flow Analysis**: Control Flow Analysis helps in understanding the program's execution path by analysing how control moves from one statement to another, including conditionals and loops. In terms of Data Flow Analysis, it concerns how data changes throughout a program, starting from when variables are defined, initialized, and used across different program points [5]. Program slicing uses control flow analysis to extract code statements that are relevant to specific control flow paths, alongside data flow analysis to extract code statements that are data-dependent on specific variables.

4. **Visualization:** Visualization techniques provide graphical representations of static and dynamic aspects of software, aiding in understanding complex relationships within the code [6]. Program slicing can utilize visualization techniques to highlight the extracted code and its connections to the rest of the program.

5. **Program Metric:** Metrics provide quantitative measures of program properties. Several indicators in the software development process such as code size, complexity, coupling, and cohesion [7]. Program slicing, while not a direct metric, can be used as a technique to extract specific code portions for further metric analysis, allowing focused evaluation on relevant parts of the program.

To sum up, program slicing uses control flow analysis to extract code statements and data flow analysis to extract code statements that are data-dependent, making them the basic building blocks for program slicing. Program slicing complements other reverse engineering techniques, such as program metrics, by providing a targeted subset of code for further analysis and understanding. Additionally, program slicing can utilize visualization techniques to highlight the extracted code and its connections to the rest of the program.

3.4 Program Slicing for Legacy System Evaluation: Determining the Optimal Approach (Scrapping, Maintenance, Reengineering, or Replacement)

Programme slicing can be highly valuable in the context of maintenance the legacy system. Due to years of maintenance, legacy systems sometimes become complex and difficult to understand. Slicing technique can be a useful for understand a legacy system, which can help to decide on whether to scrap, maintain, reengineer, or replace the system. Here's how programme slicing can help you determine what to deal with a legacy system:

1. **Scrapping**: The program slicing technique can assist in finding portions of code that are never executed or contribute little to system execution. It also aids in the identification of redundant or irrelevant code. All of these cases could be the ideal candidates for removal from the system. This contributes to decreasing system complexity and eliminating unneeded elements.

2. **Maintenance**: It may be more affordable to maintain and repair the system rather than replace it. Maintainability measures such as code cohesion and coupling can be measured using program slicing. It enables developers to isolate and focus on the system's essential functionality, so ensuring that essential features are kept and well-supported. Maintainers can analyse the impact of changes on other sections of the system by visualizing dependencies using slicing diagrams such as Control flow chart or Data flow chart [8].

3. **Reengineering**: In the context of reengineering, program slicing can identify locations with high coupling; these high-coupling areas may be modified to improve modularity and make the system more adaptable. Reengineering frequently entails enhancing code cohesion to make the system more maintainable. Program slicing assists in identifying areas where code cohesion is insufficient, guiding reengineering efforts to improve code structure [8].

4. **Replacement**: If the slices show that the system is excessively complex, with complicated dependencies and duplicated functionality, replacing the system may be more efficient. In this case, the slices can still be beneficial for understanding the system's functionality and ensuring that the new system includes all relevant features. Understanding dependencies and interfaces with other systems is critical when replacing a system. Program slicing assists in identifying key areas, allowing for a smoother transition to a new system [9].

CHAPTER 4: CONCLUSION

4.1 Conclusion

In conclusion, we applied our knowledge of software maintenance to our project. We began by searching and selecting a legacy code; ultimately, we chose the FCFS algorithm for our project. We then performed two types of slicing: static and dynamic. For static slicing, we applied it to seven variables: "arrivalTime," "burstTime," "startTime," "endTime," "turnaroundTime," "waitTime," and "NumProcess." On the other hand, we did dynamic slicing on two variables: "endTime" and "NumProcess."

Throughout the project, we applied the techniques and concepts learned in our lectures, enhancing our understanding through hands-on application. A particularly challenging part of this project was building our static tool. While there are many tools and IDEs that assist programmers in writing code, there are few tools that aid in reading and analysing existing code, despite the fact that programmers often spend more time reading code than writing it. To address this, we took the step to build the static slicing tool, and the tool is now successfully alive.

Towards the end of the project, we went deeper into differentiating between the two types of slicing techniques we utilized: static and dynamic. Additionally, we explored the advantages of slicing techniques compared to other reengineering methods, highlighting the true power of slicing in legacy systems. Overall, this project provided us with practical experience and solidified our understanding of the course material.

REFERENCES

[1] *Software engineering: Slicing* (2022) *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/software-engineering-slicing/ (Accessed: 17 December 2023).

[2] M. Weiser, "Program Slicing," in IEEE Transactions on Software Engineering, vol. SE-10, no. 4, pp. 352-357, July 1984, doi: 10.1109/TSE.1984.5010248.

[3] S. N. Singh and L. Singh, "Study of current program slicing techniques," 2014 5th International Conference - Confluence the Next Generation Information Technology Summit (Confluence), Noida, India, 2014, pp. 810-814, doi: 10.1109/CONFLUENCE.2014.6949332.

[4] A. Chandra, A. Singhal and A. Bansal, "A study of program slicing techniques for software development approaches," 2015 1st International Conference on Next Generation Computing Technologies (NGCT), Dehradun, India, 2015, pp. 622-627, doi: 10.1109/NGCT.2015.7375196.

[5] Tripathy, P. (2014) *Software evolution and maintenance: A practitioner's approach*, *Google Books*. Available at: https://books.google.com/books/about/Software_Evolution_and_Maintenance.html?id=7XrDBAAAQBAJ (Accessed: 24 November 2023).

[6] *Software visualization* (2023) *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Software_visualization (Accessed: 24 November 2023).

[7] Janani (2023) *What are software metrics? how to track and measure those metrics?*, *Atatus Blog - For DevOps Engineers, Web App Developers and Server Admins.* Available at: https://www.atatus.com/blog/what-are-software-metrics/ (Accessed: 24 November 2023).

[8] G. Colledge, An overview of program slicing, http://www0.cs.ucl.ac.uk/staff/mharman/sf.html

[9] K. Gallagher and J. R. Lyle, "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 751–761, Jan. 1991, doi: 10.1109/32.83912.