Republic of Iraq

And Scientific Research

University of Baghdad,

College of Science

Department of Computer Science

CPU SCHEDULING

A project

Submitted to the Department of Computer Science, College of Science, University of Baghdad in Partial Fulfillment of the Requirements for the Degree of B.Sc. in Computer Science

By

MARWAN ADEL OMAR

MUSTAFA JABER ZWAID

Supervised by

ALAA HARIF

5/5/2011

Acknowledgment

It gives us a great sense of pleasure to present the report of the B.Sc project undertaken during B.Sc pre final year. we owe special debt of gratitude to Mr. Alaa Harif for his constant support and guidance throughout the course of our work, his sincerity, thoroughness and perseverance have been a constant of inspiration for us . It is only his cognizant efforts that our endeavors have seen light of the day.

We also do not like to miss the opportunity to acknowledge the contribution of all faculty members of the department for their kind assistance and cooperation during the developing of our project. Last but not the least; we acknowledge our family and friends of their contribution in the completion of the project.

Abstract

The project entitled "CPU SCHEDULING", is basically a program which simulates the following scheduling algorithms:

1. FCFS (First Come First Served)
2. SJF (Shortest Job First)
3. SRTF(Shortest Remaining Time First)
4. Priority Scheduling
5. Round-Robin

CPU SCHEDULING is a key concept in computer multitasking, multiprocessing operating system and real-time operating system designs. Scheduling refers to the way processes are assigned to run on the available CPUs, since there are typically many more processes running than there are available CPUs.

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. By switching the CPU among processes, the operating system can make the computer more productive. A multiprogramming operating system allows more than one processes to be loaded into the executable memory at a time and for the loaded processes to share the CPU using time-multiplexing.

Scheduling algorithm is the method by which threads, processes or data flows are given access to system resources (e.g. processor time, communications bandwidth). The need for a scheduling algorithm arises from requirement for most modern systems to perform multitasking (execute more than one process at a time) and multiplexing (transmit multiple flows simultaneously).

Table of contents

Chapter One

Introduction

1.1 Introduction

An operating system is a program that manages the hardware and software resources of a computer . It is the first thing that is loaded into memory when we turn on the computer . Without the operating system , each programmer would have to create a way to send data to a printer , tell it how to read a disk file , and how to deal with other programs .

In the beginning programmers needed a way to handle complex input/output operations. The evolution of a computer programs and their complexities required new necessities. Because machines began to become more powerful , the time a program needed to run decreased . However , the time needed for handling off the equipment between different programs became evident and this led to program like DOS . As we can see the acronym DOS stands for Disk Operating System . This confirms that operating systems were originally made to handle these complex input/output operations like communicating among a variety of disk drives.

Earlier computers were not as powerful as they are today. In the early computer systems you would only be able to run one program at a time. For instance , you could not be writing a paper and browsing the internet all at the same time . However , today's operating systems are very capable of handling not only two but multiple applications at the same time . In fact , if a computer is not able to do this it is considered useless by most computer users.

In order for a computer to be able to handle multiple applications simultaneously, there must be an effective way of using the CPU. Several processes may be running at the same time, so there has to be some kind of order to allow each process to get its share of CPU time.

An operating system must allocate computer resources among the potentially competing-requirements of multiple processes. In the case of the processor , the resource to be allocated is execution time on the processor and the means of allocation is scheduling. The scheduling function must be designed to satisfy a number of objectives, including fairness , lack of starvation of any particular process, efficient use of

processor time, and low overhead. In addition, the scheduling function may need to take into account different levels of priority or real-time deadlines for the start or completion of certain process.

Over the years , scheduling has been the focus of intensive research , and many different algorithms have been implemented . Today , the emphasis in scheduling research is on exploiting multiprocessor systems , particularly for multithreaded applications , and real-time scheduling.

In a multiprogramming systems , multiple process exist concurrently in main memory. Each process alternates between using a processor and waiting for some event to occur, such as the completion of an I/O operation. The processor are kept busy by executing one process while the others wait , hence the key to multiprogramming is **scheduling** .

1.2 motivation

the CPU scheduling is one of the important problems in operating systems designing and build a program achieve these algorithms has been a challenge in this field , and because of there is many scheduling algorithms so we choose some of them for enforcement one of Incentives for us to build this program.

1.3 Objective Of The Project

Implementation of the CPU scheduling algorithms existing in operating systems books and researches on real world and calculate average waiting time and turnaround time with drawing a grant chart for algorithms and compare its performance to discover suitable and the best algorithm.

The system provides a clean and convenient way to test the given data and do the analysis of the CPU scheduling algorithms.

For the purpose of analysis and testing the user first specifies each process along with its information such as arrival times and burst time and then algorithms can be computed producing output in appropriate format readability.

CHAPTER Two

Theoretical Issues

CPU SCHEDULING

CPU scheduling is the basis of multi-programmed operating system .By switching the CPU among processes, the operating system can make the computer more productive. A multiprogramming operating system allows more than one processes to be loaded into the executable memory at a time and for the loaded processes to share the CPU using time-multiplexing.

Part of the reason for using multiprogramming is that the operating system itself is implemented as one or more processes , so there must be a way for the operating system and application processes to share the CPU . Another main reason is the need for process to perform I/O operations in the normal course of computation . since I/O operations ordinarily require orders of magnitude more time to complete than do CPU instructions , multiprogramming systems allocate the CPU to another process whenever a process invokes an I/O operation.

Scheduling refers to the way processes are assigned to run on the available CPUs, since there are typically mane more processes running than there are available CPUs.

2.1 BASIC CONCEPTS

In a single- processor system , only one process can run at a time; any others must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some processes running at all time, to maximize CPU utilization. The idea is relatively simple. A processor is executed until it must wait typically for the completion of some I/O request. In a simple computer system , the CPU then just sits idle. All this waiting time is wasted ; no useful work is accomplished . With multiprogramming , we try use this time productively. Several processes are kept in memory at one time. When one process has to wait , the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU.

Scheduling of this kind is a fundamental operating- system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

2.1.1 CPU-I/O BURST CYCLE

The success of CPU scheduling depends on an observed property of processes: process execution consist of a cycle of a CPU execution and I/O wait**.** Processes alternate between these two states.

A process begins with a *CPU burst*, followed by an *I/O burst*, followed by another CPU burst , then another I/O burst , and so on. Eventually, The last CPU burst ends with a system request to terminate the execution.

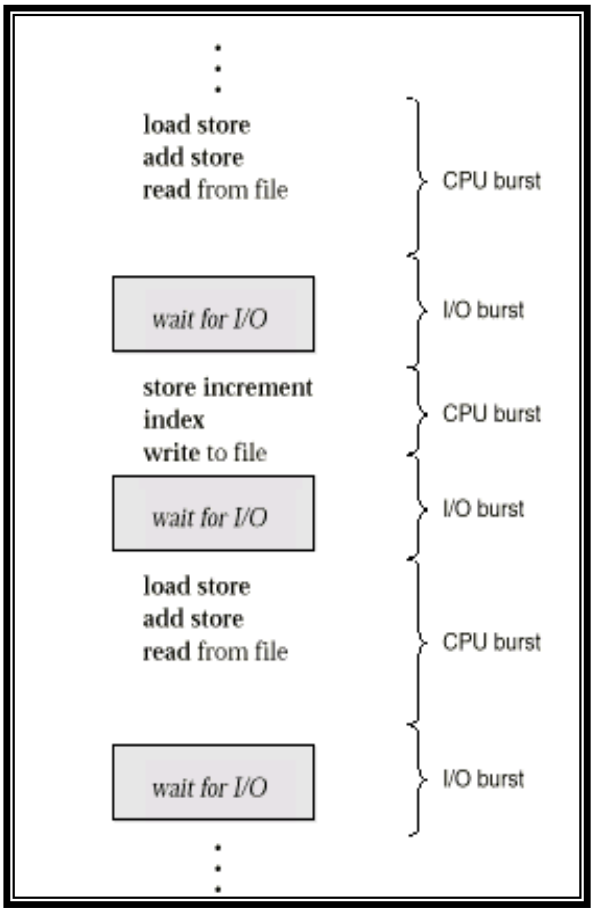
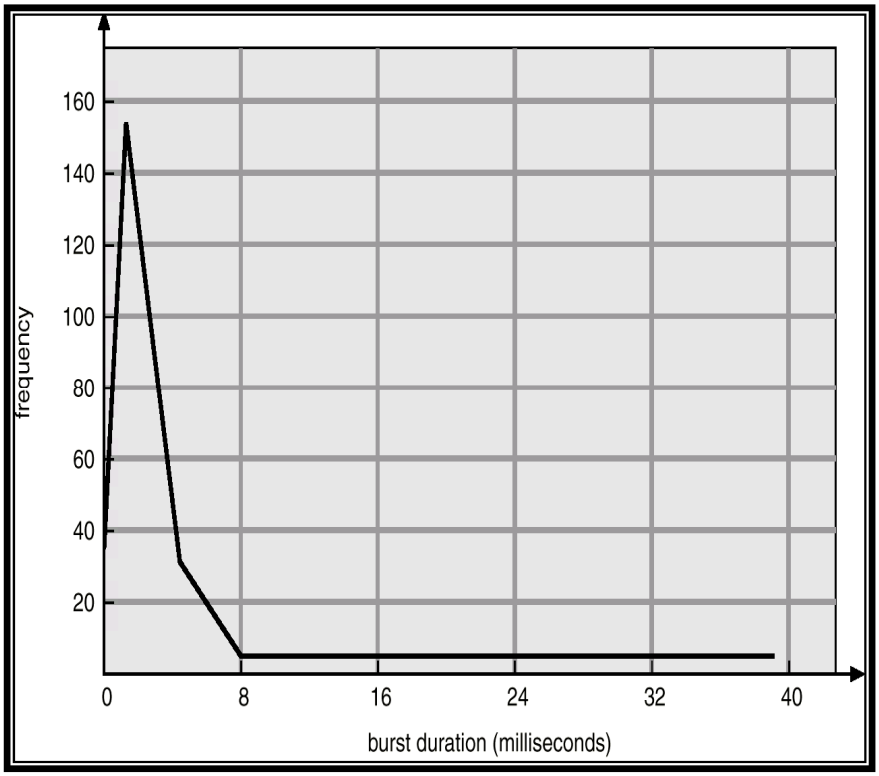


Figure 2.1 Alternating Sequence of CPU and I/O Bursts

The CPU burst durations have been measured extensively. Although they vary greatly from process to process and from computer to computer, they trend to have a frequency curve as following:



Burst duration (milliseconds)

Figure 2.2 Histogram of CPU- burst durations

The curve is generally characterized as exponential or hyper exponential, with a large number of short CPU bursts and small number of long CPU bursts. An I/O bound program typically has many short CPU bursts. A CPU – bound program might have a few long CPU bursts. This distribution can be important in the selection of an appropriate CPU- scheduling algorithm.

2.1.2 CPU SCHEDULER

Whenever, the CPU becomes idle, the operating system must select one of the processes in the ready-queue to be executed. The selection process is carried out the short-term

scheduler or CPU scheduler. The CPU scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

The ready queue is not necessarily a first-in , first-out(FIFO) queue. It can be implemented as a FIFO queue a priority queue. A tree, or simply an unordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU . The records in the queues are generally process control blocks(PCB) of the processes.

2.1.3 PREEMPTIVE SCHEDULING

CPU- scheduling decisions may take place under the following four circumstances:

1. when a process switches from the running state to the waiting state(e.g. as the result of an I/O request or an invocation of wait for the termination of one of the child processes)
2. when a process switches from the running state to the ready state (e.g. when an interrupt occurs)
3. when a process switches from the waiting state to the ready state(e.g. at completion of I/O)
4. when a process terminates

for situations 1 and 4, there is no choise in terms of scheduling. A new process(if one existing in the ready queue)must be selected for execution. There is a choise , however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4 , we say that the scheduling is non-preemptive or cooperative ; otherwise, it is preemptive. Under non-preemptive scheduling once the CPU has been allocated to a process, the process keeps the CPU until it release the CPU either by terminating or by switching to the waiting state.

Cooperative scheduling is the only method that can be used on certain hardware platforms, because it does not require the special hardware such as a timer needed for preemptive scheduling. Unfortunately, preemptive scheduling incurs a cost associated with access to shared data.

2.1.4 DISPATCHER

Another component involved in the CPU- scheduling function is the dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler; this involves:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency.

2.2 SCHEDULING CRITERIA

Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another, In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

- **CPU utilization**: We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system , it should range from 40 percent(for a lightly loaded system) to 90 percent (for a heavily used system).
- **Throughput:** if the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.

- **Turnaround time** : From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time :** The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that process spends waiting in the ready queue. Waiting time is the sum of the periods spent in the ready queue.
- **Response time:** In an interactive system , turnaround time may not be the best criterion. Often, a process can produce some output fairly early can continue computing new results while previous results are being output to the user. Thus another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

It is desirable to maximize CPU utilization and throughput and minimize turnaround time , waiting time, and response time . In most cases, we optimize the average measure . However, under some circumstances , it is desirable to optimize the minimum or maximum values rather than the average. For example, to guarantee that all users get good services, we may want to optimize the maximum response time.

Investigators have suggested that , for interactive systems (such as time-sharing systems), it is more important to minimize the variance in the response time than to minimize the average response time . A system with reasonable and predictable response time may be considered more desirable than system that is faster on the average but is highly variable. However, little work has been done on CPU-scheduling algorithms that minimize variance.

2.3 SCHEDULING ALGORITHMS

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU scheduling algorithms, In this section we describe several of them.

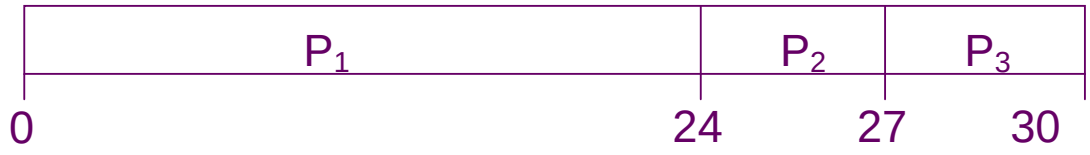
2.3.1 FIRST-COME , FIRST- SERVED SCHEDULING

By far the simplest CPU-scheduling algorithms is the first-come , first-served(FCFS) scheduling algorithm. With this scheme, the process that request the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with FIFO queue. When a process enters the ready queue , its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand.

The average waiting time under the FCFS policy, however, is often quite long. Consider the following set of processes that arrive at time 0, with length of the CPU burst given in milliseconds:

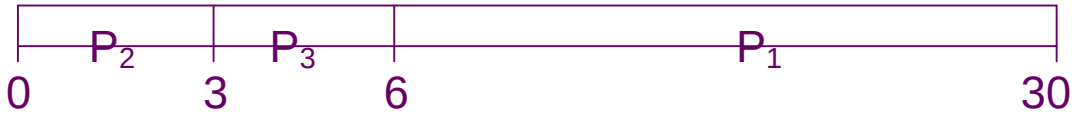
Process	Burst Time
P1	24
P2	3
P3	3

Suppose that the processes arrive in the order: *P1*, *P2*,*P3*.The Gantt Chart for the schedule is:



- Waiting time for P_1 = 0 ; p2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

➢ Suppose that the processes arrive in the order P_2, P_3, P_1. The Gantt chart for the schedule is:



- Waiting time for P_1 = 6 ; P_2 = 0; P_3 = 3
- Average waiting time: (6 + 0 + 3)/3 = 3

the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the process's CPU burst times vary greatly.
In addition, consider the performance of FCFS scheduling in a dynamic situation. Assume we have one CPU-bound process and many I/O-bound processes. As the processes flow around the system, the following scenario may result. The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/O devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. There is a convoy effect as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.
The FCFS scheduling algorithm is non-preemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

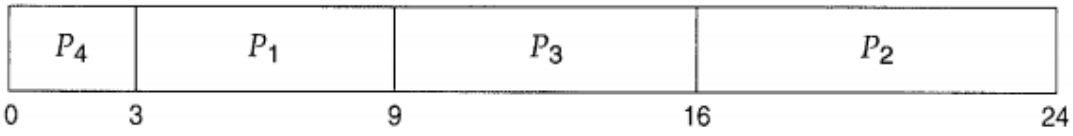
2.3.2 SHORTEST-JOB-FIRST SCHEDULING

A different approach to CPU scheduling is the shortest-job-first (SJF) scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same , FCFS scheduling is used to break the tie.

As an example of SJF scheduling , consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling , we would schedule these processes according to the following Gantt chart:



The waiting time is 3 milliseconds for process P1, 16 milliseconds for process *P2, 9* milliseconds for process *P3,* and 0 milliseconds for process *P4.* Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

The SJF scheduling algorithm is provably *optimal,* in that it gives the minimum average waiting time for a given set of processes. Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the *average* waiting time decreases.

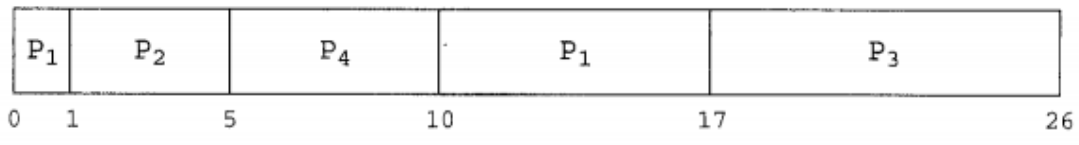
The SJF algorithm can be either preemptive or non-preemptive. The choise arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the

currently executing process. A preemptive SJF algorithm will preempt the currently execute process , whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst. Sometimes , non-preemptive SJF scheduling is also called **Shortest Process Next(SPN)** scheduling & preemptive SJF is called **Shortest remaining time(SRT)** scheduling.

As an example , consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

If the processes arrives at the ready queue at the times shown and need the identical burst times, then the resulting preemptive SJF scheduling is as depicted in the following Gantt chart:



Process *P1* is started at time 0, since it is the only process in the queue. Process *P2* arrives at time 1. The remaining time for process *P1* (7 milliseconds) is larger than the time required by process *P2* (4 milliseconds), so process *P1* is preempted, and process *P2* is scheduled. The average waiting time for this example is ((10 — 1) + (1 — 1) + (17 — 2) + (5 — 3))/4 = 26/4 6.5 milliseconds. Non-preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

2.3.3 PRIORITY SCHEDULING

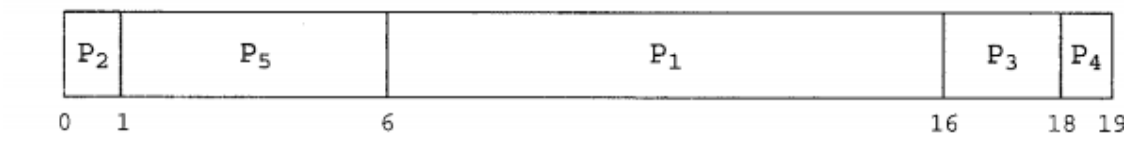
The SJF algorithm is a special case of the general priority scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority **(p**) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority and vice versa.

Note that we discuss scheduling in terms of *high* priority and *low* priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority

As an example, consider the following set of processes, assumed to have arrived at time 0, in the order *P1, l2, ..., P5.* with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political , factors.

Priority scheduling can be either preemptive or non-preemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is indefinite blocking, or starvation. A process that is ready to run but waiting for the **CPU** can be considered blocked. A

priority scheduling algorithm can leave some low- priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the **CPU.** Generally, one of two things will happen. Either the process will eventually be run, or the computer system will eventually crash and lose all unfinished low- priority processes.

A solution to the problem of indefinite blockage of low-priority processes is a **aging.** Aging is a technique of gradually increasing the priority of processes that wait in the system for along time.

2.3.4 ROUND-ROBIN SCHEDULING

The round-robin **(RR) scheduling algorithm** is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a time **quantum or** time slice, is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

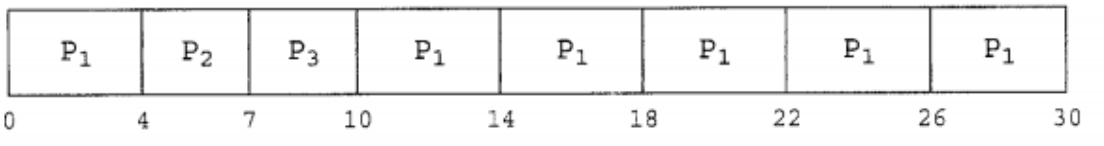
To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If we use a time quantum of 4 milliseconds, then process *Pi* gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the **CPU** is given to the next process in the queue, process *P2.* Since process *P2* does not need 4 milliseconds, it quits before its time quantum expires. The CPU is then given to the next process, process *P3.* Once each process has received 1 time quantum, the CPU is returned to process *P1* for an additional time quantum. The resulting RR schedule is



The average waiting time is 17/3 = 5.66 milliseconds

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row . If a process's CPU burst exceeds 1 time quantum, that process is *preempted* and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy. If the time quantum is extremely small (say, 1 millisecond), the RR approach is called processor sharing and creates the appearance that each of n processes has its own processor running at 1/n the speed of the real processor

In software, we need also to consider the effect of context switching on the performance of RR scheduling. Let us assume that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly.

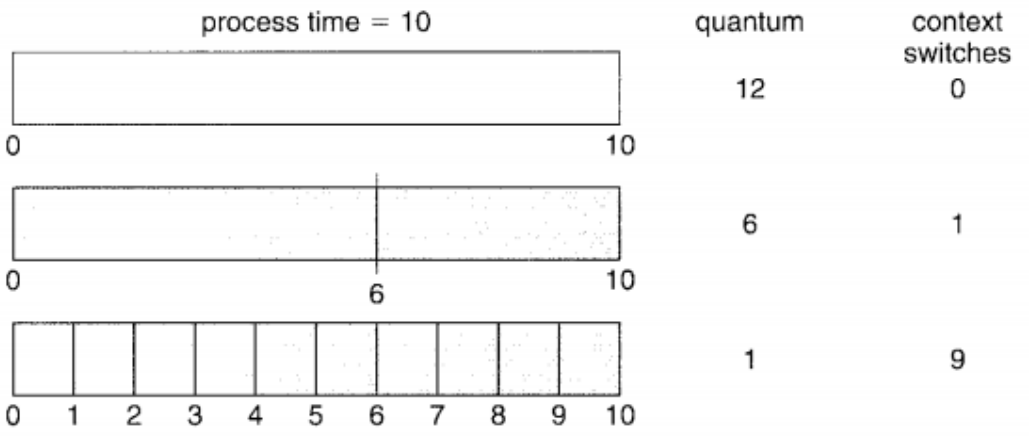


Figure 2.3 way in which a smaller time quantum increases context switches

CHAPTER THREE

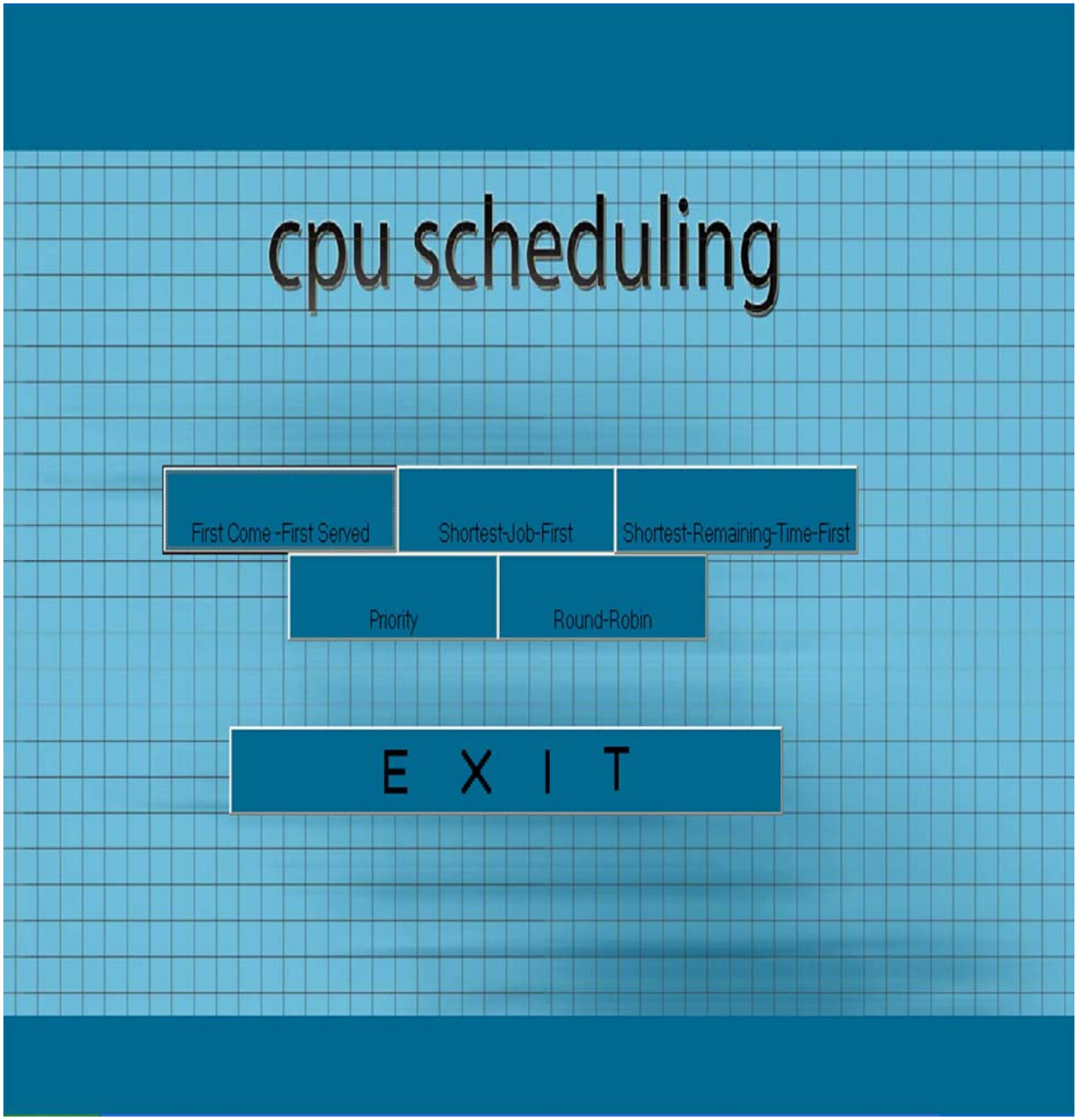


Figure 3.1 the basic screen which contain algorithms (FCFS, SJF, SRTF, RR, PRIORITY)

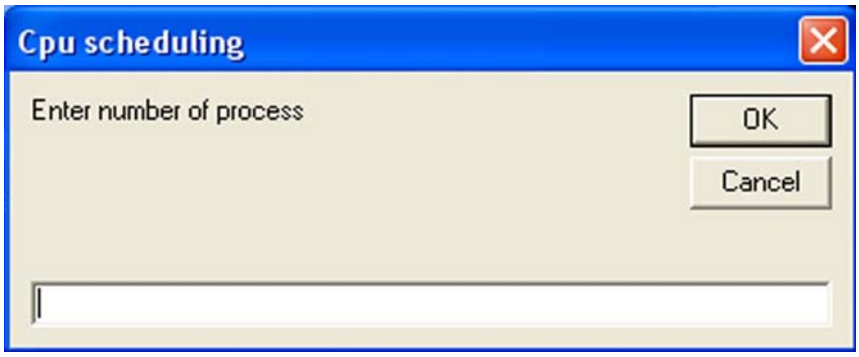


Figure 3.2 shows how enter number of processes

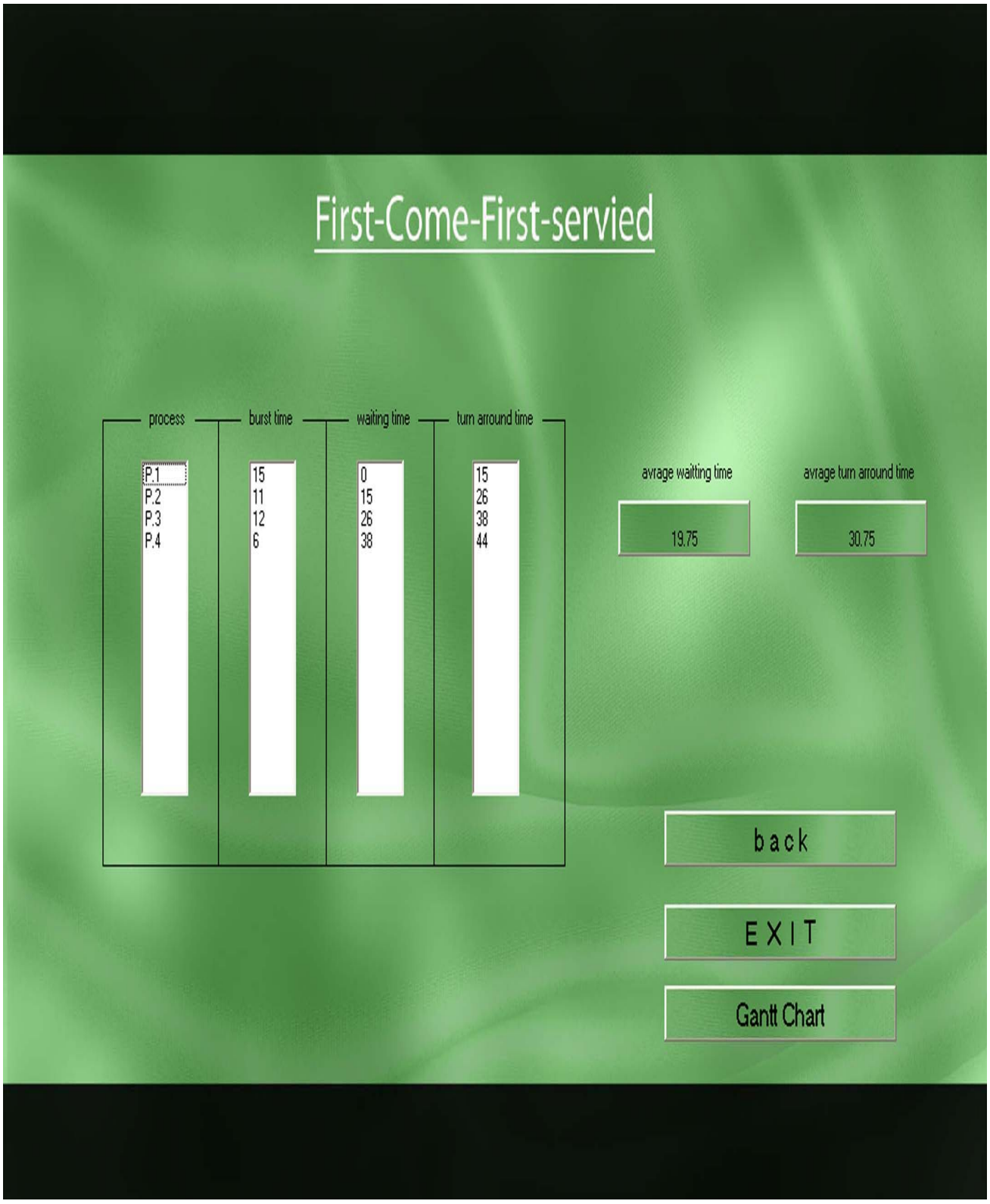


Figure 3.3 shows implementations of (First Come First Served) algorithm

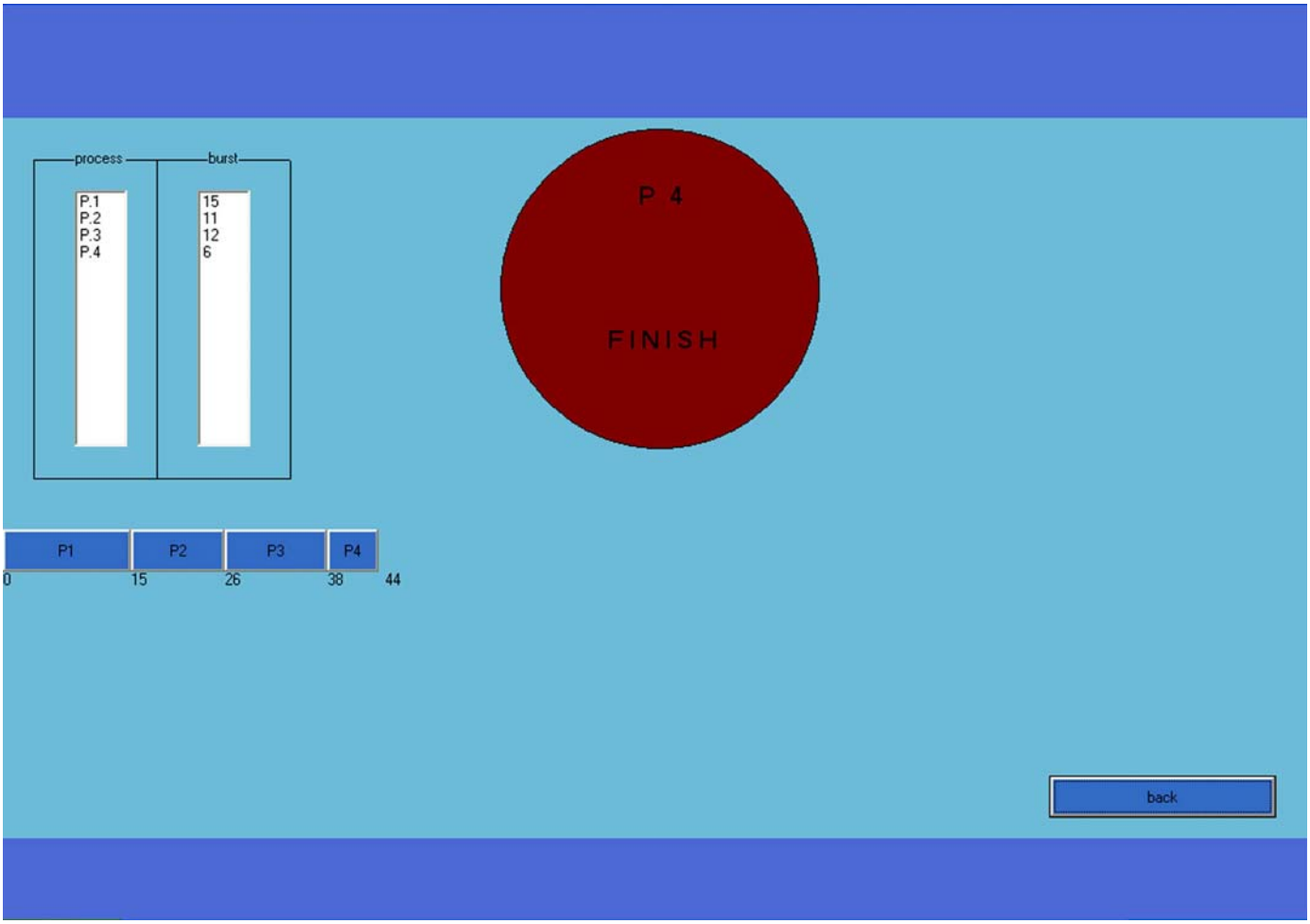


Figure 3.4 shows Gantt chart of (First Come First Served)

The code that implement the above algorithm (FCFS) is :

```
Private Sub Form_Activate()
ReDim p(c) As rec
For i = 1 To c
  With p(i)
.no = i
.burst = Int((20 * Rnd) + 1) ' Rand(1, 20)
End With
Next
```

```
For i = 1 To c
List1.AddItem ("P." & p(i).no)
List2.AddItem p(i).burst
Next
k1 = 0
k2 = 0
wtime = 0
For i = 1 To c
List3.AddItem wtime
turntime = p(i).burst + wtime
List4.AddItem turntime
wtime = wtime + p(i).burst
Next
'count the avrage waitting time
For i = 0 To List3.ListCount - 1
k1 = k1 + List3.List(i)
Next
avwtime = k1 / c
'count the avrage turn arround time
For i = 0 To List4.ListCount - 1
k2 = k2 + List4.List(i)
Next
avturntime = k2 / c
Command1.Caption = avwtime
Command2.Caption = avturntime
End Sub
```

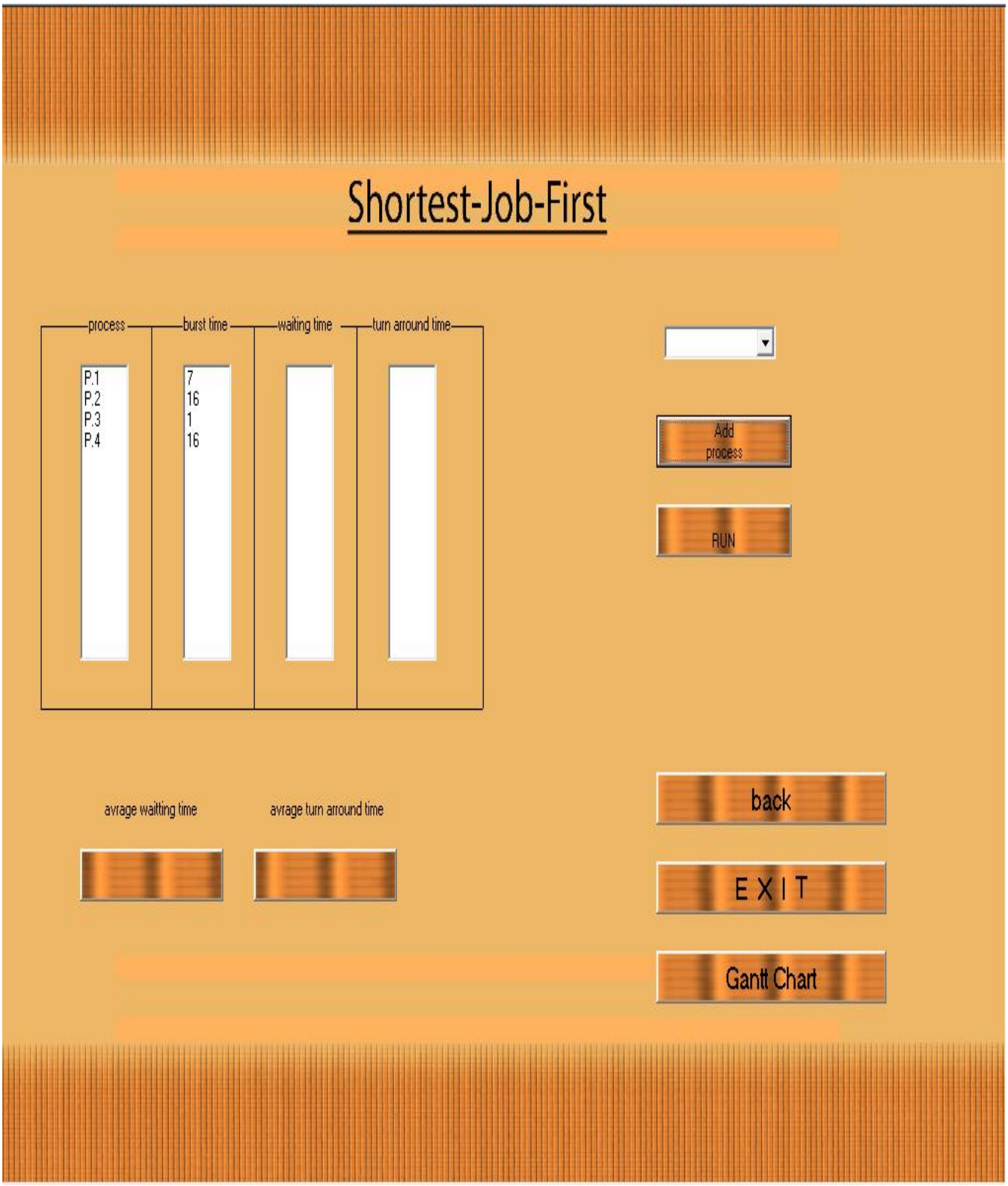


Figure 3.5 shows implementations of (Shortest Job First) algorithm before running

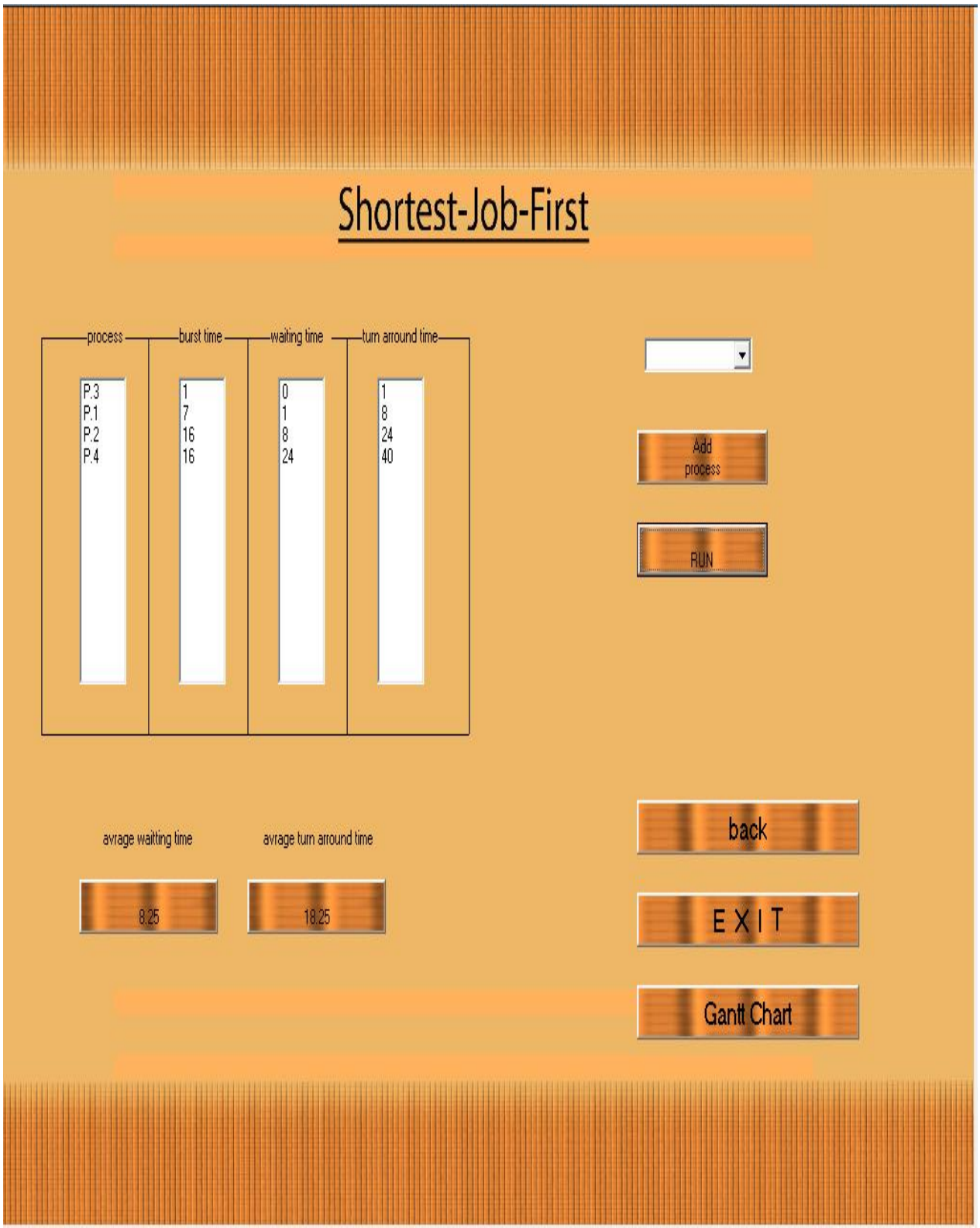


Figure 3.6 shows implementations of (Shortest Job First) algorithm after running

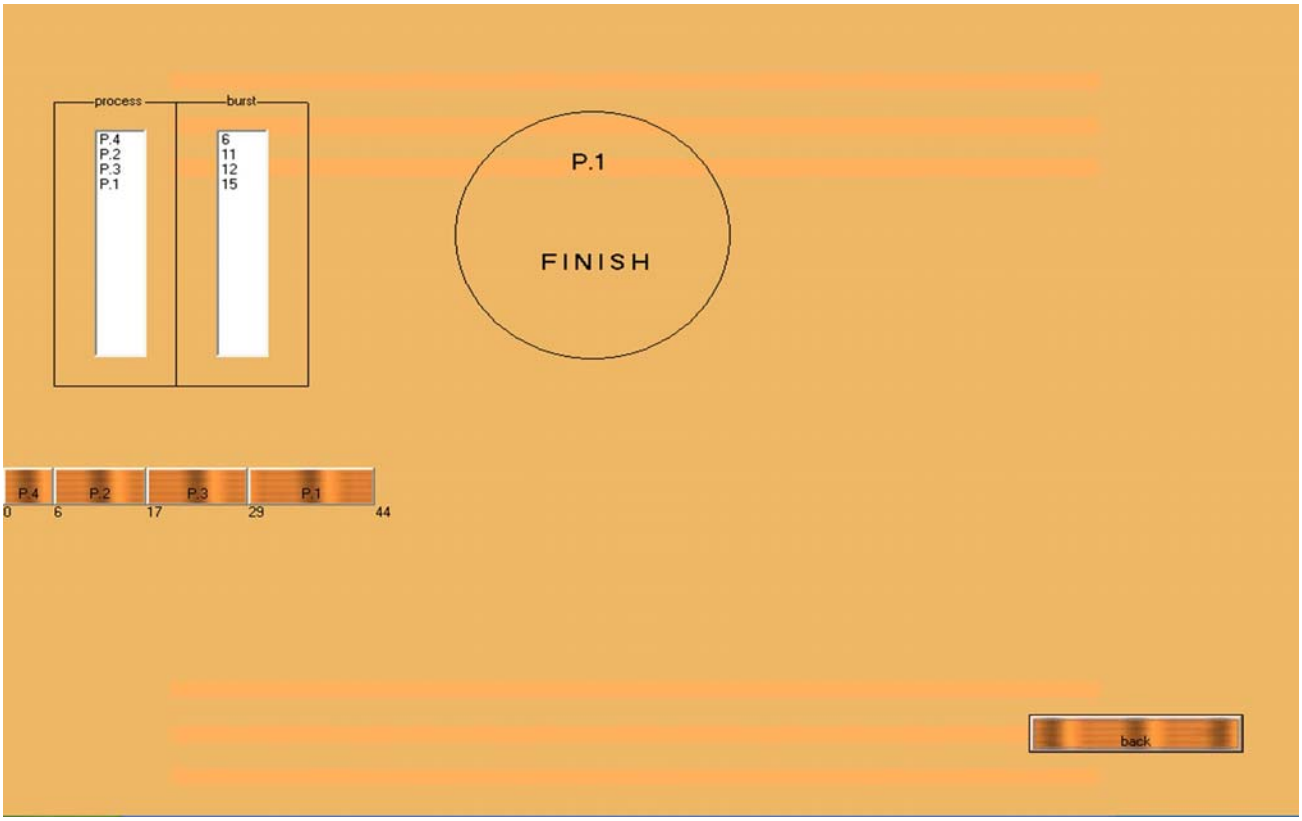


Figure 3.7 shows Gantt chart of (Shortest Job First) algorithm

The code that implement the above algorithm (SJF) is :

```
Private Sub Command2_Click()
For i = 1 To c
p(i).burst = List2.List(i - 1)
Next
For i = 0 To List2.ListCount - 1
 For j = i + 1 To List2.ListCount - 1
If (Val(List2.List(i)) > Val(List2.List(j))) Then
temp = List2.List(i)
List2.List(i) = List2.List(j)
List2.List(j) = temp
```

```
temp1 = List1.List(i)
List1.List(i) = List1.List(j)
List1.List(j) = temp1
End If
 Next
Next
k1 = 0
k2 = 0
wtime = 0
For i = 1 To c
List3.AddItem wtime
turntime = List2.List(i - 1) + wtime
List4.AddItem turntime
wtime = wtime + List2.List(i - 1)
Next
'count the avrage waitting time
For i = 0 To List3.ListCount - 1
k1 = k1 + List3.List(i)
Next
avwtime = k1 / c
'count the avrage turn arround time
For i = 0 To List4.ListCount - 1
k2 = k2 + List4.List(i)
Next
avturntime = k2 / c
    Command5.Caption = avwtime
Command6.Caption = avturntime
```

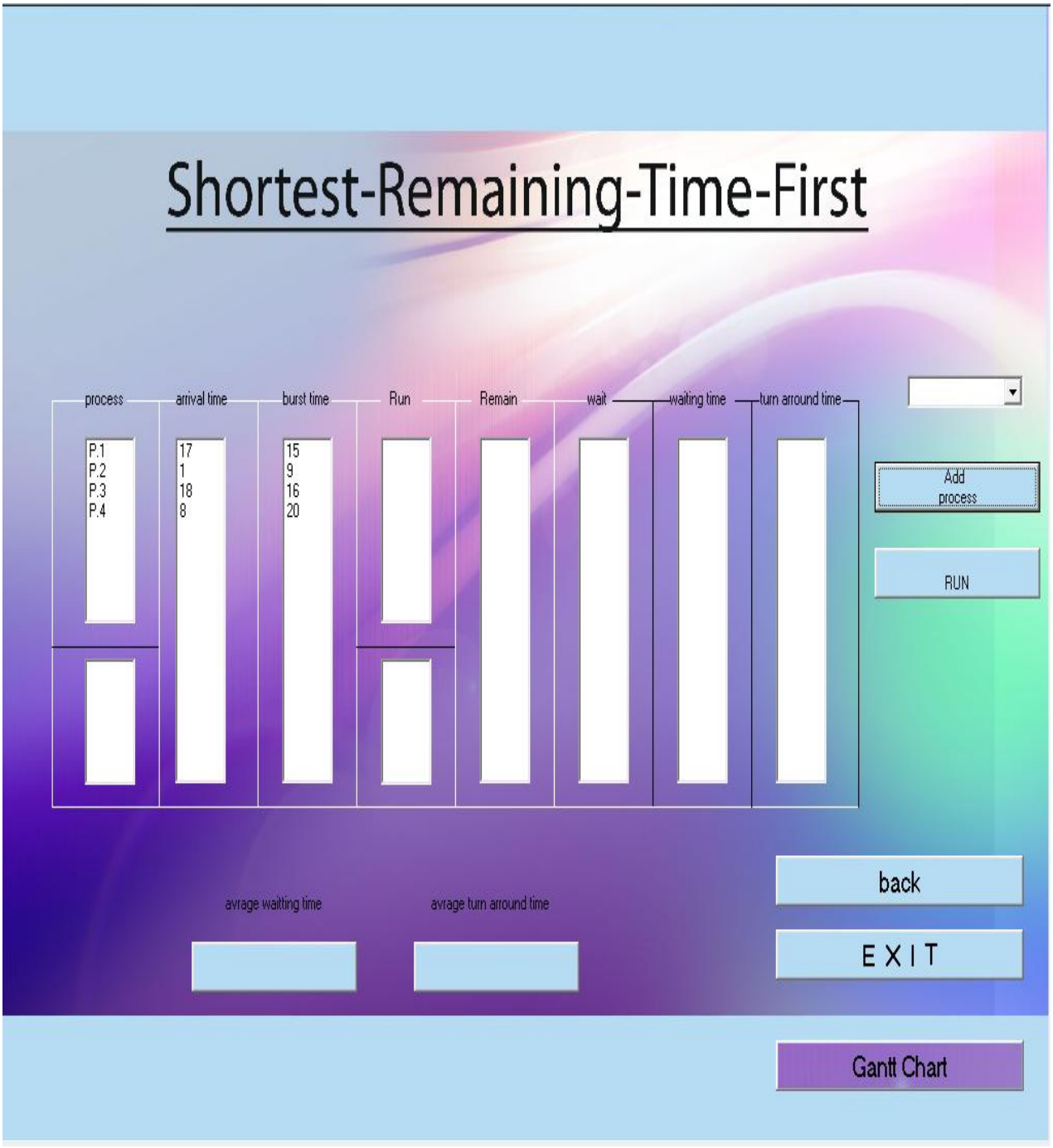


Figure 3.8 shows implementations of (Shortest Remaining Time First) algorithm before running

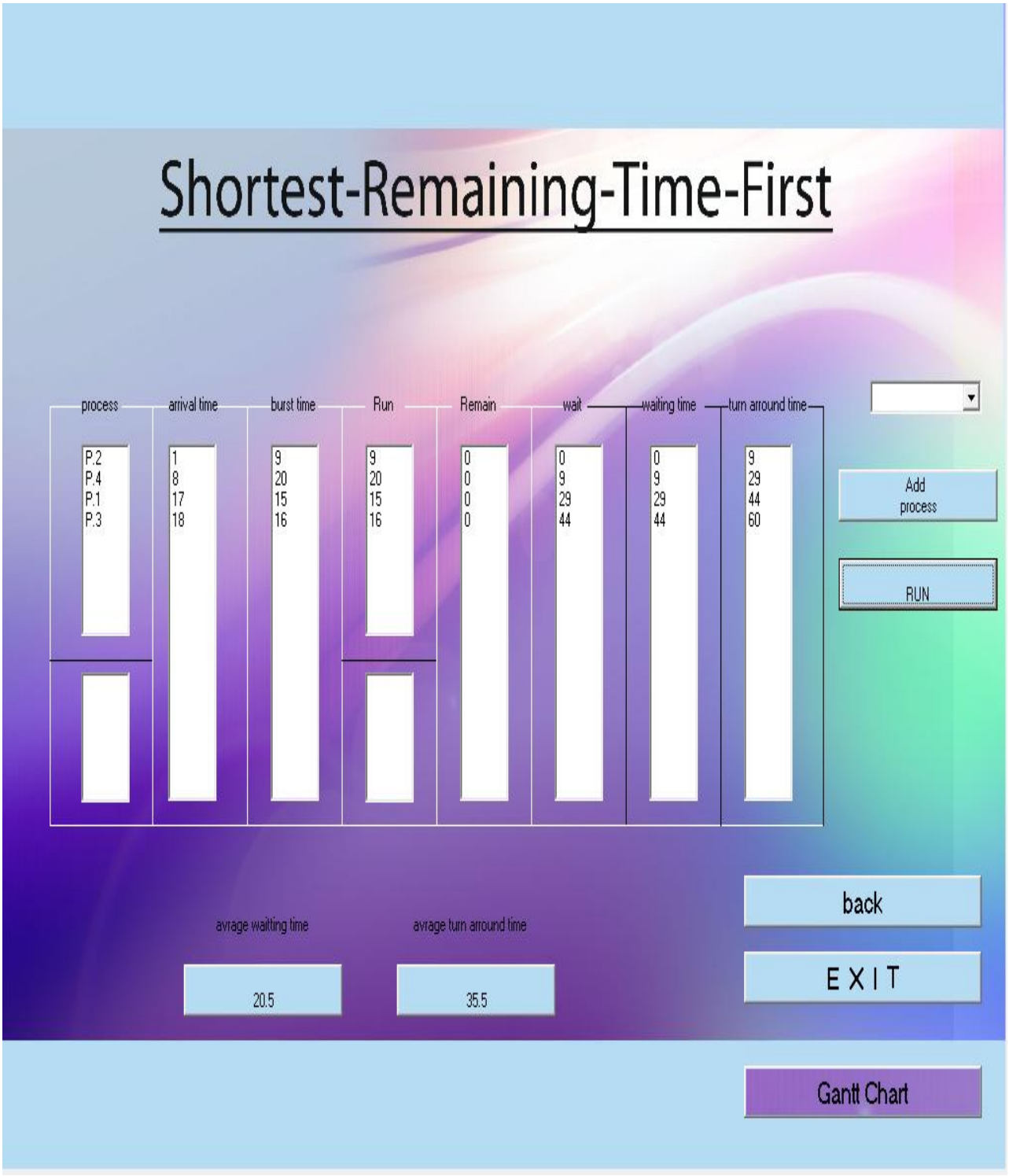


Figure 3.9 shows implementations of (Shortest Remaining Time First) algorithm after running

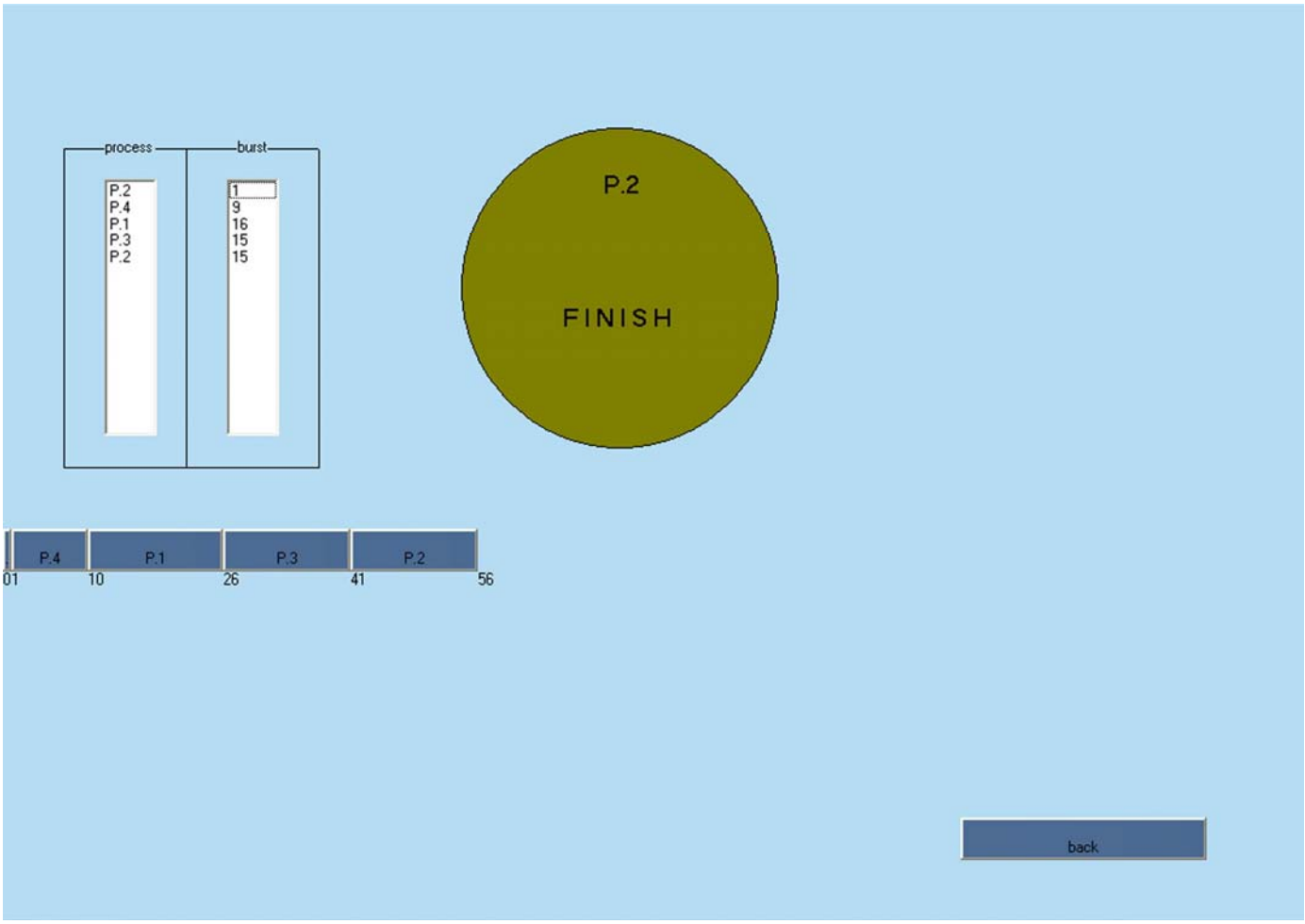


Figure 3.10 shows Gantt chart of (Shortest Remaining Time First) algorithm

The code that implement the above algorithm (SRTF) is :

```
Private Sub Command2_Click()
For i = 1 To c
p(i).arivl = List2.List(i - 1)
p(i).burst = List3.List(i - 1)
Next
For i = 0 To List2.ListCount - 1
 For j = i + 1 To List2.ListCount - 1
If (Val(List2.List(i)) = Val(List2.List(j))) Then
 List2.AddItem Val(List2.List(j)) + 1
```

```
 List2.RemoveItem (j)
ElseIf (Val(List2.List(i)) > Val(List2.List(j))) Then
temp1 = List1.List(i)
List1.List(i) = List1.List(j)
List1.List(j) = temp1
temp2 = List2.List(i)
List2.List(i) = List2.List(j)
List2.List(j) = temp2
temp3 = List3.List(i)
List3.List(i) = List3.List(j)
List3.List(j) = temp3
End If
 Next     Next
For i = 0 To List3.ListCount - 1
 If i = List3.ListCount - 1 Then
   List4.AddItem List3.List(List2.ListCount - 1)
   List5.AddItem 0
 Else
  run = Val(List2.List(i + 1)) - Val(List2.List(i))
  If Val(List3.List(i)) > run Then
    rm = Val(List3.List(i)) - run
        For j = i + 1 To List3.ListCount - 1
          If rm > Val(List3.List(j)) Then
            List4.AddItem run
            List5.AddItem rm
            Exit For
          Else
```

```
            List4.AddItem List3.List(i)
            List5.AddItem 0
            Exit For
          End If
        Next
  Else
     List4.AddItem List3.List(i)
     List5.AddItem 0
  End If
 End If
Next
     For i = 0 To List5.ListCount - 1
  If List5.List(i) <> 0 Then
    List6.AddItem List1.List(i)
    List7.AddItem List5.List(i)
  End If
Next
wt1 = 0
For i = 1 To c
  List8.AddItem wt1
  wt1 = wt1 + Val(List4.List(i - 1))
Next
For i = 0 To List7.ListCount - 1
  List8.AddItem wt1
  wt1 = wt1 + Val(List7.List(i - 1))
  Next
Label11.Caption = wt1
```

```
'count wait time for each process
X = 1
For i = 0 To List5.ListCount - 1
   If List5.List(i) <> 0 Then
      wt = Val(List8.List((List5.ListCount - 1) + X)) - Val(List8.List(i + 1)) + Val(List8.List(i))
      List9.AddItem wt
      X = X + 1
   Else
      List9.AddItem List8.List(i)
   End If
Next
'count turn arround time for each process
For i = 0 To List3.ListCount - 1
   tr = Val(List3.List(i)) + Val(List9.List(i))
   List10.AddItem tr
Next
'count avrage waitting time
'count avrage turn arround time
wait = 0
turn = 0
For i = 0 To c - 1
   wait = wait + List9.List(i)
   turn = turn + List10.List(i)
Navwtime = wait / c
avturntime = turn / c
Command5.Caption = avwtime
Command6.Caption = avturntime
```

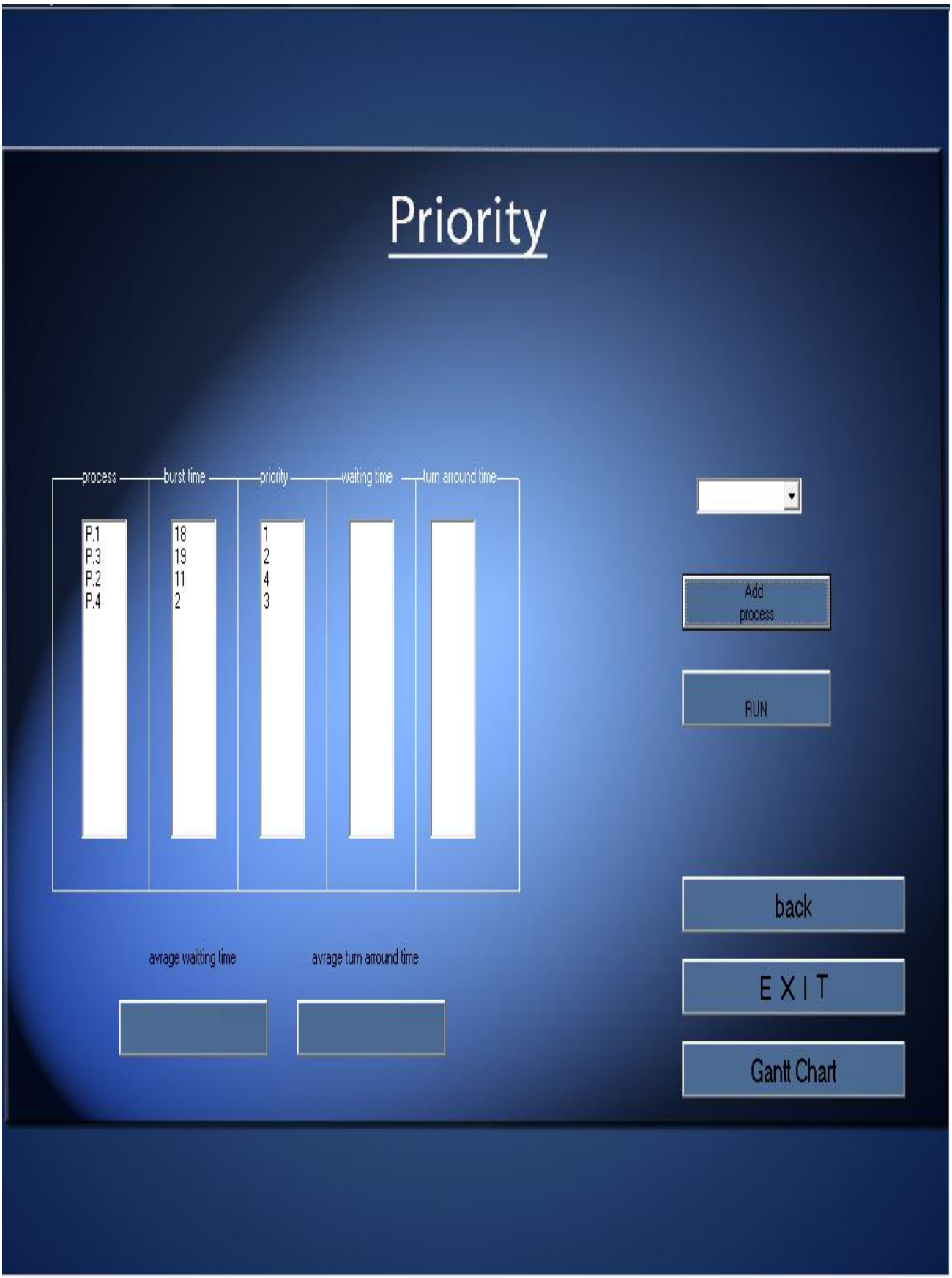


Figure 3.11 shows implementations of (Priority) algorithm before running

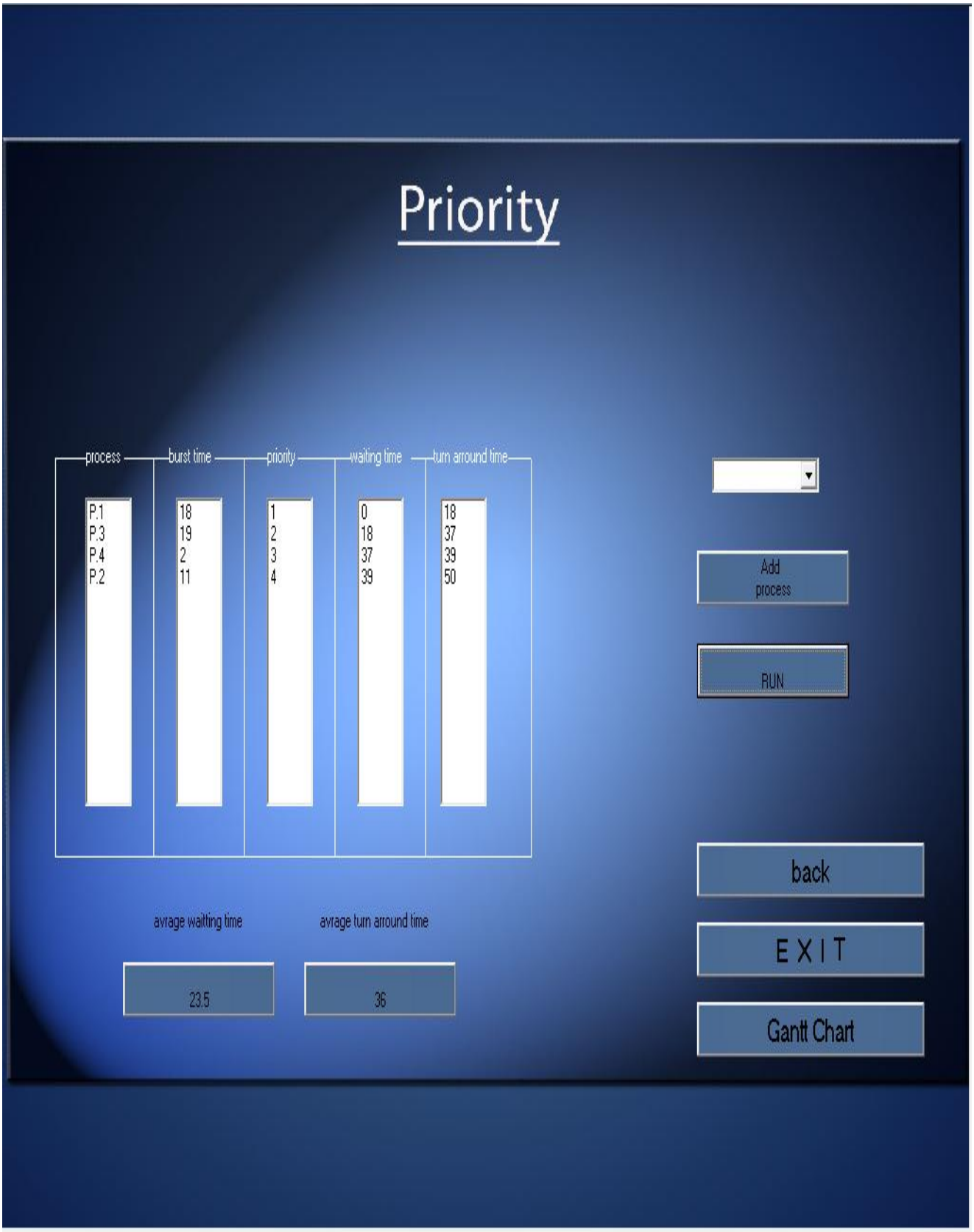


Figure 3.12 shows implementations of (Priority) algorithm after running

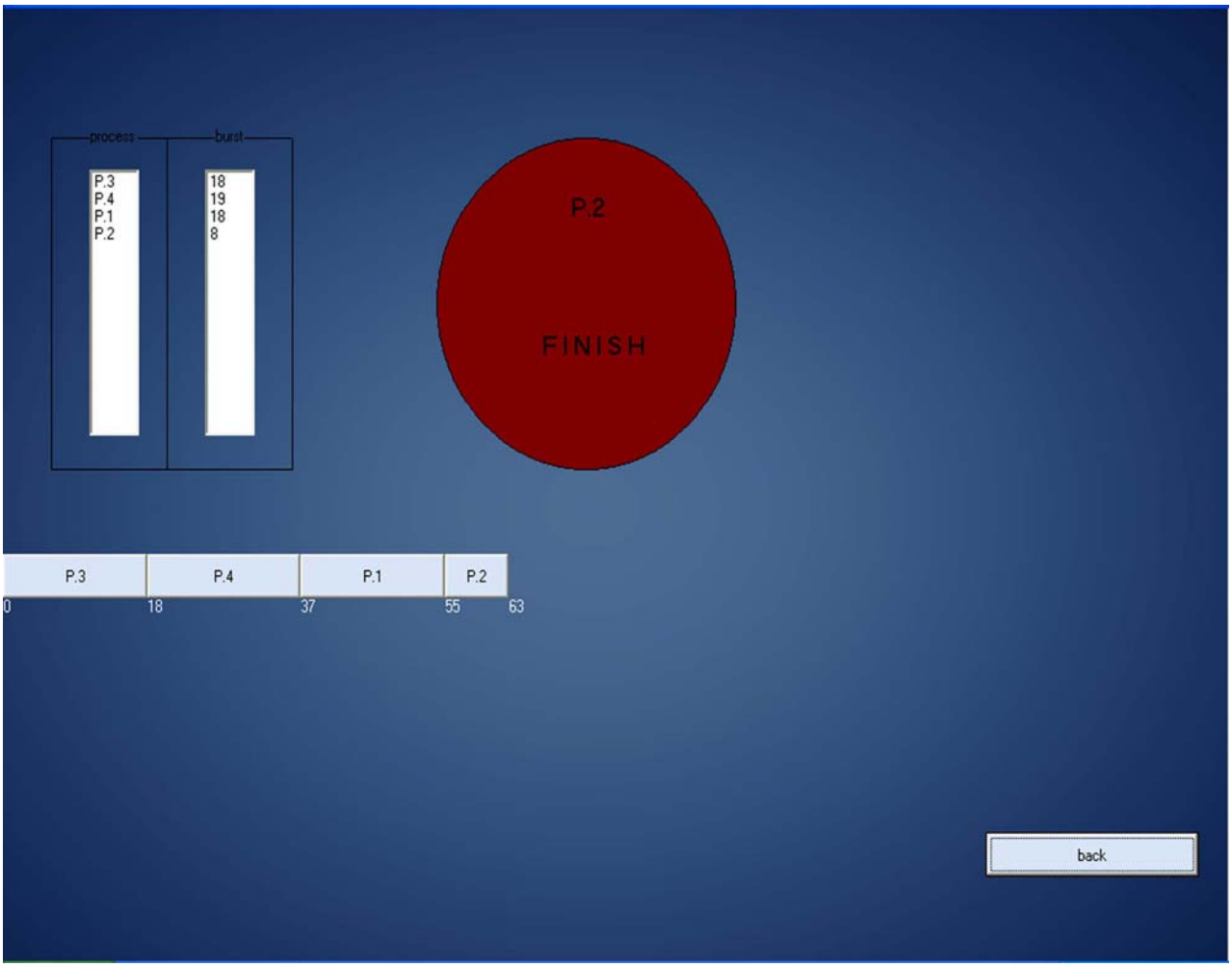


Figure 3.13 shows Gantt chart of (Priority) algorithm

The code that implement the above algorithm (Priority) is :

```
Private Sub Command2_Click()
For i = 1 To c
p(i).burst = List2.List(i - 1)
p(i).priority = List3.List(i - 1)
Next
For i = 0 To List3.ListCount - 1
 For j = i + 1 To List3.ListCount - 1
```

```
If List3.List(i) > List3.List(j) Then
temp1 = List2.List(i)
List2.List(i) = List2.List(j)
List2.List(j) = temp1
temp2 = List1.List(i)
List1.List(i) = List1.List(j)
List1.List(j) = temp2
temp3 = List3.List(i)
List3.List(i) = List3.List(j)
List3.List(j) = temp3
End If
 Next
Next
k1 = 0
k2 = 0
wtime = 0
For i = 1 To c
List4.AddItem wtime
turntime = List2.List(i - 1) + wtime
List5.AddItem turntime
wtime = wtime + List2.List(i - 1)
Next
'count the avrage waitting time
For i = 0 To List4.ListCount - 1
k1 = k1 + List4.List(i)
Next
avwtime = k1 / c
```

```
'count the avrage turn arround time
For i = 0 To List5.ListCount - 1
k2 = k2 + List5.List(i)
Next
avturntime = k2 / c
Command5.Caption = avwtime
Command6.Caption = avturntime

End Sub
```

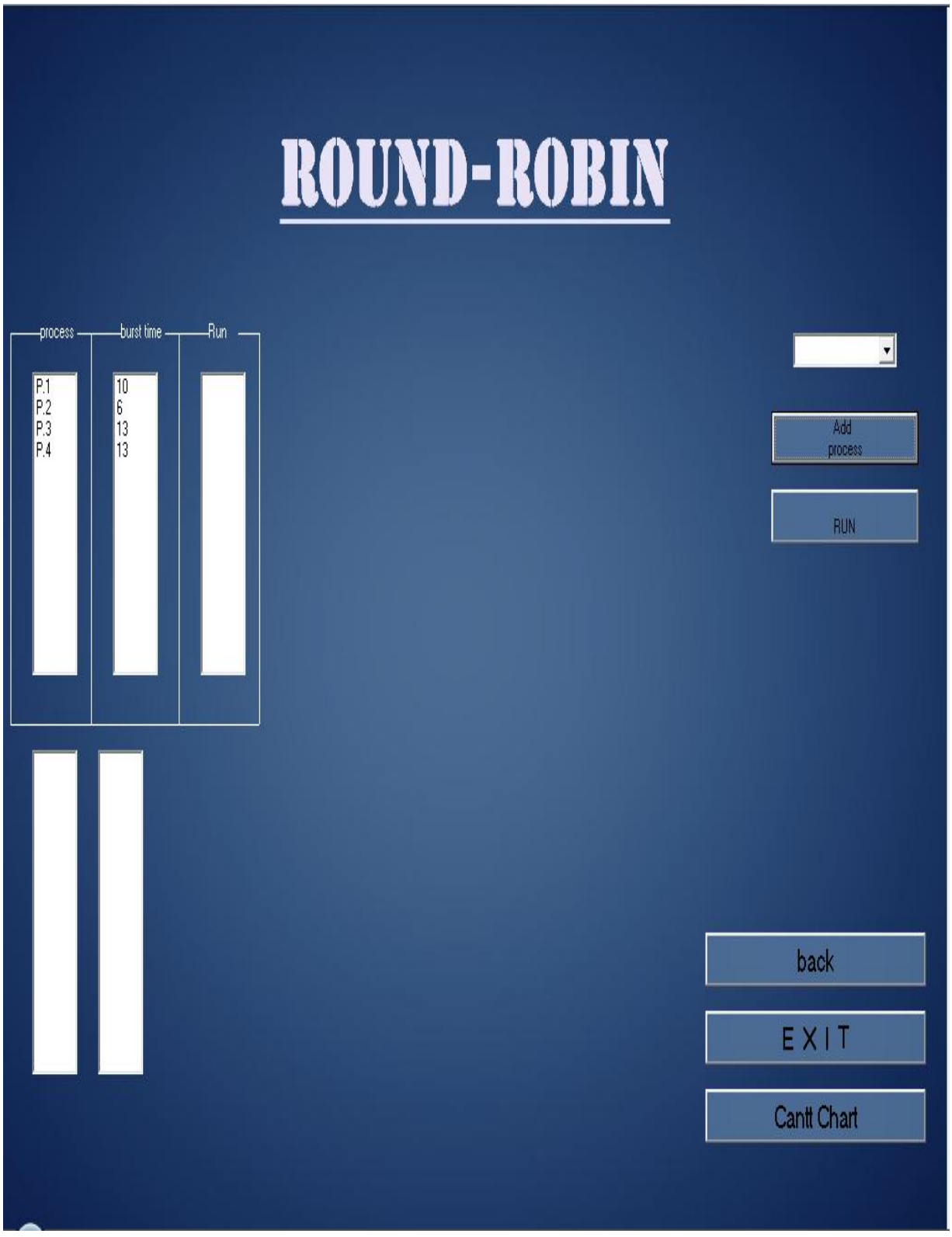


Figure 3.14 shows implementations of (Round-Robin) algorithm before running

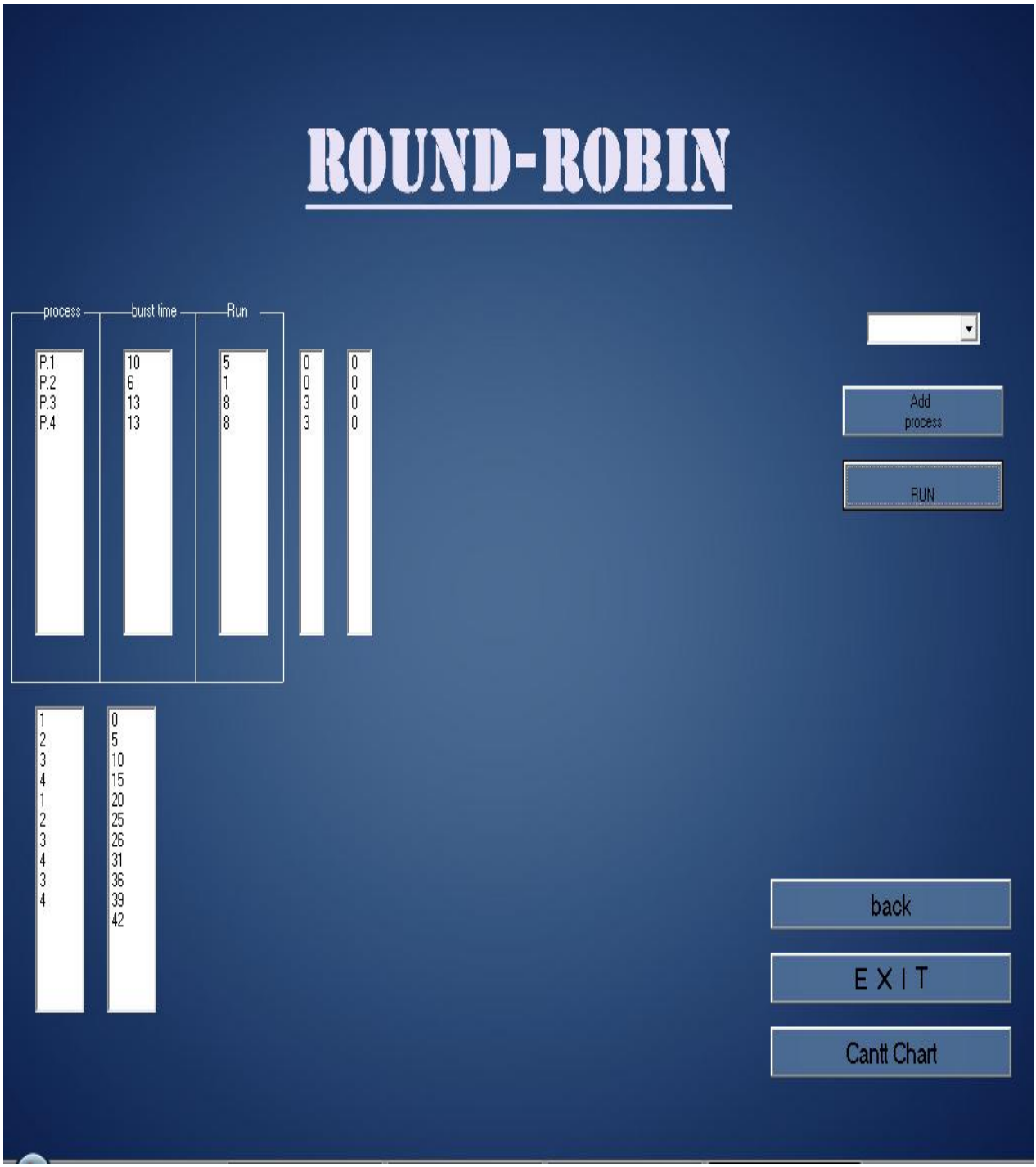


Figure 3.15 shows implementations of (Round-Robin) algorithm after running

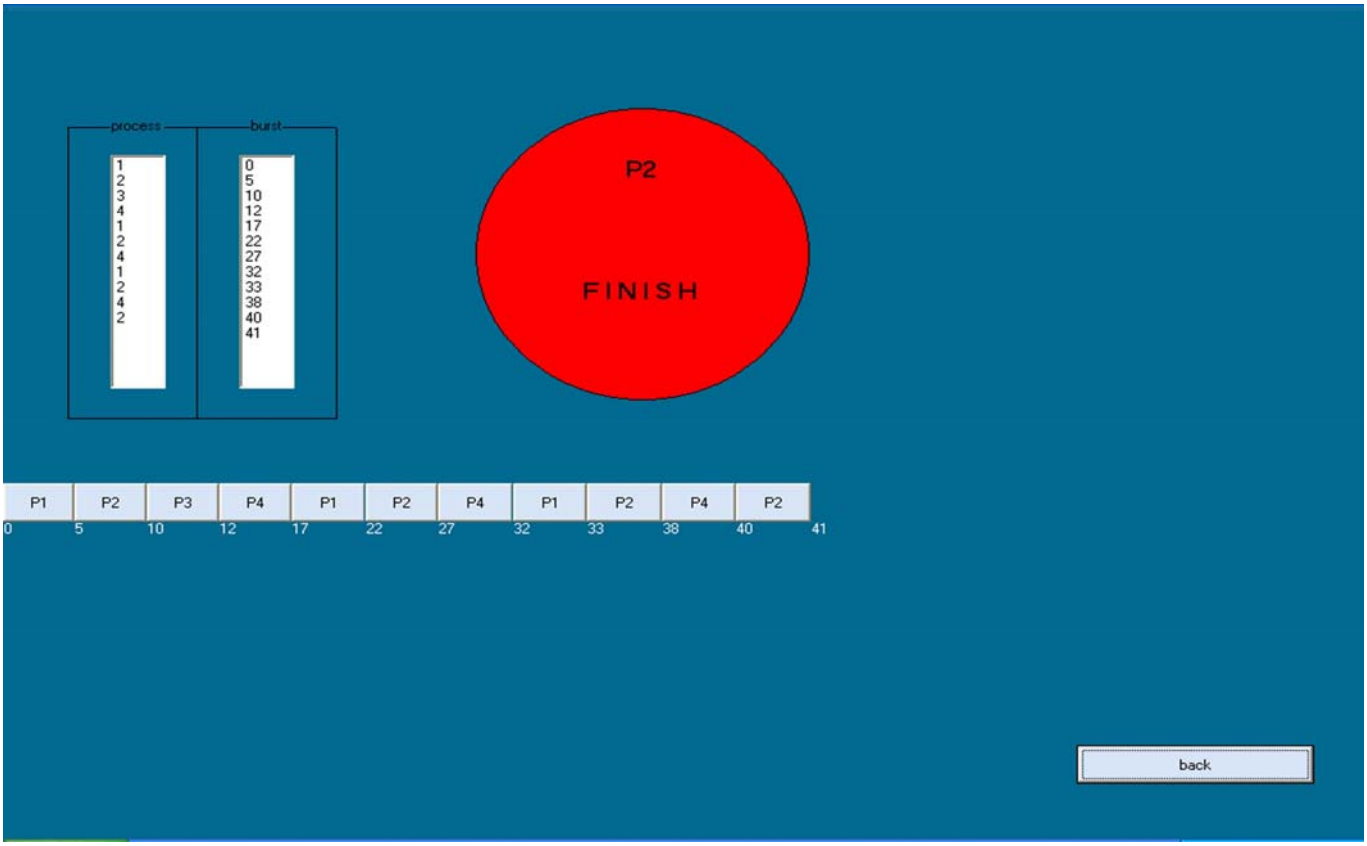


Figure 3.16 shows Gantt chart of (Round-Robin) algorithm

The code that implement the above algorithm (Priority) is :

```
Private Sub Command2_Click()
        For i = 0 To c - 1
          If ts > List2.List(i) Then
            List3(0).AddItem 0
          Else
            List3(0).AddItem List2.List(i) - ts
          End If
        Next
```

```
For j = 0 To 12
 sum = 0
For i = 0 To c - 1
   If (List3(j).List(i) = 0) Then
     List3(j + 1).Visible = True
     List3(j + 1).AddItem 0
   Else
     List3(j + 1).Visible = True
     If ts > List3(j).List(i) Then
       List3(j + 1).AddItem 0
     Else
       List3(j + 1).AddItem List3(j).List(i) - ts
     End If
   End If
  Next
   For k = 0 To c - 1
   sum = sum + List3(j).List(k)
   Next
   If sum = 0 Then
   List3(j + 1).Visible = False
   Exit For
  End If
   Next
 t = 0
For i = 0 To c - 1
 If List2.List(i) > ts Then
 List4.AddItem i + 1
```

```
List5.AddItem t
t = t + ts
Else
List4.AddItem i + 1
List5.AddItem t
t = t + List2.List(i)
End If
Next
For j = 0 To 12
If List3(j).Visible = True Then
For i = 0 To c - 1
If List3(j).List(i) <> 0 Then
If List3(j).List(i) > ts Then
List4.AddItem i + 1
List5.AddItem t
t = t + ts
Else
List4.AddItem i + 1
List5.AddItem t
t = t + List3(j).List(i)
End If
End If
Next
End If
Next
List5.AddItem t
End sub
```

Chapter four

4.1 Results

After build our project we experiment the algorithms to verify that results we obtain is exactly as the result we obtain from our theoretical study , and we found the same result so we can say our objectives has been made, but we have problems in some algorithms (shortest remaining time first, round robin) to save the temporary value of burst time for processes that preempted and to retrieve these values later.

In this project, we tried to implement all the ideas and expectations to bring this project into reality. Although, we were limited by time period and our abilities to perform these ideas and make it real. We conclude the following :

1-This idea can be implemented and applicable .

2-It can be developed and improved.

3-We faced big difficulty in dealing with the timing of the computer and link it to an active data base.

4- The preemptive algorithms are not flexible and required a special programming techniques to deal with it.

5-Although Visual basic is flexible but it is not powerful to deal with cases like this.

4.2 Future Work And Recommendation

As we end this stage of the project, we would like to recommend other researches in this field or related to :

1- Add visual effects to the program which will get more attention for the user when implementing an algorithm.

2- Add sound effects to the program to make it more attractive.

3- Use a more powerful programming like visual C++ or VB .NET , which has more modules and facilities.

4- Use other CPU scheduling algorithms like multilevel queue and others and add it to this program.

References

[1] Abraham Silberschatz , Peter Baer Galvin , Greg Gagine , "Operating System Concepts", 7th edition 153-166

[2] www.wikipedia.com

[3] www.howstuffwok.com

[4] Leo J. Cohen , " Operating System", 5th edition 309- 373

[5] Michael Kifer , Scott A. Smolka , "Introduction To Operating System Design And Implementation" 3rd edition 54-72

[6] M. Naghibzadeh " Concepts And Techniques" 101- 134

[7] Sudhir Kumar "Encyclopedia Of Operating System" 160- 205