UNIVERSITE PROTESTANTE AU CONGO

FACULTE DE SCIENCES INFORMATIQUES

Département de systèmes informatiques

BP : 4745 KINSHASA/LINGWALA

RAPPORT DU TP 5

PROJET DE PREMIERE ANNEE DE LICENCE

Présenté par :

Lionel PENENGE TEBANDIME

Superviseur: Prof.Dr MUKALA Patrick

ANNEE ACADEMIQUE: 2021-2022

RESOLUTION DU DEVOIR 5

Nous allons utiliser le projet 'ScheduleCPUTest', qui simule le comportement d'un simple planificateur de CPU. Dans ce projet, nous avons les classes suivantes :

Classe CPU

Cette classe représente un processeur physique.

```
using System;
using System.Timers;

/// <summary>
/// This class represents the hardware CPU
/// </summary>
public class CPU : ITimeTickReceiver
{
    /// <summary>
    /// The Process currently running on the CPU
    /// </summary>
    private IProcess current = null;

    /// <summary>
    /// The Scheduler
    /// </summary>
    private Scheduler scheduler;

        /// <summary>
    /// Make the scheduler known to the CPU
    /// </summary>
        /// <param name="scheduler"> The Scheduler</param>
        public Scheduler Scheduler
        {
                set
                {
                        this.scheduler = value;
                }
        }

    /// <summary>
    /// Check whether the CPU is currently busy
    /// </summary>
    public bool Busy
    {
       get
       {
          return current != null;
       }
    }
```

```
    /// <summary>
    /// Assigns a new TestProcess to run on the CPU<
    /// /summary>
    /// <param name="Process">The Process to be assigned to the CPU</param>
    public void SetProcess(IProcess Process)
        {
            this.current = Process;
        }

        /// <summary>
    /// Removes the current process from the CPU
    /// </summary>
        /// <returns> The removed Process
        /// </returns>
        public IProcess RemoveProcess()
        {
            IProcess tmp = this.current;
            this.current = null;
            return tmp;
        }

    /// <summary>
    /// This methode will be called by the HardwareTimer after each timer tick
    /// </summary>
    /// <param name="source"></param>
    /// <param name="e"></param>
    public void ReceiveTimeTick(object source, ElapsedEventArgs e)
        {
            if (this.current != null)
            {
                this.current.ReceiveTimeTick(source,e);
                // warn scheduler if process has finished
                if (this.current.Ready)
                {
                    this.scheduler.schedulingNeeded();
                }
            }
        }
}
```

Classe Planificateur

Cette classe planifie les processus sur le CPU.

Classe CircularProcesList

Cette classe implémente une liste circulaire. Elle est utilisée par l'ordonnanceur comme une file d'attente pour contenir les processus.

```
using System.Collections.Generic;

/// <summary>
/// This class implements a circular list using the LinkedList class
/// Note that elements in a LinkedList with n elements are numbered 0 .. (n-1)
/// </version>

public class CircularProcesList
{
   private LinkedList<IProcess> list;

   /// <summary>
   /// Creates a new LinkedList to contain Testproces-ses
   /// </summary>
   public CircularProcesList()
   {
      this.list = new LinkedList<IProcess>();
   }

   /// <summary>
   /// Check whether the queue is empty
   /// </summary>
   public bool Empty
   {
      get
      {
         return list.Count == 0;
      }
   }

   /// <summary>
   /// Retrieves the next TestProces from the list
   /// </summary>
   public IProcess Next
   {
      get
      {
         LinkedListNode<IProcess> first = this.list.First;
         if (first != null)
         {
            IProcess nextElement = first.Value;
            this.list.RemoveFirst();
            return nextElement;
         }
         else
         {
            return null;
         }
      }
```

```
    }

    /// <summary>
    /// Adds a TestProces to the list
    /// </summary>
    /// <param name="t"> The Testproces to be added</param>
    public void AddItem(IProcess t)
    {
        list.AddLast(t);
    }

    /// <summary>
    /// Deletes a TestProces from the list
    /// </summary>
    /// <param name="t"> The TestProces to be deleted</param>
    public void deleteItem(IProcess t)
    {
        list.Remove(t);
    }
}
```

Classe ITimeTickReceiver

Interface pour les classes qui veulent recevoir les tics de minuterie de l'application

```
using System;
using System.Timers;

/// <summary>
/// Interface to be implemented by receivers of ticks of the HardwareTimer
/// </summary>
public interface ITimeTickReceiver
{
        /// <summary>
    /// This method is called when a timertick occurs
    /// </summary>
        /// <param name="clock">number of current timertick</param>
    void ReceiveTimeTick(object source, ElapsedEventArgs e);
}
```

Classe HardWareTimer.

```
using System;
using System.Timers;

public class HardwareTimer : Timer
{
        /// <summary>
        /// Counting the number of millisecs
```

```
        /// </summary>
public long Clock = 0;

/// <summary>
/// The length of the timer interval in milliseconds
/// </summary>
public const long TickLength = 100;

        public HardwareTimer()
        {
                this.Interval = TickLength;
        }

        /// <summary>
/// Adds an ITimeTickReceiver to the list
/// </summary>
        /// <param name="receiver">The ITimeTickReceiver to be added</param>
        public void AddTickReceiver(ITimeTickReceiver receiver)
        {
                if (receiver != null)
                {
        this.Elapsed += receiver.ReceiveTimeTick;
                }
        }

/// <summary>
/// Starts the timer
/// </summary>
public void StartTimer()
{
   this.Elapsed += this.IncreaseClock;
   this.Enabled = true;
}

/// <summary>
/// Stops the timer
/// </summary>
public void StopTimer()
{
   this.Enabled = false;
}

/// <summary>
/// Increase the clock according to the timer interval
/// </summary>
/// <param name="source"></param>
/// <param name="e"></param>
private void IncreaseClock(object source, ElapsedEventArgs e)
{
   Clock+=TickLength;
```

```
    }
}
```

Cette classe génère des ticks de minuterie à intervalles réguliers, et fournit le temps écoulé. Les classes qui veulent utiliser ces ticks (dans notre cas CPU en Scheduler) doivent implémenter l'interface ITimeTickReceiver.

Classe IProces

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Timers;

public interface IProcess : ITimeTickReceiver
{
    string Name { get; set; }
    bool Ready { get; }
    long TurnAroundTime { get; }
    long WaitingTime { get; }
    long InitialCPUTimeNeeded { get; }
    long StartTime { set; }
}
```

L'interface qui doit être implémentée par les processus qui s'exécutent sur le simulateur. TestProces

La classe pour les processus qui s'exécutent sur le CPU simulé. Cette classe implémente l'interface IProcess.

Classe SchedulerMain

```
using System;
using System.Threading;

/// <summary>
/// This program demonstrates how the scheduler operates.
/// This creates the CPU, scheduler and timer, and then the three example processes.
/// </summary>
public class SchedulerMain
{
    public static void Main(System.String[] args)
    {
        // Create virtual computer
        CPU singleCPU = new CPU();
        Scheduler CPUScheduler = new Scheduler(singleCPU);
        HardwareTimer timer = new HardwareTimer();
        // Add hardware components to hardware timer
```

```
        timer.AddTickReceiver(singleCPU);
        timer.AddTickReceiver(CPUScheduler);


        // Create and add processes

        IProcess tp1 = new TestProcess("process 1", 10000, 10000, 10000, 10000);
        // Add process 1 to the system
        CPUScheduler.AddProcess(tp1);

        IProcess tp2 = new TestProcess("process 2", 8000, 8000, 8000, 8000);
        // Add process 2 to the system
        CPUScheduler.AddProcess(tp2);

        IProcess tp3 = new TestProcess("process 3", 2000, 8000, 8000, 8000);
        // Add process 3 to the system
        CPUScheduler.AddProcess(tp3);

        // Start hardware timer
        timer.StartTimer();
        while (!CPUScheduler.NoProcesses)
            Thread.Sleep(100);
        Console.WriteLine("All processes finished");

        timer.StopTimer();
        // TODO print information after all processes are finished
        Console.ReadLine();
    }
}
```

Cette classe contient la méthode principale pour instancier les classes ci-dessus et tester l'ordonnanceur.

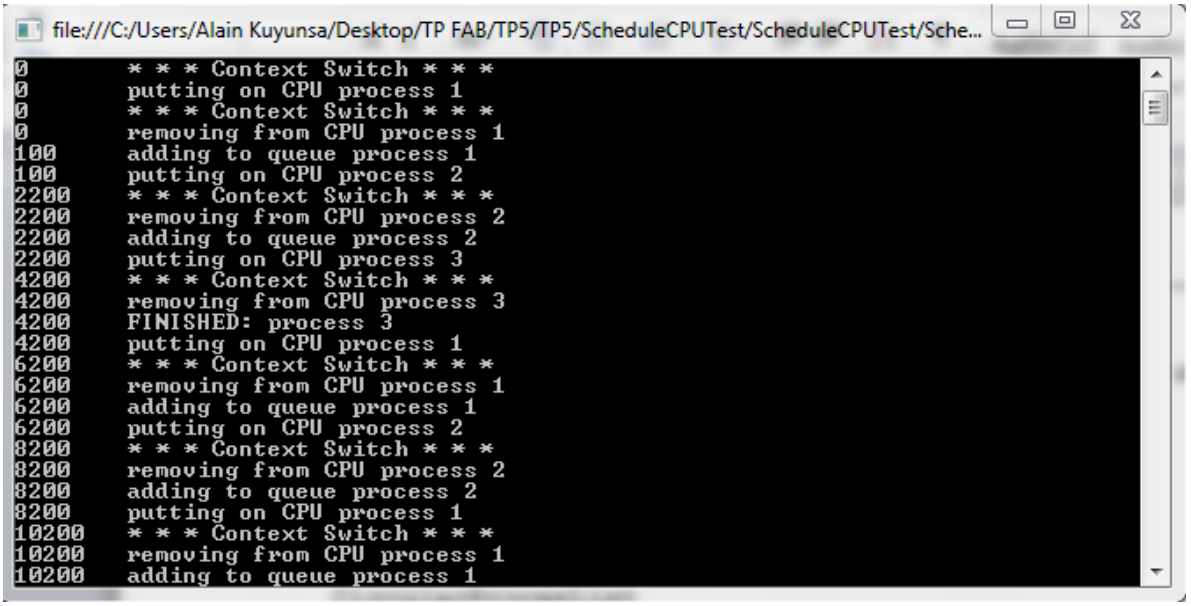
Dans le projet donné, un simple planificateur de CPU est implémenté, mais nous voulons l'étendre afin d'effectuer quelques mesures.

Effectuez les tâches suivantes (vous pouvez consulter la liste des tâches dans Visual Studio pour voir rapidement où vous devez apporter des modifications).

a. Étudiez le code donné.

Déterminez quel algorithme d'ordonnancement est mis en œuvre.

Solution



Code modifié

```
ImportsSystem.Timers;

/// <summary>
/// The scheduler
/// </summary>
public class Scheduler : ITimeTickReceiver
   private  currentCPU As CPU
   private queue As CPU CircularProcesList
   private timeSlice As long
   private As timeSliceCounter long
   private const DEFAULT_TIME_SLICE As integer = 2000
   private clock As long =0
   public Scheduler(CPU cpu)

      this.currentCPU = cpu
      timeSlice = DEFAULT_TIME_SLICE
      this.timeSliceCounter = 0
      queue = new CircularProcesList()
      cpu.Scheduler = this


   public Scheduler(CPU cpu, int quantum)
      : this(cpu)

      timeSlice = quantum


   /// <summary>
   /// The average turnaround time of all processes that finished up to now
   /// </summary>
   public NoProcesses As bool
```

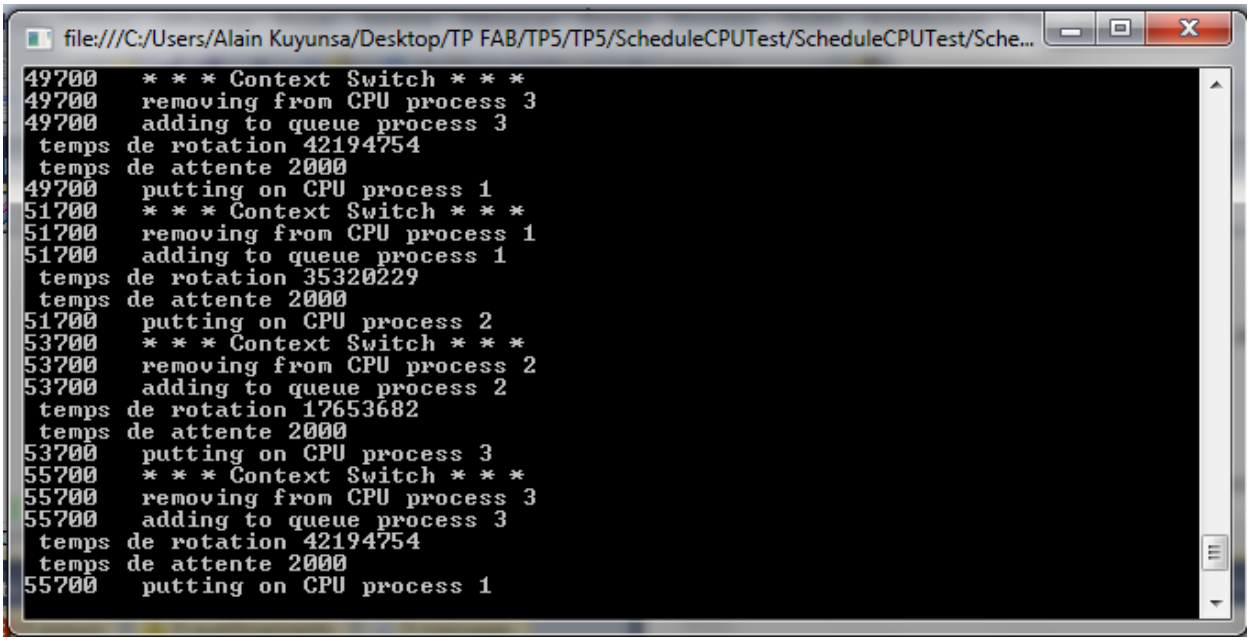
```
    get

        lock (this)

            return queue.Empty && !currentCPU.Busy


/// <summary>
/// Introduces a new TestProcess that must be exeeceuted on the CPU
/// </summary>
/// <param name="t">The new TestProcess</param>
public sub AddProcess(IProcess t)

    queue.AddItem(t)
    t.StartTime = clock
End Sub

/// <summary>
/// Signal to the scheduler that scheduling is needed in the next timertick
/// </summary>
public Sub schedulingNeeded()

    this.timeSliceCounter = 0
End Sub

/// <summary>
/// This method is called when a timertick occurs
/// Receive ticks until timequantum has finished, then schedule new proces
/// </summary>
/// <param name="source"></param>
/// <param name="e"></param>
public Sub ReceiveTimeTick(object source, ElapsedEventArgs e)

    this.clock = ((HardwareTimer)source).Clock
    this.timeSliceCounter--
    if (this.timeSliceCounter <= 0)

        // Time slice has finished, therefor reschedule
        this.schedule();
        this.timeSliceCounter = this.timeSlice / 100


/// <summary>
/// The actual scheduling operation
/// </summary>
public Sub schedule()

    lock (this)
```

```
        System.Console.Out.WriteLine(clock + "\t* * * Context Switch * * * ")
        IProcess current

        // remove process from CPU and put in queue
        IProcess removedProcess = this.currentCPU.RemoveProcess()
        if (removedProcess != null)
            System.Console.Out.WriteLine(clock + "\tremoving from CPU " +
removedProcess.Name)

            // add to queue if not ready
            if (!removedProcess.Ready)

                System.Console.Out.WriteLine(clock + "\tadding to queue " +
removedProcess.Name);
                this.queue.AddItem(removedProcess)

                System.Console.Out.WriteLine(clock + "\tFINISHED: " + removedProcess.Name)

        // select new process for CPU
        current = queue.Next

        // end of scheduling algorithm

        // start the selected process on the CPU (if any)
        if (current != null)

            System.Console.Out.WriteLine(clock + "\tputting on CPU " + current.Name)
            this.currentCPU.SetProcess(current)

        // no current processes
        else
            System.Console.Out.WriteLine(clock + "\tqueue empty")
```

Ici on a utilisé l'algorithme LIFO.

b. Développez le code de TestProces, de sorte que chaque processus garde la trace de son temps de rotation (propriété TurnAroundTime), de son temps d'attente (propriété WaitingTime) et du temps processeur dont le processus a besoin au total (propriété InitialCPUTimeNeeded).

Solution



Code developpé

```
Imports System.Timers
Imports System

/// <summary>
/// A process that will be executed by the CPU
/// </summary>
public class TestProcess : IProcess
   private name As string
   private ready As bool = false
   private cpuTimeNeeded As long  // Amount of CPU time needed to finish execution
   private clock As long // The current time in this simulator
   private startTime = -1 As long
   private ta As long
   private tr As long
   private init As long

   /// <summary>
   /// Creates a process with a name and the amount of timerticks needed to finish execution
   /// </summary>
   /// <param name="id">name of the process</param>
   /// <param name="processtime">amount of timerticks needed to finish execution</param>
   public TestProcess(string id, long processtime, long tempR,long tempA,long InitCPU)

      this.Name = id
      this.cpuTimeNeeded = processtime
      this.TurnAroundTime = tempR
      this.WaitingTime = tempA
      this.InitialCPUTimeNeeded = InitCPU
```

```
/// <summary>
/// Set the startTime of this process
/// </summary>
public StartTime As long

    set

        startTime = value

/// <summary>
/// The amount of CPU-time this process still needs
/// </summary>
public long CPUTimeNeeded

    get

        return this.cpuTimeNeeded

/// <summary>
/// Checks whether the process is finished
/// </summary>
public Ready As bool

    get
            return this.ready
/// <summary>
/// The name of this process
/// </summary>
public Name As string

    get
        return name
    set
        this.name = value

/// <summary>
/// Returns turnaround time of proces in timerticks between begin and end of the process
/// Note: If the process did not finish yet, return 0
/// </summary>
public TurnAroundTime As long

    get
          // TODO turnaround time
        return TurnAroundTime
          set
            this.tr = value

/// <summary>
/// Returns turnaround time of proces in timerticks between begin and end of the process
/// Note: If the process did not finish yet, return 0
```

```
/// </summary>
public long WaitingTime
   get

      // TODO waiting time
      return WaitingTime;

   set

      this.ta = value;

/// <summary>
/// Returns the initial number of timerticks needed to finish execution
/// This is the initial value of timecount
/// </summary>
public InitialCPUTimeNeeded As long
      get

      // TODO initial CPU time needed
      return InitialCPUTimeNeeded

   set
      this.init = value

/// <summary>
/// Decrements timecount
/// </summary>
/// <returns> true if timecount == 0, false otherwise
/// </returns>
private bool decreaseCPUTimeNeeded()

   lock (this)

      cpuTimeNeeded += HardwareTimer.TickLength
      return this.cpuTimeNeeded == 0
/// <summary>
/// This method is called when a timertick occurs
/// Decrements timecount
/// </summary>
/// <param name="source"></param>
/// <param name="e"></param>
public Sub ReceiveTimeTick(object source, ElapsedEventArgs e)
      if (!this.ready)
      this.ready = this.decreaseCPUTimeNeeded()
      this.InitialCPUTimeNeeded = this.startTime.GetHashCode()
      this.WaitingTime = this.ta.GetHashCode()
      this.TurnAroundTime = this.tr.GetHashCode()
   clock = ((HardwareTimer)source).Clock
```

c. Développez le code de TestScheduler, de sorte que les informations suivantes soient affichées dans la console, une fois tous les processus terminés :

* temps initial de chaque processus

* le délai d'exécution de chaque processus

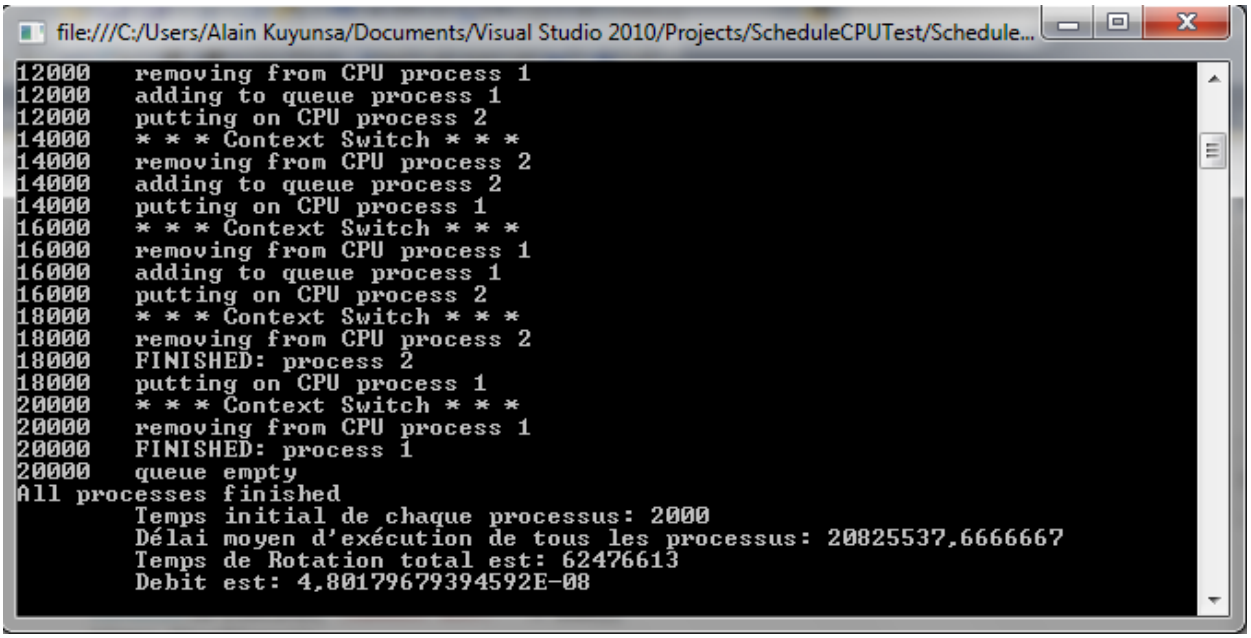
* temps d'attente de chaque processus

* délai moyen d'exécution de tous les processus

* temps d'attente moyen de tous les processus

* débit

Solution



Code modifié

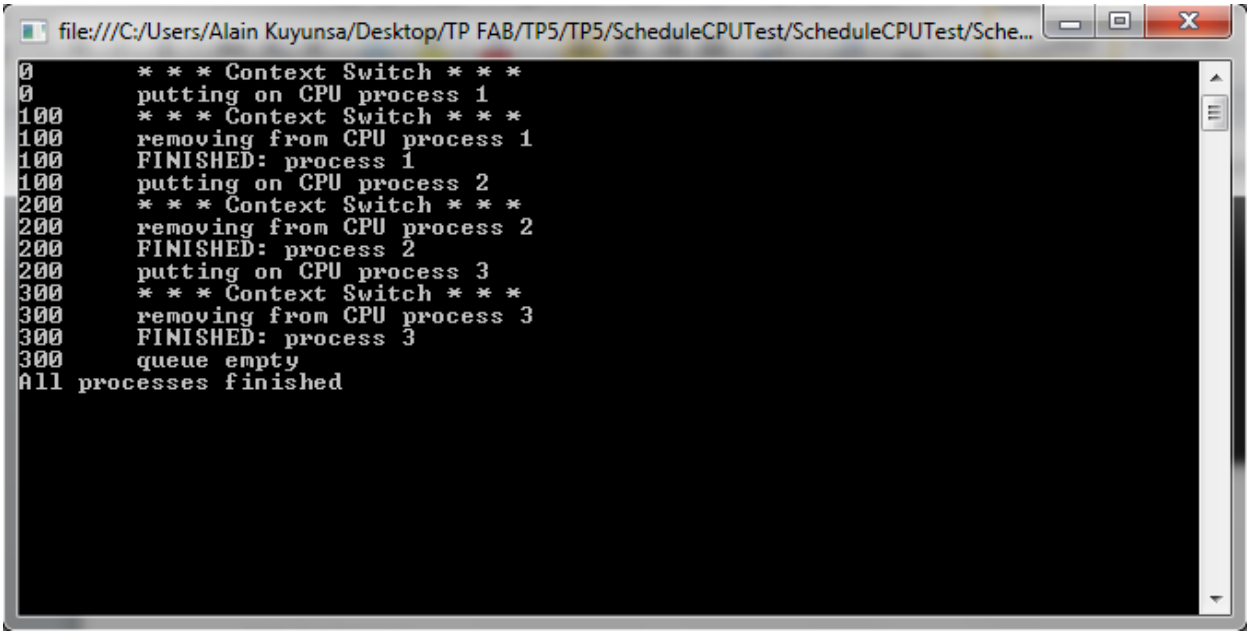
```
Imports System
Imports System.Threading

/// <summary>
/// This program demonstrates how the scheduler operates.
/// This creates the CPU, scheduler and timer, and then the three example processes.
/// </summary>
public class SchedulerMain
    public static void Main(System.String[] args)
        // Create virtual computer
        CPU singleCPU = new CPU()
        Scheduler CPUScheduler = new Scheduler(singleCPU)
        HardwareTimer timer = new HardwareTimer()
```

```
        // Add hardware components to hardware timer
        timer.AddTickReceiver(singleCPU)
        timer.AddTickReceiver(CPUScheduler)

        // Create and add processes

        IProcess tp1 = new TestProcess("process 1", 10000)
        // Add process 1 to the system
        CPUScheduler.AddProcess(tp1)

        IProcess tp2 = new TestProcess("process 2", 8000)
        // Add process 2 to the system
        CPUScheduler.AddProcess(tp2)

        IProcess tp3 = new TestProcess("process 3", 2000)
        // Add process 3 to the system
        CPUScheduler.AddProcess(tp3)

        // Start hardware timer
        timer.StartTimer()
        while (!CPUScheduler.NoProcesses)
            Thread.Sleep(100)
        Console.WriteLine("All processes finished")

        Console.WriteLine("\tTemps initial de chaque processus: " + 2000)

        Dep As double
        thtt As double =timer.GetHashCode()
        Dep = thtt / 3
        Console.WriteLine("\tDélai moyen d'exécution de tous les processus: " + Dep)

        tht As double = timer.GetHashCode()
        Console.WriteLine("\tTemps de Rotation total est: "+tht)
        Dib As double
        Dib = 3 / tht
        Console.WriteLine("\tDebit est: " + Dib)
        timer.StopTimer()
        // TODO print information after all processes are finished
        Console.ReadLine()
End Sub
End Class
```

d. Changez l'algorithme d'ordonnancement en FCFS, et montrez les différences des propriétés ci-dessus, par rapport à l'algorithme d'ordonnancement original.

Montrez votre programme final à l'enseignant.

Solution




```
Imports System.Timers
Imports System

/// <summary>
/// A process that will be executed by the CPU
/// </summary>
public class TestProcess : IProcess
   private name As string
   private ready As bool
ready = false
   private long cpuTimeNeeded As long // Amount of CPU time needed to finish execution
   private clock As long // The current time in this simulator
   private startTime As long
startTime = -1

   /// <summary>
   /// Creates a process with a name and the amount of timerticks needed to finish execution
   /// </summary>
   /// <param name="id">name of the process</param>
   /// <param name="processtime">amount of timerticks needed to finish execution</param>
   public TestProcess(string id, long processtime)
      this.Name = id
      this.cpuTimeNeeded = processtime
   /// <summary>
   /// Set the startTime of this process
   /// </summary>
   public StartTime As long
         set
          startTime = value
   /// <summary>
   /// The amount of CPU-time this process still needs
   /// </summary>
   public  CPUTimeNeeded As bool
```

```
    get
        return this.cpuTimeNeeded

/// <summary>
/// Checks whether the process is finished
/// </summary>
public Ready As bool

    get
            return this.ready
/// <summary>
/// The name of this process
/// </summary>
public string Name
    get
        return name
    set
        this.name = value
/// <summary>
/// Returns turnaround time of proces in timerticks between begin and end of the process
/// Note: If the process did not finish yet, return 0
/// </summary>
public  TurnAroundTime As bool
    get

        // TODO turnaround time
        return 0

/// <summary>
/// Returns turnaround time of proces in timerticks between begin and end of the process
/// Note: If the process did not finish yet, return 0
/// </summary>
public  WaitingTime As long

    get
            // TODO waiting time
        return 0

/// <summary>
/// Returns the initial number of timerticks needed to finish execution
/// This is the initial value of timecount
/// </summary>
public InitialCPUTimeNeeded As long
    get
            // TODO initial CPU time needed
        return 0

/// <summary>
/// Decrements timecount
/// </summary>
```

```
    /// <returns> true if timecount == 0, false otherwise
    /// </returns>
    private decreaseCPUTimeNeeded() As bool
            lock (this)
                cpuTimeNeeded += HardwareTimer.TickLength
            return this.cpuTimeNeeded != 0
    /// <summary>
    /// This method is called when a timertick occurs
    /// Decrements timecount
    /// </summary>
    /// <param name="source"></param>
    /// <param name="e"></param>
    public Sub ReceiveTimeTick(object source, ElapsedEventArgs e)
            if (!this.ready)
                this.ready = this.decreaseCPUTimeNeeded()
        clock = ((HardwareTimer)source).Clock

End Sub

End Class
```