Одеський національний політехнічний університет

Кафедра комп′ютеризованих систем управління

**Курсова робота**

з дицсциплiни **«Современные технологии программирования»**

Студента  *3*  курсу *АТ-173* групи

напрям підготовки  *151*

спеціальності *Автоматизацiя та комп`ютерно iнтегрованi технологiї*

*Филин Д.В.*

Керівник  *доц. Сперанский В.А.*

Національна шкала \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_\_\_ Оцінка: ECTS \_\_\_\_

Члени комісії \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (прізвище та ініціали

м. Одеса – 2019 рік

# ВВЕДЕНИЕ

Данная курсовая работа на тему: “Обслуживание процессором ЭВМ очереди готовых заданий” является теоретической, ориентированной на изучение концепций построения операционных систем и методов управления ресурсами процессора. Изучению этой дисциплины должно предшествовать изучение таких дисциплин, как “Программирование”, “Архитектура ЭВМ”, “Объектно-ориентированное программирование”.

Целью курсовой работы является изучение основных методов, используемых при управлении ресурсами в различных операционных системах.

Задачей курсовой работы является получение, как теоретических знаний, так и практических навыков, достаточных для проектирования и программирования системного программного обеспечения современных компьютеров, ознакомление с проблемами моделирования и анализа эффективности функционирования реальных вычислительных систем.

# Краткая теоретическая часть

## Назначение и функции операционной системы

***Операционная система (ОС)*** – это организованная совокупность программ и данных, которая выполняет функции посредника между пользователями и компьютером. ОС служит двум целям: во-первых, сделать компьютерную систему удобной для использования, и, во-вторых, эффективно использовать аппаратные средства компьютера.

ОС является *управляющей**программой*. Управляющая программа контролирует выполнение программ пользователей для предотвращения ошибок и неправильного использования компьютера.

ОС реализует множество различных функций, в том числе:

* определяет так называемый интерфейс пользователя;
* обеспечивает разделение аппаратных ресурсов между пользователями;
* дает возможность работать с общими данными в режиме коллективного пользования;
* планирует доступ пользователя к общим ресурсам;
* обеспечивает эффективное выполнение операций ввода-вывода;
* осуществляет восстановление информации и вычислительного процесса в случае ошибок.

Операционные системы могут различаться особенностями реализации внутренних алгоритмов управления основными ресурсами компьютера (процессорами, памятью, устройствами), особенностями использованных методов проектирования, типами аппаратных платформ, областями использования и многими другими свойствами.

В зависимости от особенностей использованного алгоритма управления процессором, операционные системы делят на многозадачные и однозадачные, многопользовательские и однопользовательские, на системы, поддерживающие многопоточную обработку и не поддерживающие ее, на многопроцессорные и однопроцессорные системы.

Важное влияние на облик операционной системы в целом, на возможности ее использования в той или иной области оказывают особенности и других подсистем управления локальными ресурсами ⎯ подсистем управления памятью, файлами, устройствами ввода-вывода.

Специфика ОС проявляется и в том, каким образом она реализует сетевые функции: распознавание и перенаправление в сеть запросов к удаленным ресурсам, передача сообщений по сети, выполнение удаленных запросов.

Многозадачные ОС подразделяются на три типа в соответствии с использованными при их разработке критериями эффективности:

* системы пакетной обработки (например, OC EC),
* системы разделения времени (UNIX, VMS),
* системы реального времени (QNX, RT/11).

Некоторые операционные системы могут совмещать в себе свойства систем разных типов, например, часть заданий может выполняться в режиме пакетной обработки, а часть ⎯ в режиме реального времени или в режиме разделения времени. В таких случаях режим пакетной обработки часто называют фоновым режимом.

Описать операционную систему можно только путем деления ее на меньшие компоненты. Не все ОС имеют одинаковую структуру. Однако во многих современных ОС ставится следующие компоненты:

* управление процессами;
* управление основной (оперативной) памятью;
* управление вторичной (внешней) памятью;
* управление вводом-выводом;
* управление файлами;
* защита системы;
* сетевое обслуживание;
* система интерпретации команд.

## Управление процессами

Важнейшей частью операционной системы, непосредственно влияющей на функционирование вычислительной машины, является подсистема управления процессами. ***Процесс*** (или по-другому, задача) - абстракция, описывающая выполняющуюся программу. Для операционной системы процесс представляет собой единицу работы, заявку на потребление системных ресурсов. Подсистема управления процессами планирует выполнение процессов, то есть распределяет процессорное время между несколькими одновременно существующими в системе процессами, а также занимается созданием и уничтожением процессов, обеспечивает процессы необходимыми системными ресурсами, поддерживает взаимодействие между процессами.

Жизненный цикл процесса может быть разбит на несколько состояний. Различают следующие состояния процесса:

* ***новый*** *(new****)*** — процесс только что создан, но еще не готов к выполнению;
* ***выполняемый*** *(running)* — команды процесса выполняются центральным процессором;
* ***ожидающий*** *(waiting****)*** — процесс ожидает наступления некоторого события (например, завершения ввода-вывода или поступления сигнала);
* ***готовый*** *(ready****)*** — процесс готов к выполнению и ожидает освобождения центрального процессора;
* ***завершенный*** (*terminated***)** — процесс завершил выполнение, но еще не удален из системы.

В ходе жизненного цикла каждый процесс переходит из одного состояния в другое в соответствии с алгоритмом планирования процессов, реализуемым в данной операционной системе.

### 

### Контекст и дескриптор процесса

На протяжении существования процесса его выполнение может быть многократно прервано и продолжено. Для того, чтобы возобновить выполнение процесса, необходимо восстановить состояние его операционной среды. Состояние операционной среды отображается состоянием регистров и программного счетчика, режимом работы процессора, указателями на открытые файлы, информацией о незавершенных операциях ввода-вывода, кодами ошибок, выполняемых данным процессом системных вызовов и т.д. Эта информация называется ***контекстом процесса***.

Кроме этого, операционной системе для реализации планирования процессов требуется дополнительная информация: идентификатор процесса, состояние процесса, данные о степени привилегированности процесса, место нахождения кодового сегмента и другая информация. В некоторых ОС (например, в ОС UNIX) информацию такого рода, используемую ОС для планирования процессов, называют ***дескриптором процесса***.

Очереди процессов представляют собой дескрипторы отдельных процессов, объединенные в списки. Таким образом, каждый дескриптор, кроме всего прочего, содержит, по крайней мере, один указатель на другой дескриптор, соседствующий с ним в очереди. Такая организация очередей позволяет легко их переупорядочивать, включать и исключать процессы, переводить процессы из одного состояния в другое.

Процессы в системе могут выполняться параллельно и должны создаваться и удаляться динамически. Поэтому ОС должна предоставлять механизмы порождения и прекращения процессов.

# Очередь готовых процессов

### CPU

N-cores

Ресурс R1

**Очередь к ресурсу R1**

Ресурс R2

**Очередь к ресурсу R2**

Ресурс R3

**Очередь к ресурсу R3**

**Квант времени исчерпан**

**Вытеснение**

Рис 2 – Диаграмма движения процессов

### 

### Планирование процессов

Планирование процессов включает в себя решение следующих задач:

* определение момента времени для смены выполняемого процесса;
* выбор процесса на выполнение из очереди готовых процессов;
* переключение контекстов "старого" и "нового" процессов.

Первые две задачи решаются программными средствами, а последняя в значительной степени аппаратно

Существует множество различных алгоритмов планирования процессов, по-разному решающих вышеперечисленные задачи, преследующих различные цели и обеспечивающих различное качество мультипрограммирования. Среди этого множества алгоритмов рассмотрим подробнее две группы наиболее часто встречающихся алгоритмов: алгоритмы, основанные на *квантовании,* и алгоритмы, основанные на *приоритетах.*

Существует два основных типа процедур планирования процессов - вытесняющие (preemptive) и невытесняющие (non-preemptive).

*Non-preemptive multitasking -* ***невытесняющая многозадачность*** *-* это способ планирования процессов, при котором активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление планировщику операционной системы для того, чтобы тот выбрал из очереди другой, готовый к выполнению процесс.

*Preemptive multitasking -* ***вытесняющая многозадачность*** *-* это такой способ, при котором решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается планировщиком операционной системы, а не самой активной задачей.

## Критерии планирования работы процессора

* Загрузка центрального процессора (КПД) измеряется в процентах.
* Пропускная способность процессора - количество процессов, выполненных в единицу времени.
* Время ожидания - интервал времени с момента появления процесса во входной очереди до момента его завершения.
* Время оборота - суммарное время нахождения процесса в очереди готовых процессов.

### Время отклика - время с момента попадания процесса во входную очередь до момента первого обращения к процессору Планирование по принципу SJP («кратчайшее задание—первым»)

Принцип SJF («кратчайшее задание — первым») — это дисциплина планирования без переключения, согласно которой следующим для выполнения выбирается ожидающее задание (или процесс) с минимальным оценочным рабочим временем, остающимся до завершения. Принцип SJF обеспечивает уменьшение среднего времени ожидания по сравнению с дисциплиной FIFO. Однако времена ожидания при этом колеблются в более широких пределах (т. е. менее предсказуемы), чем в случае FIFO, особенно при больших заданиях.

Дисциплина SJF оказывает предпочтение коротким заданиям (или процессам) за счет более длинных. Многие разработчики считают, что чем короче задания, тем лучшие условия для его выполнения следует создавать. Далеко не все специалисты придерживаются подобного мнения, особенно применительно к случаям, когда необходимо учитывать приоритетность заданий.

Механизм SJP выбирает для обслуживания задания таким образом, чтобы очередное задание завершалось и покидало систему как можно быстрее. Благодаря этому обеспечивается уменьшение количества ожидающих заданий, а также количества заданий, стоящих в очереди за длинными заданиями. В результате дисциплина SJF позволяет свести к минимуму среднее время ожидания для заданий, проходящих через систему.

Очевидная проблема, связанная с реализацией принципа SJF, состоит в том, что он требует точно знать, сколько времени будет выполняться задание или процесс, а такой информации обычно не бывает. Самое большее, что может сделать механизм SJF — это полагаться на оценочные значения времен выполнения, указываемые пользователями. В условиях производственного счета, когда одни и те же задачи выполняются регулярно, довольно точные и обоснованные оценки, по-видимому, возможны. Однако в исследовательской работе пользователи редко знают, как долго будут выполняться их программы.

Ориентация механизма планирования на оценки пользователей приводит к любопытным последствиям. Если пользователи знают, что система оказывает предпочтение заданиям с малыми оценочными временами выполнения, они могут задавать малые оценочные времена. Можно, однако, спроектировать планировщик заданий таким образом, чтобы избавить их от подобного соблазна. Пользователя можно заранее предупредить о том, что в случае, если его задание будет выполняться дольше оценочного времени, то оно будет просто выводиться из системы, причем пользователю придется платить за всю работу. Второе возможное решение заключается в том, чтобы выполнять задание в течение указанного оценочного времени плюс небольшой дополнительный процент, а затем «консервировать» задание, т. е. запоминать его в текущем виде, с тем чтобы впоследствии можно было вновь продолжить его выполнение. При этом пользователю, естественно, придется оплачивать затраты на консервацию и последующую расконсервацию (т. е. перезапуск), и, кроме того, он пострадает от задержки в завершении его задания. Существует еще одно возможное решение — выполнять задание в течение оценочного времени по обычному тарифу, а затем за до­полнительное время предъявлять счет по повышенному тарифу, значительно превышающему обычный. При таких условиях пользователь, указывающий нереально низкие оценочные времена, чтобы получить улучшенное обслуживание, будет в итоге платить значительно больше, чем обычно.

Принцип SJF подобно FIFO—это дисциплина обслуживания без переключения, поэтому ее не рекомендуется применять в системах разделения времени, где необходимо гарантировать приемлемые времена ответа

**Вариант 18**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n/n |  | Стратегия планирования | Наличие вытеснения | Способ организации очереди | | | | Динамическое повышение приоритета | Критерий вытеснения для SJF | |
|  | упорядоченный  список | не упорядочен.  список | список частично упорядочивается  через t тактов | каждому  приоритету своя очередь |
|  | По интервалу непрерывного выполнения | По оставшемуся времени |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 18. | 3 | SJF | - |  | + |  |  |  |  |  |

public sealed partial class ProcessDomain

    {

        [Required(AllowEmptyStrings = false, ErrorMessage = "Need Process Name"), MinLength(20, ErrorMessage = "Invalid Name")]

        public string ProcessName { get; set; }

        [Required(ErrorMessage = "Need Process ID")]

        public int ProcessID { get; set; }

        [Required(ErrorMessage = "Need Process Completion Time")]

        public int CompletionTime { get; set; }

        [Required(ErrorMessage = "Need Arrival Time")]

        public int ArrivalTime { get; set; }

        [Required(ErrorMessage = "Need Burst Time")]

        public int Burst\_Time { get; set; }

        public ProcessPriority Priority { get; set; }

        public ProcessState State { get; set; }

    }

    public sealed partial class ProcessDomain

    {

        static int count = 0;

        public ProcessDomain()

        {

            ProcessName = $"P{count++}";

            ProcessID = count;

            CompletionTime = SimpleRandomGenerator.GetRendomTime();

            ArrivalTime = SimpleRandomGenerator.GetRendomTime();

            Burst\_Time = SimpleRandomGenerator.GetRendomTime();

            State = ProcessState.New;

            Priority = ProcessPriority.Normal;

        }

        public ProcessDomain(string Name, int processID, ProcessState state, ProcessPriority priority)

            : this(Name, processID, 0, 0, 0, state, priority)

        {

            CompletionTime = SimpleRandomGenerator.GetRendomTime();

            ArrivalTime = SimpleRandomGenerator.GetRendomTime();

            Burst\_Time = SimpleRandomGenerator.GetRendomTime();

        }

        public ProcessDomain(string Name, int processID, int coTime, int arTime, int brTime, ProcessState state, ProcessPriority priority)

        {

            ProcessName = Name;

            ProcessID = processID;

            CompletionTime = coTime;

            ArrivalTime = arTime;

            Burst\_Time = brTime;

            State = state;

            Priority = priority;

        }

        public override string ToString()

        {

            StringBuilder builder = new StringBuilder();

            return builder.Append($"Process Name: {ProcessName}\n Process ID: {ProcessID}\n Completion Time: {CompletionTime}\n Arrival Time:{ArrivalTime}\n " +

                $"Burst Time:{Burst\_Time}\n Process State:{State}\n Priority:{Priority}").ToString();

        }

    }

Данный код предаставляет собой Базовую модель реализации каждого взятого процесса в отдельности. Класс закрыт для наследования и разбит на две составляющих, первая из которых набор свойств и второе конструкторы с методов ToString() для вывода информации на экран.

public enum ProcessState

    {

        Completed,

        Running,

        New

    }

public enum ProcessPriority

{

    High,

    Normal,

    Low,

    VeryLow

}

Представленные два выше перечилсения являются показываю состояния процесса во время исполнения и его приоритет непосредственно в ОС.

В реализации программы приоритет взят условно, так как чистый алгоритм SJF не представляет из себя работу с приоритетами.

public static class SimpleRandomGenerator

{

    private static Random randomGenerator;

    static SimpleRandomGenerator()

    {

        randomGenerator = new Random();

    }

    public static int GetRendomTime()

    {

        return randomGenerator.Next(1, 5);

    }

}

SimpleRandomGenerator класс представляет задания время выполнения процесса псевдорандомным образом. Статическим он объявлен потому что нету надобности в создании его экземпляра, а нужна работа с методом GetRandomTime.

public class ClockGenerator

    {

        static int count = 0;

        public ClockGenerator(TimerCallback callback, Timer timer)

        {

            // устанавливаем метод обратного вызова

            callback = new TimerCallback(Count);

            // создаем таймер

            timer = new Timer(callback, count, 0, 1000);

        }

        private void Count(object obj)

        {

            count++;

            Console.WriteLine($"{count} seconds have passed");

        }

        public void Stop()

        {

            try

            {

                Environment.Exit(0);

            }

            catch (ThreadAbortException threadEx)

            {

                Console.WriteLine("All processes done! {0}", threadEx.Message);

            }

        }

    }

Даныый класс явлеят из себя тактовый генератор. В приложении WindowsForms используется класс Timer и Stopwatch. Класс сделан для тестирования приложения в консоле.

public class SJFScheduler

   {

       List<ProcessDomain> processes;

       List<ProcessDomain> runningProcess;

       readonly object SJFLock = new object();

       bool workDone = false;

       public SJFScheduler()

       {

           processes = new List<ProcessDomain>();

           runningProcess = new List<ProcessDomain>();

       }

       public SJFScheduler(IEnumerable<ProcessDomain> Processes)

       {

           processes = new List<ProcessDomain>();

           runningProcess = new List<ProcessDomain>();

           processes.AddRange(Processes);

       }

       public void AddProcess(ProcessDomain process)

       {

           workDone = false;

           if (workDone == false)

           {

               lock (SJFLock)

               {

                   processes.Add(process);

               }

           }

       }

       public List<ProcessDomain> GetProcesses

       {

           get => processes;

       }

       public async Task<List<ProcessDomain>> StartConvertToRunningOperation(CancellationToken Token)

       {

           return await WorkWithRunningList(Token);

       }

       public async Task<List<ProcessDomain>> Start(CancellationToken Token)

       {

           return await Work(Token);

       }

       private async Task<List<ProcessDomain>> Work(CancellationToken ct)

       {

           List<ProcessDomain> workResult = new List<ProcessDomain>();

           while (runningProcess.Exists(p => p.State == ProcessState.Running) && !ct.IsCancellationRequested)

           {

               ProcessDomain execProc = null;

               lock (SJFLock)

               {

                   int minDuration = runningProcess.Where(proc => proc.State == ProcessState.Running).Min(p => p.Burst\_Time);

                   execProc = runningProcess.First(p => p.Burst\_Time == minDuration && p.State == ProcessState.Running);

                   execProc.State = ProcessState.Completed;

                   workResult.Add(execProc);

               }

               //for work imitation

               Thread.Sleep(execProc.Burst\_Time \* 1000);

           }

           workDone = true;

           return workResult;

       }

       private async Task<List<ProcessDomain>> WorkWithRunningList(CancellationToken ct)

       {

           foreach (var item in processes)

           {

               if (processes.Exists(proc => proc.State == ProcessState.Running))

                   item.State = ProcessState.New;

               else if (processes.Exists(proc => proc.State == ProcessState.Interrupted))

                   item.State = ProcessState.New;

               else if (processes.Exists(proc => proc.State == ProcessState.Completed))

                   item.State = ProcessState.New;

           }

           while (processes.Exists(procRun => procRun.State == ProcessState.New))

           {

               ProcessDomain tempDomainObject = null;

               lock (SJFLock)

               {

                   if (ct.IsCancellationRequested)

                   {

                       ct.ThrowIfCancellationRequested();

                       tempDomainObject = processes.Find(p => p.State == ProcessState.New);

                       tempDomainObject.State = ProcessState.Interrupted;

                       runningProcess.Add(tempDomainObject);

                   }

                   else

                   {

                       tempDomainObject = processes.Find(p => p.State == ProcessState.New);

                       tempDomainObject.State = ProcessState.Running;

                       runningProcess.Add(tempDomainObject);

                   }

               }

           }

           return runningProcess;

       }

   }

Класс SJFScheduler представляет из себя планировщик процессов по алгоритму SJF. В начале иницилизируются два списка:

1. List<ProcessDomain> processes;
2. List<ProcessDomain> runningProcess;

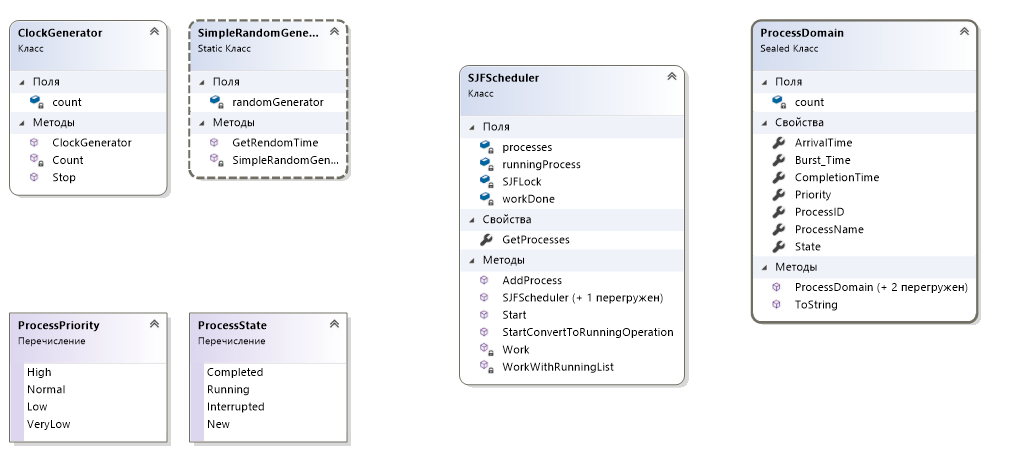
Один из которых содержит задачи переданные ему при создание объекта планировщика, а второй служит для перевода задач в действия. Так же имеется свойство GetProcesses которое возвращает список.

Среди методов у нас есть:

1. AddProcess(ProcessDomain) – служит для добавляения процессов в список. В Форме он реализован путем добавления через кнопку. Свойства ID и Имени процесса пользователь пишет сам, приоритет и состояние заранее даются процессу при создании, а свойства времени получает случайное значения с генератора случайных чисел.
2. Work(CancelletionToken) – метод выполняет основное действие для алгоритма SJF. Внутри метода мы создаем локальное хранилище в виде List<ProcessDomain> workResult который отвечает за сохранения процессов, которые имею статус Completed. Данный метод выполняется до того момента, пока в списке процессов содержаться незавершенные операции м до момента прирывания процесса. Оператор lock() обеспечивате синхронизацию между потоками/задачами, во избежания «борьбы» между ними. Задача замораживается на период того, сколько выполняется задача. (Конструкция Thread.Sleep(execProc.Burst\_Time \* 1000)).
3. WorkWithRunningList(CancelletionToken) – метод служит переводом состояний процесса в состояние Running. Так же с помощью оператора lock() обеспечиваться синхронизация между потоками/задачами.

Так же есть два метода, который являются обертками для задача:

StartConvertOperationToRunning(CancelletionToken) и Start(CancelletionToken).



**Рисунок 1.** Диаграмма классов Модели