МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Кафедра ИСУ

Сравнительный анализ управления приложениями (процессами) в ОС Unix-Linux и Windows

Курсовой проект

Наумович Андрей Игнатович
студент 3 курса, 2 группы
специальность «Информатика»

Научный руководитель:
Старший преподаватель Безверхий А.А.

Минск, 2018

ОГЛАВЛЕНИЕ

Реферат

Курсовой проект, 34 с., 10 источников, 11 таблиц.
СРАВНИТЕЛЬНЫЙ АНАЛИЗ УПРАВЛЕНИЯ ПРИЛОЖЕНИЯМИ (ПРОЦЕССАМИ) В ОС UNIX-LINUX И WINDOWS.

Объект исследования – Процессы. Концепция процесса в современных ОС. Особенности управления в ОС Unix-Linux.Особенности управления в ОС Windows.

Цель работы – Изучить основные концепции и особенности по источникам литературы. Проанализировать и описать общие черты и особенности исследуемой области в рассматриваемых системах. Подготовить один и тот же пример приложения, демонстрирующего управление процессами, реализовать на языке C/C++ в каждой из рассматриваемых систем. Указать достоинства и недостатки каждого из подходов.

Методы исследования – Работа с литературой, подготовка приложения.

Результаты – Рассмотрены типичные подходы к планированию процессов. Рассмотрены подходы к планированию в указанных ОС. Реализовано приложение в каждой из ОС. Приведены сравнения времени работы приложения в указанных ОС.

Рэферат

Курсавой праект, 34 с., 10 крыніц, 11 табліц.

ПАРАЎНАЛЬНЫ АНАЛІЗ КІРАВАННЯ ПРЫКЛАДНЫМІ ПРАГРАМАМІ (ПРАЦЭСАМІ) У АС UNIX-LINUX І WINDOWS.

Аб'ект даследавання – Працэсы. Канцэпцыя працэсу ў сучасных АС. Асаблівасці кіравання ў АС Unix-Linux. Асаблівасці кіравання ў АС Windows.

Мэта працы – Вывучыць асноўныя канцэпцыі і асаблівасці па крыніцах літаратуры. Прааналізаваць і апісаць агульныя рысы і асаблівасці доследнай вобласці ў дадзеных сістэмах. Падрыхтаваць адзін і той жа прыклад праграмы, які

дэманструе кіраванне працэсамі, рэалізаваць на мове C/C ++ у кожнай з дадзеных сістэм. Пазначыць вартасці і недахопы кожнага з падыходаў.

Метады даследавання – Праца з літаратурай, падрыхтоўка праграмы.

Вынікі – Разгледжаны тыповыя падыходы да планавання працэсаў. Разгледжаны падыходы да планавання ў названых АС. Рэалізавана прыкладанне ў кожнай з АС. Прыведзены параўнання часу працы прыкладання ў названых АС.

Abstract

Course project, 34 p., 10 sources, 11 tables.

COMPARATIVE ANALYSIS OF APPLICATION (PROCESSES) MANAGEMENT IS OS UNIX-LINUX AND WINDOWS.

Object of research – Processes. Process concept in modern OS. Special features of process management of OS Unix-Linux. Special features of process management of OS Windows.

Purpose – Learn the basic concepts and features using the literary sources. Analyze and describe the general features and specialities of the studied area in the systems under consideration. Develop the same sample application that demonstrates process management, implement in each of the systems under consideration using C/C++ . Identify the advantages and the disadvantages of each approach considered.

Research methods – Literature exploring, implementing application.

Results – Typical process management approaches were considered. Process management approaches in listed OS were studied. Application was implemented to run under OS provided. Comparasion of the application operating time provided in listed OS.

ВВЕДЕНИЕ

Современные компьютеры выполняют множество задач одновременно. К примеру, во время работы операционной системы одновременно исполняется несколько задач: обновление графического интерфейса, обработка нажатий клавиш клавиатуры, мыши, работа фоновых служб. Для поддержания системы в актуальном состоянии и моделирования параллельной обработки необходима поддержка механизмов и алгоритмов, назначающих задачи на выполнение. В работе будут рассмотрены алгоритмы планирования процессов и их варианты реализаций в некоторых операционных системах.

Основные определения и понятия

Блокировка процесса - остановка выполнения всех инструкций процесса. Одной из причин появления является необходимость синхронизации выполнения множества процессов.
Квант - временной интервал, в течение которого процесс может выполняться.
Оборотное время - среднее время, прошедшее между отправлением задачи на обработку и получением решения. Вычисляется статистически.
Приоритет процесса - характеристика процесса, в каком-то смысле обозначает требовательность процесса к ресурсам системы.
Производительность - число заданий, выполняемых в единицу времени.
Процесс - выполняющаяся программа и все ее элементы: адресное пространство, глобальные переменные, регистры, стек и так далее.
Система разделения времени - система, поддерживающая распределение вычислительных ресурсов между многими пользователями с помощью мультипрограммирования и многозадачности.
Такт процессора - промежуток времени между двумя последовательными срабатываниями системного таймера.
ЦП - центральный процессор.
является необходимость синхронизации выполнения множества процессов.
LIL - List Of Lists - список , каждый элемент которого является списком.
Обозначение « - много меньше.

Глава 1. Система управления процессами.

Данная глава рассматривает основные характеристики процессов, а также некоторые стратегии планирования выполнения процессов в однопроцессорной среде.

1.1 Общая информация о процессах.

1.1.1 Состояния процессов.

Непосредственно во время выполнения процесс может находиться в основных трех состояниях [1]:

— Блокировка.

— Выполнение.

— Готовность.

Блокировка. Процесс блокируется из-за того, что его выполнение не может быть продолжено по тем или иным соображениям. Наиболее частые из них - решение планировщика поставить на выполнение другую задачу, зависимость от данных, которые еще не были вычислены, ожидание какого-либо внешнего события. Соответственно, из этого состояния он может быть выведем возникновением события, разрешающего возникшую зависимость.
Выполнение. Процесс в текущий момент времени занимает ЦП машины.
Готовность. Данные, необходимые для работы, текущего процесса получены и процесс ожидает решения ОС предоставить ему процессорное время.
Переходы между состояниями выполнения и готовности тесно связаны с работой планировщика процессов. Его основной миссией является задача распределения процессорного времени между процессами таким образом, чтобы обеспечить наилучшую производительность, уменьшить энергопотребление и так далее.
Понятно, что процесс просто так появиться и начать исполняться в системе не может. Механизм создания процессов должен быть реализован таким образом, чтобы эта операция не приводила к серьезным задержкам в обработке других задач или вообще к приостановке всей системы. На этом этапе семейства ОС реализуют такую функциональность следующим образом:

— Unix+Linux. Основа системы процессов - строгая иерархия (отношения типа предок-родитель). Иерархию процессов удобно представлять в виде корневого дерева. При этом каждая из вершин - родительский процесс для

множества процессов, лежащих в поддеревьях этой вершин. Такое устройство позволяет эффективно организовывать обработку системных сигналов: сигнал приходит в корень дерева, по необходимости обрабатывается, и дублируется во все поддеревья. Например, в системе UNIX при инициализации системы создается дерево, корень которого - процесс $init$. Этот процесс разветвляется, порождая по одному процессу на каждый из терминалов. После этого продолжается наращивание дерева от новых оболочек приема команд. Порождение нового процесса явно разделено на 2 этапа:

— Создание точной копии родительского процесса, выделение копии уникального идентификатора процесса, добавление записи в таблицу процессов с указанием идентификатора дочернего процесса, содержимое адресного пространства также копируется.

— Помещение нового исполняемого кода в дочерний процесс. (Опционально)

После этого родительский и дочерний процессы продолжают свое выполнение независимо друг от друга.

— Windows. В системе Windows кроме идентификатора процесс идентифицируется с помощью дескриптора процесса [5] [4]. Идентификатор уникален и неизменен, в то время как дескрипторов для одного и того же процесса может существовать несколько. Это позволяет регулировать разрешения доступа к процессу, например, наследование дескриптора. Регулирование параметров доступа подразумевает, что с процессом могут синхронизировать свое выполнение другие процессы

После того, как процесс выполнит свою задачу, необходимость в выделении процессорного времени ему отпадает. Однако, запись в таблице дескрипторов для завершившегося процесса может быть не удалена сразу. Это объясняется особенностями организации системы процессов ОС.

1.1.2 Контекст процесса.

Можно сказать, что процесс характеризуется своим контекстом. **Контекст процесса** включается в себя: значения регистров, указатель стека, текущее состояние процесса, счетчик команд, идентификатор, информация об использованном процессорном времени, а для многопроцессорных систем может также указываться конкретный тип процессора, на котором желательно выполнение.
Предположим, что необходимо сменить текущий исполняющийся процесс на другой. Каждый из процессов должен работать только со своим контекстом, чтобы обеспечить корректность вычислений. Таким образом , возникает необходимость

сменить текущий контекст на нужный. Контекст каждого процесса можно представить как вектор, условно разбитый на блоки. Например, первый блок может содержать id процесса, второй - состояние регистров и так далее. Объединив все такие векторы, получим таблицу, которая называется **таблицей процессов**. Таким образом, при смене текущего исполняющегося на ЦП процесса операционной системе достаточно обращаться в таблицу процессов, чтобы получить необходимый контекст.

1.1.3 Обработка прерываний таймера.

Компьютер всегда имеет встроенный таймер, срабатывающий через определенные временные интервалы. Например, стандартное время между 2 последовательными срабатываниями (такт процессора) для ОС Windows - около 16 мс, для Unix - около 10мс. Само по себе окончание такта процессора ничего не значит, но оно инициирует прерывание таймера.

Некоторые из задач, выполняемые обработчиком прерывания таймера:

— Действия, связанные с планированием процессов.

— Происходит пересчет времени, в течение которого процесс занимал процессор.

— Обновление системного времени.

Из-за того, что таймер срабатывает достаточно часто, обработка таких прерываний является одной из наиболее важных задач в системе управления процессами.

1.2 Введение в систему планирования.

1.2.1 Необходимость в системе планирования процессов.

Рассмотрим компьютер с одноядерным процессором и одним процессом A, исполняющим, например, алгоритм решения задачи Коши для дифференциального уравнения в частных производных. Решение задачи может занимать длительное время в силу того, что решение вычисляется на двумерной сетке узлов. Пока процесс в системе единственен, проблем не возникает.
Пусть появилась возможность запустить еще один процесс B, единственная цель которого - вывести на экран результат операции "$2 * 6$". Если не вмешиваться в выполнение процесса A, то он будет выполняться до тех пор, пока не выполнит всю

необходимую работу, затем он **добровольно** освободит ЦП, а его место займет процесс B. Однако, не совсем правильно, что процесс с малым потреблением ресурсов B должен дожидаться окончания выполнения требовательного к ресурсам процесса A. Более того, процессы могут использовать устройства ввода-вывода, сетевое соединение, взаимодействовать между собой и другими объектами системы, что представляет дополнительные сложности при планировании выполнения процессов.

Для разрешения вопроса, какой процесс должен вскоре выполняться, вводятся **алгоритмы планирования**, а исполняющий их объект называется **планировщиком процессов**.
Для некоторых платформ критично потребление ими электроэнергии, поэтому перед планировщиком процессов может ставиться задача повышения энергоэффективности:
Пусть имеется n процессов $P_1, P_2, ..., P_n$. Необходимо выработать такую стратегию назначения процессов на выполнение, чтобы минимизировать время простоя процессора.
В описанных выше ситуациях считалось, что все процессы не имеют абсолютно никакого отношения друг и другу, т.е. являются полностью независимыми. При решении некоторых задач бывает возможно сформулировать алгоритм решения так, чтобы какие-то части могли выполняться параллельно и независимо друг от друга. Например, при выполнении умножения матрицы на вектор происходит умножение каждой отдельной строки на этот вектор, независимо друг от друга. Такие операции при больших размерностях часто бывает выгодно проводить в отдельном процессе (при условии, что выигрыш во времени вычислений покроет издержки на создание процессов и затраты на планирование), аккумулируя результат позже.

1.2.2 Классификация сред выполнения.

Всякий алгоритм имеет свои слабые и сильные стороны, которые определяют сферу его применения. Алгоритмы планирования выполнения процессов не являются исключением. Стратегия работы планировщика процессов должна удовлетворять требованиям к системе, в которой он используется. Различают три основных типа сред выполнения [1]:

— **Системы реального времени**. Основная задача такой системы - завершение выполнения задачи в пределах жестких временным рамок. Как пример можно привести работу автопилота в современных самолетах. При возникновении непредвиденных факторов во время полета скорость отклика автопилота очень критична, поскольку этим определяется, будет ли сохранен контроль над полетом или будет утрачен.

Планировщик в такой системе должен планировать задачи так, чтобы каждая из них уложилась в свои временные рамки, обеспечить необходимое время отклика.

— **Системы пакетной обработки**. Основное предназначение - решение задач вычислительного характера, которые не требуют немедленного результата. Главными показателями качества работы системы пакетной обработки являются производительность и оборотное время. Существует третий показатель - загрузка центрального процессора, но он не несет важной информации. Соответственно, алгоритм планирования должен быть выбран таким образом, чтобы значительно увеличить или показатель производительности, или показатель оборотного времени, или оба,но в меньшей степени.

— **Интерактивные системы**. Для этого типа систем критическим является минимизация времени отклика на некоторое действие. Например, пользователь, запуская терминал и выполняя в нем команду поиска файлов с определенным шаблоном имени в текущей директории, ожидает начать получать результат практически немедленно, а не после того, как фоновый процесс загрузки обновлений для ОС завершит свою работу. Задачей планировщика в этом случае является обеспечение времени отклика, возможно, в ущерб некоторым другим (фоновым) процессам.

1.3 Планирование в системах реального времени.

Многие системы реального времени используют механизм событий для функционирования. Например, есть некоторое количество физических устройств - детекторов, которые при фиксировании заданного события отправляют запрос компьютеру, сообщающий, что этот сигнал нужно обработать, причем в сжатые сроки. Различают системы реального времени с **мягкими** и **жесткими** ограничениями.
Системы реального времени с **мягкими** ограничениями не дают никакой гарантии относительно крайнего срока завершения процесса, но просто гарантируют, что такому процессу будет оказано предпочтение при планировании. Системы реального времени с **жесткими** ограничениями, напротив, дают жесткую гарантию относительно времени завершения процесса. Стоит заметить, что реализация системы реального времени требует не только реализации специального алгоритма планирования, но также изменений в системе в целом. Так, например, если рассматривать сигналы от внешних детекторов как прерывания, то необходимо серьезное усовершенствование системы смены контекста процесса. При возникновении прерывания нужно сохранить контекст текущего процесса, загрузить обработчик прерывания,выполнить обработку, вернуть контекст прерванного процесса на исполнение.

1.3.1 EDF подход к планированию.

Свяжем приоритеты $pr_i(t)$ процессов p_i в текущий момент времени t с расстоянием до крайних сроков их завершения T_i, $i = 1..n$, например, следующим образом : $pr_i(t) = \frac{1}{T_i - t}$. Суть алгоритма заключена в следующем:

— Если в системе нет активных процессов реального времени, и поступает сигнал на инициализацию нового процесса, новый процесс создается и назначается на выполнение, вытесняя текущий процесс.

— Пусть в системе есть активные процессы реального времени. При создании нового процесса порождается системное прерывание, обработчик которого должен выполнить перерасчет приоритетов, так как может оказаться, что вновь поступивший процесс должен быть завершен раньше, чем все остальные процессы. На выполнение будет назначен только что поступивший процесс, поскольку его приоритет самый высокий в силу введенной формулы.

— При завершении выполнения некоторого процесса реального времени также выполняется перерасчет приоритетов, на выполнение назначается процесс с наибольшим приоритетом.

Описание алгоритма объясняет его название [10]: EDF - Earliest-Deadline-First Scheduling - Алгоритм планирования по ближайшему сроку завершения.
Стоит отметить, что приоритеты процессов являются динамическими, то есть могут и будут изменяться в процессе работы планировщика, причем значение приоритета обратно пропорционально времени нахождения процесса в очереди на выполнение. Более того, алгоритм EDF накладывает существенное ограничение: планировщику должны быть предоставлены крайние сроки завершения обработки процессов от каждого внешнего детектора (их нужно каким-то образом рассчитывать). Утверждается, что алгоритм является теоретически оптимальным в том смысле, что если существует такое распределение процессов в очереди на выполнение, что каждый будет завершен до своего крайнего срока, то оно будет найдено алгоритмом. Однако, на практике с этим могут возникнуть проблемы из-за издержек на промежуточные работы по смене контекста процесса. В этом случае крайние сроки придется сместить.

1.4 Планирование в пакетных системах.

1.4.1 Первый пришел - первый обслужен. (FIFO)

Самым простым является алгоритм "Первым пришел - первым обслужен" [7] [1]. При использовании такого алгоритма роль планировщика заключается в предо-

ставлении процессорного времени процессам в порядке "живой очереди". Приоритетом процесса в таком алгоритме является время поступления процесса на обработку: чем раньше он поступил, тем выше его приоритет. Процесс имеет право занимать ЦП сколь угодно долго: до завершения или до перехода в состояние блокировки. В последнем случае при пробуждении процесс помещается в конец очереди.

Недостатки алгоритма:

— Не учитывает особенностей процессов, кроме их приоритета. Из-за этого легко смоделировать ситуацию, когда процесс, не требовательный к ресурсам, может долго ожидать процессорного времени из-за гораздо более требовательных процессов, поступивших ранее. Также алгоритм не учитывает ограниченность по скорости вычислений и времени работы устройств ввода-вывода. Под этим понимается ресурс, к которому наиболее требователен процесс: ЦП или устройство ввода-вывода.

— Из-за простоты алгоритма его сложно масштабировать: при увеличении количества процессов в очереди существенно увеличивается оборотное время процесса.

Достоинства алгоритма:

— Весьма прост для понимания и реализации. При реализации можно отказаться от приоритетной очереди, заменив ее двусвязным списком. При поступлении нового процесса на обработку он помещается в конец списка, при назначении процесса на выполнение извлекается процесс из начала списка. Работа с началом и концом двусвязного списка осуществляется за гарантированное время $O(1)$, то есть эффективность смены исполняющегося процесса напрямую зависит от эффективности смены контекста процессов и не зависит от количества процессов в очереди.

1.4.2 Сначала самое короткое задание. (SJF)

Пусть имеется один процессор, зафиксировано множество процессов $\{p_i\}, i = 1..n$ и заранее известно необходимое время выполнения каждого процесса $\{T_i\}, i = 1..n$. Ставится задача минимизация среднего оборотного времени процесса. Рассмотрим алгоритм ее решения:

— Отсортируем процессы таким образом, что $T_{k_1} \leqslant T_{k_2} \leqslant ... \leqslant T_{k_n}$ и поместим в стек (самый быстрый процесс на верхушке).

— Пока стек не пуст, после завершения очередного процесса извлекаем новый процесс из стека и назначаем на исполнение.

Утверждается, что при таком назначении среднее время оборота процессов будет минимальным. Покажем это.
Пусть значения времени выполнения процессов расположены в стеке в произвольном порядке. Тогда для первого процесса время оборота будет T_{k_1}, для второго в стеке $T_{k_1} + T_{k_2}$, для j-го процесса : $\sum_{i=1}^{j} T_{K_i}$. Среднее время оборота равно среднему арифметическому: $T = \frac{\sum_{l=1}^{n}\sum_{i=1}^{j} T_{k_i}}{n} \to min$. Расписав сумму, нетрудно получить, что $T = T_{k_1} + \frac{n-1}{n}T_{k_2} + \frac{n-2}{n}T_{k_3} + ... + \frac{1}{n}T_{k_n}$. Понятно, что взаимный порядок T_{k_i} для минимизации T нужно выбрать следующим: $T_{k_1} \leqslant T_{k_2} \leqslant ... \leqslant T_{k_n}$.
Приведенный выше алгоритм известен под названием "Сначала самое короткое задание Shortest job first - (SJF) [1].
Недостатки:

— Требуется, чтобы новые процессы не поступали на протяжении всего времени выполнения.

— Если разрешить добавлять новые процессы, то поступивший процесс нужно добавить в стек, что в худшем случае требует $O(n)$ времени и $O(n)$ дополнительной памяти.

— Время работы каждого процесса также должно быть заранее известно. Частично этот недостаток можно нивелировать методами оценки времени выполнения.

— Предполагается, что процессы не прерываются на работу с вводом-выводом, а занимают непрерывный временной интервал для работы с ЦП.

Достоинства:

— Позволяет минимизировать среднее оборотное время при выполнении всех ограничений.

— На базе SJF можно строить дальнейшие модификации.

Рассмотрим одну из таких модификаций. Введем приоритет ”оставшееся время выполнения”. Чем меньше это числовое значение, тем выше приоритет. Теперь планировщик будет использовать приоритетную очередь вместо стека. Модификация позволяет быстрее добавлять новые процессы в очередь, а также задачи, не требовательные к ресурсам, будут иметь низкое оборотное время, поскольку будут находиться в начале очереди. С другой стороны, остается не решенной проблема с выполнением требовательных к ресурсам процессов, если в очередь постоянно поступают процессы, которые можно завершить за короткий промежуток времени.

1.5 Планирование в интерактивных системах.

1.5.1 Циклическая схема.

Наиболее простым алгоритмом планирования в интерактивных системах является циклический алгоритм [1] [7] . Вводится временной квант, в течение которого процесс может занимать ЦП. В данной реализации квант связан не с процессом, а с процессором. Планировщик процессов организует FIFO очередь, в которую помещаются все процессы в порядке поступления на выполнение. Предположим, что в данный момент времени ЦП занят некоторым процессом. Возможны следующие ситуации:

— Временной квант исчерпан, процесс завершен. В этом случае он снимается с выполнения, на его место поступает процесс из очереди.

— Временной квант исчерпан, но процесс не завершен. Он снимается с выполнения и помещается в конец очереди, его место занимает процесс из очереди.

— Временной квант не исчерпан, но процесс завершил выполнение или перешел в состояние блокировки. На исполнение поставим новый процесс из очереди, а текущий либо перенесем в конец, либо завершим.

Замечание. Квант обновляется, когда происходит смена контекста процесса и является одинаковым для всех процессов в очереди.

Достоинства:

— Простота реализации. Для хранения используется очередь, а такой тип квантования можно реализовать с помощью механизма прерываний, в частности, с помощью прерываний системного таймера [см. 1.1.3]. Стоит учесть, что смена контекста, вообще говоря, может происходить не на каждом прерывании таймера.

— Не дает требовательным к ресурсам процессам надолго занимать ЦП.

Недостатки:

— Не подходит для процессов, активно работающих с внешними устройствами. При ожидании результата операции, как уже отмечалось, процесс переходит в конец очереди и вынужден ожидать дольше, чем мог бы.

— Сложность выбора длительности кванта времени. Длительность тесно связана с эффективностью реализации смены контекста. Если квант времени сделать слишком малым, то расходы на смену контекста могут занимать значительную долю процессорного времени. Если квант окажется слишком большим, то такое планирование перестанет удовлетворять требованиям интерактивных систем - пользователь будет ожидать результата команды непозволительно долго.

Рассмотрим модификацию циклической схемы. Временной квант теперь будет ассоциирован с процессом, а не процессором. Числовая характеристика процесса будет храниться в его записи в таблице процессов. Заводится две очереди. В одной из них хранятся процессы, которые не исчерпали свой квант, в другой - процессы, полностью исчерпавшие свой квант. При смене контекста процесса будет приниматься решение, в какую из очередей поместить процесс, снимающийся с выполнения (исходя из того, исчерпал ли он свой квант). При добавлении процесса в очередь с процессами, исчерпавшими свой квант, обновим его личный квант. Если очередь с не исчерпавшими квант процессами опустеет, инвертируем роли очередей.

1.5.2 Следующим самое короткое задание. (SJN)

В интерактивных системах можно воспользоваться идеями алгоритма SJF [см. 1.4.2], поскольку достоинства SJF вписываются в их концепцию. В алгоритме SJN приоритетом будет предположительное время выполнения процесса, для вычисления которого рассмотрим несколько подходов:

— **Статическое предсказание.** Основывается на информации о процессах, которая, как следует из названия, является статической, то есть не изменяется во время выполнения процесса. Предположения о продолжительности выполнения можно выдвигать на основании размера дискового пространства, занимаемого исходным кодом программы, выполняющейся в процессе; на основании типа процесса в том смысле, что утилита командной строки, скорее всего, выполнится быстрее другого пользовательского процесса. Однако, такие оценки часто могут оказываться неверными и малоэффективными.

— **Динамическое предсказание.** Основывается на информации о процессе, полученной непосредственно во время выполнения. Предположим, что имеется информация о времени работы $T_1, T_2, ..., T_n$ конкретного процесса P исходя из предыдущих запусков этого процесса. На основании этих данных вводится **оценка по сроку давности (aging)**: $T = \sum_{k=0}^{n-1} (1-\alpha)^k T_{n-k}$. Параметр $a \in [0, 1]$ позволяет регулировать вклад запусков в оценку времени работы. Часто бывает удобно принимать $a = \frac{1}{2}$ в силу простоты деления на степень двойки.

Достоинства:

— Учет оценки времени выполнения при планировании.

— Пригоден для использования в системах, где работа пользователя завязана на системных утилитах (позволяет задействовать оценку по сроку давности).

Недостатки:

— Высокое время оборота для требовательных к системным ресурсам процессов.

— Внесение изменений в исходный код программы разработчиком, вообще говоря, сводит на нет усилия по формированию оценки по сроку давности (Может быть добавлен вызов подпрограммы, выполнение которой требует большого процессорного времени).

— Информацию, необходимую для оценивания по сроку давности, нужно хранить таким образом, чтобы скорость доступа к ней была крайне высока, поскольку от этого напрямую зависит быстродействие всей системы.

1.5.3 Справедливое планирование.

Важную идею в систему планирования вносит алгоритм справедливого планирования [1] . Пусть в системе есть N пользователей, каждый из которых запускает процессы, не обязательно в равных объемах. Алгоритмы, рассмотренные до этого, не распределяли процессорное время между пользовательскими группами процессов.
Предположим, что пользователь с номером i запустил K_i процессов, а пользователь j - K_j процессов, причем $K_i << K_j$. При использовании циклической схемы,как нетрудно заметить, пользователь j получит гораздо больше процессорного времени, нежели пользователь i. Из соображений "справедливости", объем задач одного пользователя не должен влиять на выделение процессорного времени другому пользователю. Будем полагать, что пользователь с номером i должен получить T_i долю процессорного времени, $\sum T_i = 1, T_i \geqslant 0, i = 1..n$ и ему принадлежит группа процессов P_i, суммарное время владения ЦП которой в момент времени T равно $S_i(T)$.

Тогда доля времени , в течение которого выполнялись процессы из группы P_i : $\tau_i = \frac{S_i(T)}{T}$. Возможны 2 ситуации:

— $\tau_i < T_i$. Группе процессов было выделено недостаточно времени, тогда планировщик будет выделять процессорное время этой группе.

— $\tau_i \geqslant T_i$. Процессам группы P_i было выделено больше времени, чем требуется. Эти процессы не будут выполняться, пока τ_i не станет меньше порога.

Достоинства:

— Алгоритм позволяет регулировать доступ к ЦП в многопользовательских системах.

— Планирование является справедливым.

Недостатки:

— Усложнение алгоритма и увеличение времени работы планировщика , поскольку сначала необходимо выбрать группу, затем процесс внутри группы.

Упростим идею. Вместо групп пользовательских процессов будем рассматривать N независимых процессов, каждый из которых претендует на $T_i = \frac{1}{N}$ долю процессорного времени. Далее используем схему, изложенную выше, относительно процессов, а не пользовательских групп. Реализовать эту схему гораздо проще за счет использования таблицы процессов. В записи таблицы будем хранить информацию о времени создания процесса, а также суммарное время, в течение которого он выполнялся на ЦП. На основании этого нетрудно вычислить долю использованного процессорного времени. Суммарное время выполнения процесса будем редактировать при смене контекста на основании данных системного таймера и отображать результат на соответствующую этому процессу запись в таблице процессов.

1.5.4 Приоритетные алгоритмы планирования.

В алгоритмах работы планировщика, рассмотренных до этого, неоднократно затрагивалось понятие приоритета процесса. Построение гибкая системы приоритетов позволяет улучшить работу планировщика [1] [2] .

Пусть планировщик выбирает процесс с наибольшим приоритетом. На каждом прерывании системного таймера будем понижать приоритет исполняющегося процесса на некоторую фиксированную величину. Когда приоритет текущего процесса станет меньше наибольшего приоритета процессов из очереди, сменим текущий процесс. Понятно, что в конце концов приоритеты процессов станут не положительными. После того,как это произошло для отдельного процесса, он больше не будет участвовать в борьбе за процессорное время.

Чтобы "перезапустить"планирование, присвоим всем процессам их изначальные приоритеты и продолжим выполнение с процесса с наивысшим приоритетом. Минусы такого подхода в объеме времени, затраченном на планирование и высоком оборотном времени для процессов с изначально низким приоритетом.Более того, для процессов, ограниченных скоростью работы устройств ввода-вывода, процессор нужно предоставлять немедленно после их выхода из состояния блокировки, иначе они будут также иметь высокое время оборота.

Введем для каждого процесса квант времени, в течение которого процесс может занимать ЦП, и разрешим системе задавать значение процесса, а не просто понижать его. Пусть процесс израсходовал q-ю часть своего кванта до перехода в режим блокировки, $q \in]0, 1]$. Если задать его приоритет p по формуле $p = \frac{1}{q}$, то процессы, перешедшие в режим блокировки вскоре после начала работы на ЦП, в том числе процессы, ограниченные скоростью работы устройств ввода-вывода, смогут получать процессорное время вскоре после выхода из состояния блокировки (их приоритеты будут достаточно высоки) [см. 2.2]. После того, как все процессы исчерпают свои кванты, обновим их и продолжим выполнение с самого приоритетного процесса.

Сгруппируем процессы. За каждой группой закрепим подотрезок упорядоченного списка значений приоритетов в ОС (так, чтобы пересечение подотрезков было пустым множеством и в то же время разбиение полностью покрывало исходное множество приоритетов) и назначим процессы в соответствующие им группы. Теперь приоритетное планирование (в какой либо из форм) будет применяться к группам, а внутри группы можно использовать другой способ планирования. Остается отметить, что при группировке процессов необходимо менять приоритеты групп так, чтобы группам с низким приоритетом также доставалось процессорное время.

Глава 2. Алгоритмы планирования в некоторых ОС

2.1 Linux

2.1.1 До версии ядра Linux 2.6

До выхода ядра версии 2.6 Linux системы использовали планировщик задач, который назывался **O(n)** [9]. Все процессы хранились в общей очереди, а процессорное время делилось на ”эпохи”. Каждый из процессов мог выполняться в течение присвоенного ему кванта времени. Если процесс не использовал свой квант до конца и досрочно перешел в состояние блокировки, то наследующей эпохе, когда ему выделится ЦП, этот процесс получит дополнительное время. Если процесс квант израсходовал полностью, то на следующей эпохе его квант обновится.

Достоинства:

— Простота понимания и реализации.

— Попытка компенсировать процессам, ограниченным скоростью работы устройств ввода-вывода, не израсходованное процессорное время.

Недостатки:

— Линейное от количества процессов в системе время работы. Алгоритм, чтобы выбрать новый процесс для исполнения, просматривает все процессы в очереди, чтобы найти процесс, который еще не выполнялся на данной эпохе. Таким образом, при большом числе процессов планировщик тратит много процессорного времени на потребности планирования вместо полезной работы, а это плохо.

— В многопроцессорных системах при алгоритме планирования ”O(n)” процесс мог быть назначен на исполнение любому из процессоров. Это приводило к проблеме ”холодного старта”: задача назначена на один процессор, а ее данные находятся в кэше процессора, на котором она выполнялась до этого. Данные необходимо переместить, а это требует дополнительных затрат. Проблему можно частично решить, сохраняя информацию о процессоре, на котором исполнялся процесс, в дескрипторе процесса, затем назначать процесс на тот же процессор. Однако, такой подход серьезно усложнит алгоритм (ожидание освобождение ”своего” процессора).

— В многопроцессорных системах появляется проблема синхронизации процессоров: несколько процессоров одновременно требуют доступа к очереди процессов. Их работу необходимо согласовывать с помощью объектов синхронизации, в которые помещается очередь процессов. Такой подход не является эффективным: пока один процессор работает с очередью, что занимает в худшем случае O(n) времени, остальные процессоры могут простаивать и не выполняют полезной работы.

В силу изложенных выше недостатков стала очевидна необходимость совершенствования алгоритма планирования, что реализовывалось в последующих версиях ядра.

2.1.2 Версия ядра Linux 2.6

В этой версии ядра на смену алгоритму "O(n)" пришел алгоритм "O(1)" [9]. Алгоритм является дальнейшим развитием модификации циклической схемы [см. 1.5.1].
Присвоим каждому процессу квант времени и приоритет: чем меньше числовое значение, тем выше приоритет. За процессором закрепим структуру LIL, обозначим ее $L_{Active} = L_A$, а ее элементы - $L_{A_i}, i = 0..139$. Каждый L_{A_i} будет хранить процессы с приоритетом i, которые не исчерпали свой квант. Аналогично введем структуру LIL с именем $L_{Blocked} = L_B$, в которой будем хранить классы приоритетов с процессами, исчерпавшими свой квант.

Пусть $bit - array$ - битовый массив, такой, что его бит с номером i принимает значение 1, если L_{A_i} не пуст, иначе 0.
При планировании будет выбираться не пустой $L_{A_i} : i \to min$ (класс с наивысшим приоритетом), а внутри класса применяется циклическая схема. Сделать такой выбор можно с помощью применения операции $find - first - bit - set$ к $bit - array$, которую поддерживает большинство архитектур процессоров. После перехода активного процесса в состояние блокировки происходит пересчет его кванта и приоритета. В зависимости от значения кванта, процесс будет помещаться в одну из структур L_A, L_B.Если процесс помещен в L_B, сразу обновим его квант. Но совсем не обязательно, что процесс попадет в ячейку с тем же номером, что и ранее: планировщик учитывает специфику процесса (например, как быстро после активации перешел в состояние блокировки) и соответствующим образом изменяет его класс приоритета (справедливо только для пользовательских задач, классы реального времени не затрагиваются). Также применяется механизм вытеснения: если при выполнении процесса из некоторой группы с приоритетом i, в группе с приоритетом j появится готовый к выполнению процесс и при этом $i > j$ (приоритет j выше i), то произойдет досрочная замена исполняющегося процесса

на более приоритетный (вытеснение процесса). Когда все L_{A_i} опустеют, поменяем L_A, L_B местами и продолжим планирование.
Недостатки:

— Процессы из групп с низким приоритетом могут ”голодать” , если в системе много процессов с более высоким приоритетом. Как отмечалось, планировщик может изменять приоритеты процессов, тогда этот недостаток можно частично компенсировать понижением приоритетов остальных процессов или повышением приоритета данной группы. Как реализовать систему динамических приоритетов, будет показано позже [см. 2.2].

2.1.3 Версия ядра Linux 2.6.23

В версии ядра 2.6.23 введен алгоритм *Completely Fair Scheduler (CFS)* [8] [3], базой которого является упрощение ”справедливого” алгоритма [см. 1.5.3]. CFS дополнительно вводит понятие **минимальной детализации** - минимальный отрезок времени, в течение которого будет выполняться процесс. Это сделано для того, чтобы минимизировать издержки на смену контекста при увеличении числа процессов.
Главным недостатком такого подхода в пользовательской системе я считаю ”теоретически” недостаточный уровень интерактивности, так как процессорное время делится между всеми процессами поровну, как итог, оборотное время увеличивается. С другой стороны, CFS будет полезно использовать в ситуациях, когда нужно сбалансировать процессы по потреблению ресурсов (например, сервер, обрабатывающий пользовательские запросы). CFS решает проблему процессов, ограниченных скоростью работы устройств ввода-вывода, за счет того, что после выхода из блокировки такого процесса он вскоре получит процессорное время, поскольку его доля суммарного израсходованного времени будет меньше необходимой.

2.2 UNIX-BSD

В целом стандартный алгоритм планировщика в ядре 4.3BSD схож с алгоритмом ”O(1)”, некоторые различия приведены в Таблице 2. Также в этом разделе будут более подробно рассмотрены некоторые моменты, связанные с планированием, которые опускались до этого [2] .

Признак для сравнения	**4.3BSD**	**Linux 2.6**
Количество очередей для хранения процессов	32	140
Диапазон значений приоритетов	0-127	0-139
Очередь отвечает за номера приоритетов	4 подряд идущих с соответствующим смещением	соответствует номеру очереди
Распределение приоритетов по назначению процесса	0-49 - для ядра, 50-127 - прикладные процессы	Первые 100 - задачи реального времени, последние 40 - пользовательские задачи
Функция нахождения первого ненулевого бита	Присутствует для 32-битного числа.	Для битового массива

Таблица 2: **Некоторые различия в реализации алгоритма O(1) в системах 4.3BSD и Linux 2.6**

Раньше неоднократно замечалось , что процессы, ограниченные скоростью работы устройств ввода-вывода,после выхода из состояния блокировки должны как можно скорее получить ЦП в свое пользование. Рассмотрим механизм, с помощью которого обеспечивается такое подведение на примере 4.3BSD.
Каждому событию, ресурсу, системному вызову поставим в соответствие так называемый **приоритет сна**, величина которого лежит в диапазоне приоритетов ядра (0-49). После того, как процесс выходит из состояния блокировки, система его приоритет приравнивает к приоритету сна ресурса, события, системного вызова, с которым работал процесс. Так как приоритеты ядра выше всех остальных приоритетов, то такие процессы будут приняты к исполнению раньше всех остальных пользовательских процессов. После этого приоритет процесса сбрасывается к первоначальному , и процесс участвует в планировании наравне с остальными до следующей блокировки.
На приоритет процесса влияет не только степень использования последнего кванта времени, но и **значение любезности**. Назначение этой характеристики - задание степени ”уступчивости” процессорного времени конкретным процессом. Стандартное значение любезности для процесса - 20 при диапазоне значений от 0 до 39.
Так как системы 4.3BSD и Linux являются UNIX-подобными и часто используются в качестве серверных ОС, то их можно позиционировать как **системы разделения времени** [см. определение], для которых важно условное равенство всех задач. Как я уже отмечал [см. 2.1.3], эти системы динамически изменяют приоритеты процессов, чтобы процессы разделяли ЦП примерно поровну. Вводится понятие **фактора полураспада**, значение которого обозначим переменной $decay$.

В 4.3BSD он вычисляется по формуле:

$$decay = \frac{2 * load\_av}{2 * load\_av + 1}$$

Здесь $load\_av$ - среднее количество процессов, готовых к выполнению, за последнюю секунду (пересчет запускается раз в секунду). К разных системах формулы для расчета могут отличаться, вплоть до использования константного значения. Фактор полураспада $decay$ влияет на процессорное время, выделяемое процессу, в зависимости от загруженности системы. Если загрузка велика, то значение $decay$ расположено близко к единице, и он не играет существенной роли. Если же готовых процессов не много, то влияние коэффициента будет ощутимо - снижение приоритетов будет происходить быстро, что решает проблему процессов с изначально низким приоритетом (”голодание”), так как они получат процессорное время после снижения приоритетов более предпочтительных процессов.
Приоритет процесса usr_proc_pr пересчитывается по формуле:

$$usr\_proc\_pr = USER\_DEFAULT\_PR + \frac{cpu\_usage}{4} + 2 * nice\_value$$

Здесь $USER\_DEFAULT\_PR$ - стандартное значение приоритета для пользовательских задач, в 4.3BSD равно 50, cpu_usage - результат последнего измерения использования процессорного времени, $nice\_value$ - значение любезности для текущего процесса. Можно сделать следующие выводы:

— Чем больше процесс использовал ЦП в последний раз (cpu_usage), тем выше будет числовое значение приоритета, тем ниже фактический приоритет процесса.

— Чем выше значение ”любезности”, тем выше числовое значение приоритета, тем ниже фактический приоритет. Значение любезности можно добровольно понижать, заложив такую логику в исходном коде программы или использовав утилиту $renice$ терминала.

— Нет необходимости обновлять приоритеты обходом списка процессов, эту задачу решает использование формулы.

Таким образом, планировщик в 4.3BSD является достаточно эффективным для средне нагруженных систем, но все же имеет несколько ограничений:

— При увеличении количества процессов пересчет $decay$ каждую секунду станет не выгоден.

— Не дает гарантий по времени реакции для приложений реального времени.

— Реализовано ”мягкое” вытеснение: если система функционирует в режиме ядра , то процесс будет вытеснен только при переходе обратно в пользовательский режим.

2.3 Windows

В Windows начиная с версии NT используется алгоритм, схожий с рассмотренным в Linux и 4.3BSD Unix. Главное различие - в системе приоритетов [4] [5] . Система классов приоритетов **процессов** описана в Таблице 3

Название класса приоритетов	**Краткое описание**
Idle	Процесс выполняется, когда система не занята другой работой. Подходит для фоновой работы
Below normal	Промежуточный класс
Normal	Основной класс приоритетов, который используют большинство приложений
Above normal	Промежуточный класс
High	Процесс с приоритетами из этого класса обязан немедленно реагировать на события
Real-time	Приоритеты процессов реального времени, такие процессы имеют право вытеснять даже компоненты ОС

Таблица 3: **Классы приоритетов процессов в Windows**

С классом приоритетов процесса связаны приоритеты его потоков. После присвоения процессу класса приоритетов, необходимо указать относительные приоритеты потоков (Таблица 4) в пределах процесса, которому поток принадлежит. Числовое значение приоритета формируется самой системой исходя из класса приоритетов процесса и относительного приоритета потока. Числовые значения часто меняются в процессе развития и доработки системы.

Относительный приоритет потока	**Краткое описание**
Idle	Приоритет потока равен 16 в классе real-time и 1 в остальных классах
Lowest; Below normal; Normal; Above Normal	К обычному приоритету для класса прибавляется значение с соответствующим номером из списка [-2; -1; 0; 1; 2]
Time-critical	Приоритет потока равен 31 в классе real-time и 15 в других процессах

Таблица 4: **Относительные приоритеты потоков в Windows**

Глава 3. Практическая часть

3.1 Формулировка задачи

Рассмотрим следующую задачу. Пусть имеется некоторое WebAPI,в котором для каждого запроса, приходящего в API, фиксируется время поступления, во время обработки собирается другая служебная информация про запрос, а при завершении обработки запроса - фиксируется время завершения. После этого получившаяся информация для этого запроса записывается в лог-файл. С наступлением очередного дня для логирования создается новый файл, в который перенаправляется вывод логов. Ставится задача отсортировать лог-файлы по времени поступления запросов. Одним из возможных способов решения поставленной задачи является использование утилит командной строки (например, команда sort в UNIX). Однако, дата поступления процесса может не находиться в начале строки, поэтому лексикографическая сортировка может не принести желаемых результатов. Более того, время и дата могут записываться в различных форматах в зависимости от локальных настроек сервера. Поэтому возникает необходимость написать программу, назначение которой - сортировка с конкретным предикатом. Таким образом, при изменении формата логирования на сервере или цели сортировки будет необходимо изменить только правило сортировки, а не переписывать всю программу.
Замечание. В реализации сделано упрощение: для простоты происходит сортировка строк длины 20, чтобы смоделировать издержки на разбор и преобразование строк в даты исходных лог-файлов.
В следующих разделах будут рассмотрены некоторые детали реализации программной части в различных операционных системах с приведением данных о времени работы программы.

3.2 Общее описание решения

Так как содержимое файлов может быть достаточно объемным, сортировка может заняться существенное время. Далее предполагается, что содержимое файлов полностью помещается в оперативную память компьютера, то есть отсутствует необходимость применять алгоритмы внешней сортировки. Чтобы уменьшить предполагаемое время сортировки, воспользуемся псевдо-параллельной обработкой. В задаче явно прослеживаются независимые друг от друга фрагменты: файлы, которые должны быть отсортированы, являются независимыми по отношению друг к другу. Исходя из этого, рассмотрим следующий алгоритм:

— Процесс, назначение которого - непосредственная сортировка, будем называть $Sorter$.

— Главный процесс назовем $Distributor$, его назначение - порождение группы обработчиков $Sorter$ и передача им параметров, необходимых для распределения нагрузки внутри рабочей группы.

— $Distributor$ принимает от пользователя количество файлов (для простоты), которые нужно отсортировать, а также количество процессов типа $Sorter$ в рабочей группе.

— $Distributor$ порождает процессы из рабочей группы, в каждый из них передается порядковый номер процесса и все параметры, которые $Distributor$ получил сам.

— Каждый из процессов $Sorter$ принимает переданные параметры и на их основе выбирает назначенные ему файлы, читает из каждого данные, сортирует их, записывает результат в новый файл, возвращает код завершения.

— $Distributor$ ожидает завершения всех порожденных процессов, сообщая пользователю об ошибках, если таковые возникли в дочерних процессах. Убедившись, что все процессы из рабочей группы успешно завершили работу, $Distributor$ прекращает свое выполнение.

Генерация входных данных осуществляется с помощью Python-скрипта, вызов которого имеет формат:

```
python3 input_generator.py --count N --length M OutputPath
```

Пример входных данных:

```
6XWBQJLGNBCU8HCFXZWKVYD2P9MQ6D
QMR2P3EEFWYJNEG140NHUN0M52500H
OFMCCC0R44YVNN83N7M0NEAZ9P30FK
```

Реализация $Sorter$ одинакова для систем Unix, Linux, Windwos и для сортировки использует C++ STL. Различия в реализации в перечисленных выше ОС появляются в $Distributor$, поскольку разные ОС предоставляют разные API для выполнения системных вызовов.

3.3 Некоторые особенности реализации Distributor в Windows

Запуск нового процесса происходит с помощью системного вызова $CreateProcess$ (Листинг 1).

```
CreateProcess(executable_name, const_cast<char*>(kCmdArgs),
                NULL, NULL, FALSE, 0, NULL,
```

```
NULL, &StartupInfo, &ProcessInfo))
```

Листинг 1: Запуск процессов

Системный вызов возвращает handle процесса, который записывается в массив и хранится для ожидания завершения каждого дочернего процесса с последующим закрытием их дескрипторов (Листинг 2).

```
WaitForMultipleObjects(PROC_COUNT, &proc_info[0],
                                                TRUE, INFINITE);
for (auto i = 0; i < PROC_COUNT; ++i) {
        CloseHandle(proc_info[i]);
}
```

Листинг 2: Ожидание и закрытие дескрипторов процессов

Здесь $proc\_info$ - массив, в котором хранятся handle процессов.

3.4 Некоторые особенности реализации Distributor в Unix/Linux

Порождение нового процесса в ОС Linux/Unix происходит с помощью системного вызова $fork()$ [3]. Порожденный процесс является практически полной копией родительского процесса и почти сразу начинает свое выполнение. Затем , с помощью семейства системных вызовов $exec()$ [3] происходит замещение текущего содержания адресного пространства, и процесс готов к выполнению новой программы.
Пусть вызвана операция $fork()$. При успешном создании нового процесса оба из них продолжают выполнение от вызова $fork()$ (Пример из исходного кода приведен в Листинге 3.).

```
pid_t current_pid = fork();
int ret_code = execl(
        "sorter","sorter", CMD_ARGS, (char*)0
);
```

Листинг 3: Запуск процессов в Linux/Unix

Однако они получат различные коды возврата, которые являются текущим pid процесса. В дочернем процессе успешный вызов $fork()$ вернет 0, в родительском процессе - pid дочернего. С помощью возврата функции ветвления можно определять дальнейшее поведение родительского и дочернего процессов.
Функции из семейства $exec()$ замещают текущий образ новым, загружая в память программу и аргументы, определяемые параметрами системного вызова. В своей реализации я использовал системный вызов $execl()$, поскольку он позволяет передавать несколько параметров командной строки, что мне и требуется.

После того, как *Distributor* распределил нагрузку, он, как и в реализации под Windows, ждет завершения дочерних процессов (т.е. переходит в состояние блокировки). Пример кода приведен в Листинге 4.

```
while (-1 == waitpid(pids[i], &status, 0));
if (!WIFEXITED(status) || WEXITSTATUS(status) != 0) {
        std::cout << ERROR_MSG;
        exit(1);
}
```

Листинг 4: Ожидание завершения процессов в Linux/Unix

3.5 Тестирование программы в FreeBSD 11.0, Linux Ubuntu 18.04 LTS, Windows 10

ОС FreeBSD и Linux Ubuntu запускались на виртуальной машине Oracle VM VirtualBox в конфигурации : 2 ядра, 1 поток, 2048 МБ оперативной памяти, остальные настройки по умолчанию. Система Windows 10 установлена в качестве основной системы на компьютере. ЦП : Intel Core i7-4500U (2 ядра, 4 потока), 8Гб ОЗУ. Результаты экспериментов представлены в Таблицах 5, 6, 8, 7, 9, 10.

	Число файлов	5	10	20	50
Число строк					
100		15ms	16ms	25ms	50ms
1000		40ms	79ms	146ms	433ms
10000		412ms	796ms	1556ms	3973ms
100000		4196ms	8500ms	17804ms	42490ms

Таблица 5: **Тесты в ОС FreeBSD при 1 процессе** *Sorter*

	Число файлов	5	10	20	50
Число строк					
100		20ms	23ms	22ms	35ms
1000		33ms	60ms	129ms	308ms
10000		264ms	602ms	1191ms	2975ms
100000		2764ms	5097ms	10750ms	25129ms

Таблица 6: **Тесты в ОС FreeBSD при 2 процессах** *Sorter*

	Число файлов	5	10	20	50
Число строк					
100		6ms	8ms	16ms	32ms
1000		27ms	51ms	101ms	212ms
10000		264ms	452ms	893ms	2308ms
100000		2482ms	4939ms	9565ms	24121ms

Таблица 7: **Тесты в ОС Ubuntu при 1 процессе** $Sorter$

	Число файлов	5	10	20	50
Число строк					
100		4ms	8ms	14ms	24ms
1000		17ms	26ms	47ms	143ms
10000		222ms	339ms	530ms	1290ms
100000		1747ms	2792ms	5474ms	13472ms

Таблица 8: **Тесты в ОС Ubuntu при 2 процессах** $Sorter$

Поскольку системы FreeBSD и Ubuntu недавно были установлены, то в них не было достаточного количества активных процессов, чтобы протестировать работу программы в условиях конкуренции со стороны остальных активных процессов системы. Как следствие, изменение значения (как увеличение, так и уменьшение) $nice\_value$ для процессов типа $Sorter$ значимых изменений в Таблицы 5, 6, 7, 8 не внесло. С точки зрения логики, при увеличении числа процессов в k раз ожидается и уменьшение времени выполнения в k раз. Однако это не так. Потеря во времени связана с конвейерной обработкой команд, затратами на работу планировщика процессов, на переключение контекстов процессов и прочими факторами. Особенно явно эту проблему можно наблюдать при сравнении данных из таблиц 9, 10.

	Число файлов	5	10	20	50
Число строк					
100		45ms	61ms	74ms	146ms
1000		79ms	126ms	222ms	690ms
10000		236ms	378ms	871ms	1897ms
100000		927ms	1720ms	3784ms	9914ms

Таблица 9: **Тесты в ОС Windows 10 при 1 процессе** $Sorter$

	Число файлов	5	10	20	50
Число строк					
100		142ms	121ms	150ms	183ms
1000		157ms	214ms	299ms	579ms
10000		287ms	434ms	1019ms	1110ms
100000		850ms	1125ms	2147ms	5120ms

Таблица 10: **Тесты в ОС Windows 10 при 4 процессах** $Sorter$

Таким образом, при больших объемах и количествах файлов выгоднее использовать обработку несколькими процессами вместо последовательной обработки. Исходя из результатов тестирования, время выполнения в ОС Windows 10 удалось уменьшить примерно в 2 раза при стандартных значениях приоритетов. Были предприняты попытки изменять класс приоритетов процессов в ОС Windows 10, однако время выполнения сократить не удалось. Предлагаю следующее объяснение:
”Узким местом” реализации является чтение текстовых файлов с жесткого диска компьютера. В языке C++ используется буферизированный потоковый ввод, то есть файл читается не целиком, а кэшируется некоторая его часть. В реализации после чтения очередной строки происходит ее добавление в $vector$,после чего считывается следующая строка и так далее, то есть устройство ввода-вывода используется очень интенсивно, даже несмотря на кэширование данных. Как отмечалось выше, для хранения содержимого файлов используется контейнер $vector$, причем данные в него добавляются с помощью операции $push\_back(T)$. Проведем еще одну серию тестов, где вместо $vector$ будет использован $deque$. Результаты отражены в Таблице 11

	Число файлов	5	10	20	50
Тип контейнера, число процессов					
vector, 1 Sorter		927ms	1720ms	3784ms	9914ms
deque, 1 Sorter		1359ms	2455ms	4652ms	12603ms
vector, 4 Sorters		850ms	1125ms	2147ms	5120ms
deque, 4 Sorters		765ms	1242ms	2315ms	5322ms

Таблица 11: **Тесты в ОС Windows 10 при 4 процессах** $Sorter$

Полученные результаты косвенно подтверждают, что самая дорогостоящая по времени часть программы - именно ввод-вывод. Возможно, поэтому изменение класса приоритета для процессов типа $Sorter$ существенно не влияют на итоговое время работы программы.

ЗАКЛЮЧЕНИЕ

Таким образом, в рамках данной работы было выполнено следующее:

1. Изучены основные подходы и концепции в системах планирования процессов.
2. Рассмотрены различные алгоритмы планирования: EDF, FIFO, SJN, SJF, Fair, циклическая схема.
3. Рассмотрены алгоритмы планирования в ОС Windows, Unix, Linux. Приведены некоторые сходства и различия, достоинства и недостатки.
4. В ОС FreeBSD 11, Ubuntu 18.04 LTS, Windows 10 реализовано приложение, демонстрирующее выгоду от использования многопроцессной обработки данных в сравнении с последовательным выполнением (1 процесс типа $Sorter$).

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Таненбаум Э., Современные операционные системы. 3-е изд., СПб, 2010г., ISBN 978-5-49807-306-4 [Тип ресурса?]/ –Режим доступа: ? – Дата доступа: 28.11.2018

[2] Вахалия Ю., Unix изнутри, СПб, 2003г., ISBN 5-94723-013-5 – Дата доступа: 06.12.2018

[3] Лав Р., Linux. Системное программирование. 2-е изд., СПб, 2014г., ISBN 978-5-496-00747-4 – Дата доступа: 04.12.2018

[4] Рихтер Дж., Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows, 4-е изд., СПб, 2001г., ISBN 5-272-00384-5 – Дата доступа: 06.12.2018

[5] Харт, Джонсон, М., Системное программирование в среде Windows, 3-е издание, М., 2005г., ISBN 5-8459-0879-5 – Дата доступа: 06.12.2018

[6] Назар К., Рихтер Дж., Windows via C/C++. Программирование на языке Visual C++, СПб, 2009г., ISBN 978-5-7502-0367-3 – Дата доступа: 07.12.2018

[7] Ubuntu Manpage: sched [Электронный ресурс]/ –Режим доступа: http://manpages.ubuntu.com/manpages/cosmic/man7/sched.7.html – Дата доступа: 04.12.2018

[8] Inside the Linux 2.6 Completely Fair Scheduler [Электронный ресурс]/ –Режим доступа: https://developer.ibm.com/tutorials/l-completely-fair-scheduler/ – Дата доступа: 10.12.2018

[9] Планировщик задач Linux [Электронный ресурс]/ –Режим доступа: https://www.ibm.com/developerworks/ru/library/l-scheduler/index.html – Дата доступа: 02.12.2018

[10] Silberschatz et al., Applied Operating System Concepts, 9th ed, 2013. [Электронный ресурс]/ –Режим доступа: http://iips.icci.edu.iq/images/exam/Abraham-Silberschatz-Operating-System-Concepts—9th2012.12.pdf – Дата доступа: 30.11.2018